Politechnika Wrocławska

Wydział Informatyki i Zarządzania
kierunek studiów: Informatyka

Praca dyplomowa – inżynierska

Gra z gatunku RPG jako aplikacja Webowa

Drozdowski Karol
Dwojak Jakub

słowa kluczowe:
Gra
Aplikacja
Webowa

krótkie streszczenie:
Gra z gatunku RPG, tworzona na przeglądarkę internetową, przeznaczona na komputery osobiste. Każdy użytkownik ma możliwość założenia swojej postaci. Podstawowym elementem pełniącym kluczową rolę w grze będzie mapa, na której będą zachodziły wszelkie interakcje tj. walki z przeciwnikami. Za pokonanych przeciwników postać zostanie wynagrodzona punktami doświadczenia, które to będą wymieniane na poziomy, a te natomiast na punkty dzięki którym nasz bohater będzie nabywać nowych umiejętności.

opiekun pracy dyplomowej	Dr Marek Kopel	.....	.....
	Tytuł/stopień naukowy/imię i nazwisko	*ocena*	*podpis*
Ostateczna ocena za pracę dyplomową			
Przewodniczący Komisji egzaminu dyplomowego	.....	.....	.....
	Tytuł/stopień naukowy/imię i nazwisko	*ocena*	*podpis*

*Do celów archiwalnych pracę dyplomową zakwalifikowano do:**

a) kategorii A (akta wieczyste)

b) kategorii BE 50 (po 50 latach podlegające ekspertyzie)

** niepotrzebne skreślić*

pieczątka wydziałowa

Wrocław
2019

Streszczenie

Głównym celem pracy jest projekt i implementacja gry z gatunku RPG jako aplikacji webowej. Pobocznymi natomiast są poznanie technologii oraz wzbogacenie wiedzy dotyczącej tworzenia gier komputerowych. Do pozyskania jej wykonany jest przegląd dziedzinowy oraz analiza wymagań. W pracy opisane są wykorzystywane technologie w aplikacji. W oparciu o zebraną wiedzę, napisany jest projekt oraz implementacja. Przedstawione są problemy związane z komunikacją między użytkownikami w czasie rzeczywistym oraz procesem realizacji gry komputerowej wraz z rozwiązaniami. Do osiągnięcia głównego celu, niezbędny jest pełny przegląd dziedzinowy oraz wiedza techniczna. Są one wymagane, ponieważ temat pracy dotyka wielu płaszczyzn powiązanych z technologiami webowymi.

Abstract

The main purpose of the work is the design and implementation of the RPG game as a web application. Secondary reasons are learning about web technologies and enriching knowledge about creating computer games. To obtain it, a domain review and requirements analysis need to be covered. The paper describes the technologies used in the application. Based on the collected knowledge, the design and implementation are written. Problems related to communication between users in real time and the process of implementing a computer game are described and solved. To achieve the main goal, an entire domain review and technical knowledge are necessary. Those are obligatory because the topic of the thesis is related to many parts of web technologies.

Spis Treści

Spis pojęć

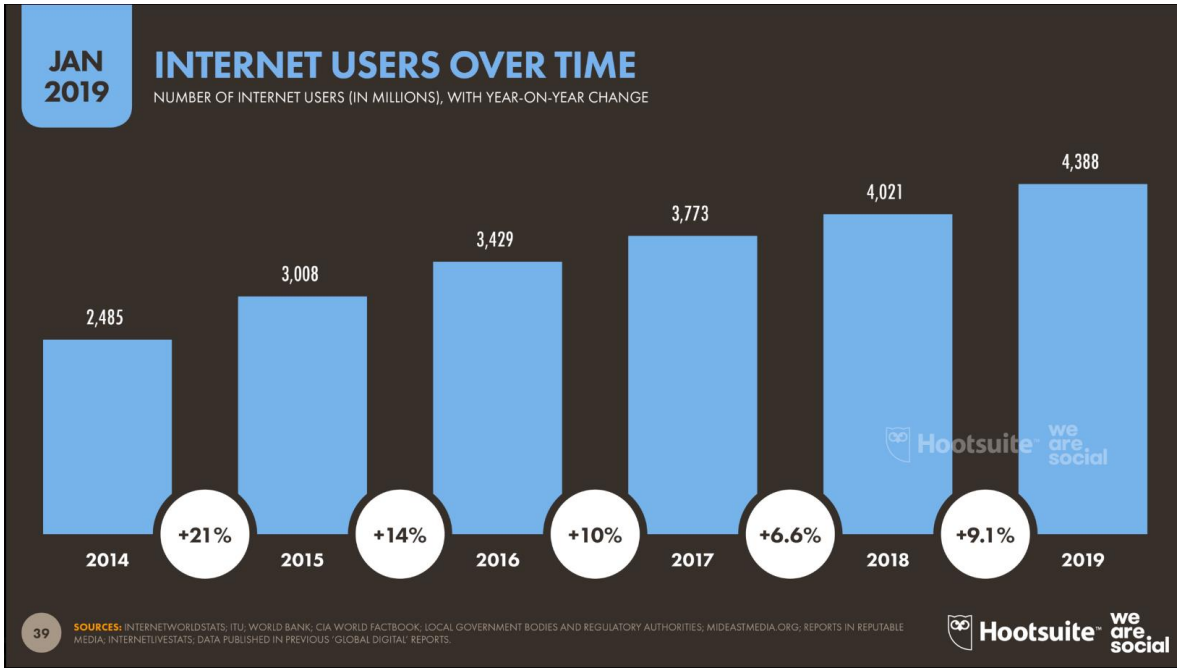
Baza danych	Zbiór danych zapisanych wedle ściśle określonych reguł.
Biblioteka Programistyczna	Plik, który zawiera gotowe do wykorzystania dane, funkcje oraz klasy. Może zostać łatwo wykorzystany przez programistę w kodzie źródłowym. Mocno przyspiesza implementację.
Ekwipowany przedmiot	Przedmiot jest określany mianem ekwipowanego po spełnieniu wszystkich jego warunków oraz oznaczeniu go tym statusem przez gracza. Ekwipowany przedmiot zmienia statystyki użytkownika w zależności od rodzaju przedmiotu.
Endpoint	Interfejs dostarczony przez aplikację, umożliwiający komunikację z nią.
Framework	Zbiór bibliotek programistycznych. Biblioteki wchodzące w jego skład związane są zazwyczaj z jedną tematyką.
Gracz	Zalogowany użytkownik w aplikacji.
Indeks	Wartość liczbowa wskazująca na konkretny element w strukturze danych.
Klient	Program komputerowy uruchomiony na maszynie użytkownika korzystającego z usług aplikacji. Klient jest twórcą żądań, które są następnie przesyłane do serwera.
Layout	Układ elementów w aplikacji definiujący jej wygląd.
Mana	Inaczej energia potrzebna do wykonywania czynności innych niż fizyczne.
Maszyna Wirtualna	Oprogramowanie działające jako emulator innej maszyny. Wykorzystywany najczęściej do testowania funkcjonalności na innych systemach operacyjnych niż ten, na którym jest uruchamiana maszyna wirtualna.
Oprogramowanie	Zapisane informacje w postaci instrukcji, interfejsów, klas i danych w odpowiednio czytelnym dla komputera formacie. Oprogramowanie jest produktem programistów.
Poziom	Wartość liczbowa odpowiadająca rozwoju postaci analogicznie do punktów doświadczenia, lecz w bardziej ogólnym stopniu. Punkty doświadczenia składają się na poziomy

Przedmiot	Obiekt w grze posiadający zdefiniowane cechy określające jego zastosowanie. Często by móc go ekwipować, muszą zostać spełnione pewne wymagania.
Punkty Doświadczenia	Wartość liczbowa odpowiadająca rozwoju postaci.
Punkty many	Wartość liczbowa opisująca aktualny stan many gracza w grze.
Punkty życia	Wartość liczbowa opisująca aktualny stan zdrowia gracza w grze.
Relacja	Związek między danymi w relacyjnej bazie danych.
RPG	Skrót od *Role Playing Game.* Gatunek gier, polegający na wcielaniu się w postać.
Serwer	Program komputerowy obsługujący żądania innych programów zwanych klientami. Programy kliencie zazwyczaj uruchomione są na innych, zewnętrznych maszynach.
SSL	Secure Socket Layer - Protokół wykorzystywany do bezpiecznej transmisji danych.
Tilemap	Obraz podzielony na siatkę o zdefiniowanym rozmiarze. Jest wykorzystywany w celach tworzenia widoków do gier 2D.
Umiejętność	Dodatkowa funkcjonalność jaką może wykorzystać gracz. Początkowo zablokowana. W celu odblokowania umiejętności, gracz musi spełnić określone warunki.
URL	Adres strony internetowej w sieci www.
VPS	Virtual Private Server – wirtualna maszyna, udostępniona prywatnemu użytkownikowi.
XML	Język znaczników przeznaczony do prezentacji danych w postaci struktur.

Wstęp

Gra komputerowa, to aplikacja, która ma na celu dostarczenia rozrywki użytkownikowi. Może zostać skategoryzowana na podstawie rozgrywki. Przykładem takiej kategorii jest RPG. Polega ona na kontrolowaniu postaci, która jest rozwijana wraz z czasem. Gra uruchamiana na przeglądarkach internetowych nazywana jest webową. Istnieje możliwość połączenia powyższych cech. W rezultacie otrzymuje się grę z gatunku RPG jako aplikację webową.

W dzisiejszych czasach popularność Internetu wzrasta (Rys. 0.1). W związku z tym każda dziedzina w coraz większym stopniu korzysta z sieci. Celem jest dotarcie do szerszej grupy odbiorców. Do takich dziedzin należy między innymi rynek gier wideo. Wraz ze zwiększeniem grupy odbiorców następuje wzrost liczby zainteresowanych. Gatunek RPG jest na tyle uniwersalny, że można dopasować go do dużej liczby graczy. Jest to motywacją do wdrożenia własnego rozwiązania spełniającego powyższe kryteria.



Rys. 1.1 Wykres przedstawiający wzrost użytkowników korzystających z Internetu na przestrzeni lat 2014-2019 [1]

Głównym celem pracy jest projekt i implementacja takiego rozwiązania, które może być określone grą z gatunku RPG jako aplikacja webowa. Zagadnienie to wymaga znajomości wielu skomplikowanych technologii oraz wiedzy dziedzinowej. Prowadzi to do kolejnych celów pobocznych, którymi są:

- Poznanie technologii webowych
- Wzbogacenie wiedzy dotyczącej tworzenia gier komputerowych

Praca podzielona jest na część analizy wymagań, projektu oraz implementacji. Pierwsza część dotyczy określenia funkcjonalności potrzebnych do poprawnego działania aplikacji. W części projektowej zawarty jest szczegółowy zarys produktu końcowego wraz z dokładnym opisem technicznym. Implementacja obejmuje informacje na temat realizacji projektu.

Na początku części projektowej opisane są użyte technologie. Następnie podane są informacje na temat architektury aplikacji. W dalszej części znajdują się opisy struktury danych. Na końcu zawarty jest interfejs użytkownika wraz z diagramem przypadków użycia.

Część implementacyjna zaczyna się opisem struktury danych oraz zarządzania użytkownikami w systemie. Następnie zawarta jest wizualizacja oraz synchronizacja danych pomiędzy klientem i serwerem. W dalszej części, opisana jest komunikacja między graczami w postaci czatu. Całość zakończona jest informacjami o zastosowanym systemie walk oraz rozwoju postaci

Praca jest pisana przez dwie osoby. Rozdziały i podrozdziały zawierają oznaczenia:

- [J.D.] – Część pisana przez Jakuba Dwojaka
- [K.D.] – Część pisana przez Karola Drozdowskiego
- [K.D., J.D.] – Część pisana wspólnie

1. Analiza wymagań [K.D., J.D.]

W celu pełnego zrozumienia funkcjonalności, które zaspokajają potrzeby grupy docelowej aplikacji, dokonuje się odpowiednio analizy wymagań funkcjonalnych oraz niefunkcjonalnych. W obu przypadkach wypisane są nazwy, za pomocą których są rozpoznawane wymagania oraz ich krótki opis. Oprócz tego definiowany jest priorytet (Tab. 1.1) (Tab. 1.2), który wyznacza jak istotne jest to wymaganie. Zakładamy, że mniejsza wartość liczbowa oznacza wyższy priorytet.

Tab. 1.1 Tabela priorytetów wymagań funkcjonalnych

Wymaganie	Priorytet
FUN/001 – Użytkownik rejestruje się do gry.	1
FUN/002 – Użytkownik loguje się do gry.	2
FUN/003 – Użytkownik komunikuje się z pozostałymi graczami.	3
FUN/004 – Wyświetlana jest mapa oraz obiekty na niej.	3
FUN/005 – Użytkownik porusza się po mapie.	3
FUN/006 – Użytkownik odbywa walkę z przeciwnikami.	4
FUN/007 – Użytkownik rozwija swoją postać poprzez zdobywanie punktów Doświadczenia.	5
FUN/008 – System zapisuje stan gry w celu umożliwienia jej kontynuacji.	4

NFUN/001 – System obsługuje minimum 50 aktywnych użytkowników jednocześnie
NFUN/002 – Czas reakcji serwera nie może przekroczyć 100ms.

Tab. 1.2 Tabela wymagań niefunkcjonalnych

Wymaganie	Priorytet
NFUN/001 – System obsługuje minimum 50 aktywnych użytkowników jednocześnie	4
NFUN/002 – Czas reakcji serwera nie może przekroczyć 100ms.	4

2. Przegląd dziedzinowy [K.D., J.D.]

Praca jest napisana w oparciu o literaturę związaną z projektowaniem gier komputerowych [2], dobrymi praktykami pisania kodu w języku Java [3] [4] oraz książkami zawierającymi wiedzę dotyczącą poszczególnych zagadnień, wykorzystywanych w trakcie implementacji.

2.1. Główne technologie po stronie serwera [K.D.]

Do obsługi głównej logiki aplikacji jest stosowany język Java. Jest to język wykorzystujący paradygmat obiektowy. Powstał on jako następca języka C. Główne zalety tego języka to:
- Uproszczenie pisania kodu względem innych obiektowych języków programowania
- Możliwość programowania wielowątkowego
- Mnogość łatwo dostępnych bibliotek
- Działanie na wielu platformach
- Programowanie sieciowe
- Bezpieczeństwo aplikacji

O języku Java pisze w swoim dziele pt „Thinking in Java. Edycja polska. Wydanie IV" autor „Bruce Eckel":
„To, co wywarło na mnie największe wrażenie, kiedy poznawałem Javę, to fakt, że wśród innych celów projektantów z firmy Sun znalazła się także redukcja złożoności dla programisty. To tak jakby powiedzieć „Nie dbamy o nic poza zmniejszeniem czasu i trudności tworzenia porządnego kodu". W początkowej fazie dążenie to kończyło się otrzymaniem kodu, który niestety nie działał zbyt szybko (chociaż w przeszłości było wiele obietnic dotyczących zwiększenia szybkości działania programów), lecz zmniejszenie czasu pracy programisty było zdumiewające – potrzebował on teraz połowy lub nawet mniej czasu niż na napisanie równoważnego programu w C++. Przyczynia się to do dużych oszczędności, ale to nie wszystkie zalety Javy. Java idzie dalej, upraszczając wszystkie skomplikowane zadania, jak wielowątkowość i programowanie sieciowe, cechy języka lub biblioteki. Rozwiązuje także kilka naprawdę bardzo skomplikowanych problemów dotyczących programów działających na dowolnej platformie sprzętowej, dynamicznej zmiany kodu czy nawet bezpieczeństwa, które, na skali złożoności, można umieścić gdzieś pomiędzy „utrudnieniem" a „problemem nie do rozwiązania". Zatem pomimo problemów z wydajnością, widać, iż możliwości Javy są olbrzymie, przez co może ona uczynić z nas znacznie wydajniejszych programistów." [4].

Aplikacja tworzona w oparciu o ten język pozwala na szybsze stworzenie logiki umożliwiającej prawidłowe funkcjonowanie gry. Jest to główny powód, dla którego jest wybrany jako podstawowe narzędzie do pracy.

Większość funkcjonalności po stronie serwera opartych jest o technologię Spring Boot. Jest to popularny zbiór, który zawiera w sobie biblioteki napisane w języku Java. Pozwalają między innymi na:
- Proste i bezpieczne połączenie z bazą danych
- Szybką konfigurację całej aplikacji
- Prostą konfigurację logów aplikacji
- Posiada dużą, łatwo dostępną dokumentację
- Jest stale rozwijany
- Jest używany przez duże grono osób

O frameworku Spring Boot pisze Craig Walls w swoim dziele pt. „Spring in Action". "Jedną z najlepszych rzeczy jakie wykonuje Spring Boot jest fakt, iż automatycznie dostarcza fundamentalną hydraulikę aplikacji, pozostawiając Cię jako programistę, aby skupić się przede wszystkim na logice unikalnej dla Twojej aplikacji (tł. autor.)." [5].

Aplikacja tworzona w oparciu o ten zbiór, pozwala na skupienie się tylko na samych funkcjonalnościach gry, zamiast na jej fundamentach.

2.2. Główne technologie po stronie klienta [J.D.]

Do obsługi części aplikacji, po stronie klienta wykorzystywany jest język programowania JavaScript. Jest najlepiej zintegrowany z innymi technologiami internetowymi. Zawiera sporą liczbę bibliotek, które dają możliwość szybkiego budowania i działania aplikacji. Duże wsparcie społeczności pozwala na prostsze pozbywanie się potencjalnych błędów w aplikacji oraz zapobieganie ich ewentualnego powstawania. Potrzebne szczegóły korzystania z technologii można znaleźć w książce autorstwa Deborah Kurata [6].

Dużą część działań po stronie klienta odbywa się z udziałem technologii React. Jest to biblioteka wykorzystywana w języku programowania JavaScript. Pozwala na budowę aplikacji internetowej za pomocą komponentów. Szczegóły działania opisują w swoim dziele autorzy książki „Fullstack React: The Complete Guide to ReactJS and Friends" [7]. Dzięki React, treści na stronie internetowej odświeżane są tylko wtedy, gdy nastąpi zmiana stanu komponentu. W przeciwieństwie do konkurencyjnych technologii w tym zakresie, React jest przeznaczony do aplikacji, gdzie zawartość strony jest często odświeżana. W przypadku gry występuje takie właśnie zjawisko.

Jako silnik gry wykorzystywany jest technologia Phaser. Jest to framework do języka programowania JavaScript oparty na platformie HTML5, który dostarcza wielu gotowych funkcji do tworzenia gier webowych. Pozwala między innymi na obsługę ruchów postaci po utworzonej mapie. Najlepszym źródłem wiedzy jest książka zawierająca wszystkie elementy najnowszej wersji biblioteki [8].

2.3. Zarządzanie danymi [K.D.]

Główną rolę w zarządzaniu danymi w aplikacji pełni biblioteka Hibernate. Wchodzi ona między innymi w skład zbioru Spring Boot. Pozwala na proste połączenie z bazą danych i wykonywanie na niej operacji. Służy nie tylko do łączenia, lecz również do mapowania danych z relacyjnej bazy danych na świat obiektowy. Prawidłowe posługiwanie się tą biblioteką opisuje Christian Bauer wraz z Gavinem Kingiem w swoim dziele pt. „Hibernate w akcji" [9].

Dzięki wykorzystaniu Hibernate kod jest ujednolicony. Wszystko można zapisać obiektowo bez potrzeby przechodzenia pomiędzy kilkoma językami. Jako system zarządzania relacyjną bazą danych wybrany jest PostgreSQL ze względu na darmową licencję. Więcej zalet opisuje Gregory Smith w swoim dziele pt. „PostgreSQL 9.0 High Performance" [10].

Przydatnym narzędziem jest również Liquibase. To otwarte oprogramowanie służące do śledzenia, zarządzania i wdrażania zmian w schematach baz danych. Po technologii posługuje się językiem XML. Dzięki niej można między innymi tworzyć wszystkie tabele, kolumny, relacje, indeksy i tym podobne w kilku krokach. To narzędzie jest wybrane ze względu na jego prostotę oraz możliwości jakie oferuje.

Ze względu na potrzebę przechowywania danych również w pamięci podręcznej, wykorzystana jest technologia Hazelcast. Działa podobnie do relacyjnych baz danych. Oprócz przechowywania samych danych, oferuje również możliwość wykonywania na nich operacji. Żeby przyspieszyć pracę, dane przekształca się w obiekty i wrzuca do pamięci operacyjnej.
Jego głównymi zaletami są:
- Znaczne przyspieszenie działania aplikacji
- Bezpłatny dostęp do środowiska.
Szerzej o technologii pisze Mat Johns w dziele pt. „Getting Started with Hazelcast" [11].

2.4. Komunikacja w systemie [J.D.]

Najważniejsza komunikacja odbywa się poprzez technologię WebSocket, pozwala ona na dwukierunkową wymianę informacji za pośrednictwem jednego gniazda TCP. W wymianie udział bierze przeglądarka internetowa (klient) oraz serwer. Websocket inaczej znany jest również pod nazwą http 2.0. Wszystkie działania związane z tą technologią zostały wykonane korzystając z wiedzy zawartej w książce autorstwa Andrew Lombardi [12].

Zaletą tej technologii jest ciągłe połączenie pomiędzy klientem a serwerem, a co za tym idzie wymiana informacji w czasie rzeczywistym i brak obciążenia sieci. Jest to potrzebne zarówno w samej grze (gdzie obsługiwany jest ruch postaci na mapie) jak i do prawidłowego działania chatu. Pozostała część komunikacji odbywa się poprzez protokół http.

2.5. Testowanie [K.D.]

Testy jednostkowe w aplikacji są implementowane z wykorzystaniem biblioteki JUnit. Została napisana w języku Java. Z tego powodu jest z nim kompatybilna. Pozwala na pisanie testów jednostkowych oraz generowanie raportów. Dobre praktyki testowania opisuje Roy Osherove w swoim dziele pt. „The Art of Unit Testing" [13].

Do weryfikacji jakości powstałego kodu służy platforma SonarQube. Pozwala ona na statyczną analizę kodu dla wielu języków programowania. Dzięki SonarQube, można z łatwością utrzymać kod w porządku, pozbyć się powtarzalnych błędów oraz potencjalnych podatności aplikacji. Elementami analizy są:
- wykrywanie nieprzejrzystego kodu
- wykrywanie błędów
- wykrywanie podatności na atak
- pokrycie kodu testami jednostokowymi
- złożoność kodu dla poszczególnych metod
- duplikacje kodu
Oprócz tego, SonarQube pozwala na śledzenie historii analiz oraz dostarcza możliwość generowania wykresów ewolucji aplikacji.
O platformie SonarQube pisze Patroklos P. Papapetrou wraz z G. Ann Campbell w dziele pt. „SonarQube in Action" [14].

2.6. Pozostałe technologie [J.D.]

W celu przyspieszania procesu tworzenia oprogramowania wykorzystana jest technologia JHipster. Pozwalaja na generowanie elementów bazowych aplikacji. Podczas tego procesu, JHipster pyta o podanie szeregu informacji, na podstawie których generator utworzy niezbędne podstawy spełniające potrzeby użytkownika.

W efekcie otrzymany jest gotowy na edycję kod źródłowy serwera (w języku Java) oraz klienta (w jezyku JavaScript).
W skład gotowego kodu źródłowego serwera wchodzą:
- Zabezpieczenie aplikacji
- Podstawowa konfiguracja
- Skonfigurowane narzędzie automatyzujące budowanie aplikacji
- Skonfigurowane narzędzie do tworzenia logów z aplikacji
- Skonfigurowane konta użytkowników
- Skonfigurowany do działania protokół http

W skład gotowego kodu źródłowego klienta wchodzą:
- Gotowy podstawowy layout
- Skonfigurowane narzędzie automatyzujące budowanie aplikacji
- Zabezpieczenia aplikacji
- Moduł logowania do aplikacji
- Obsługa błędów z serwera
- Skonfigurowany do działania protokół http

Dodatkowo można posłużyć się narzędziem generującym wszystkie modele danych na podstawie definicji zapisanych w narzędziu „JDL-Studio". Wszystkie modele generowane są wraz z niezbędną logiką aplikacji pozwalającą na edytowanie ich z poziomu klienta.

Do ułatwienia wdrażania aplikacji, wykorzystane jest narzędzie Docker. Działa analogicznie do maszyny wirtualnej. Dzięki niej, można uruchomić wiele potrzebnych narzędzi oraz bibliotek nie martwiąc się o ich platformowość. Definicje potrzebnych komponentów do uruchomienia aplikacji wraz z ich konfiguracją zapisuje się w jednym pliku konfiguracyjnym. Takie rozwiązanie umożliwia uruchomienie wszystkiego co niezbędne do działania programu w zaledwie kilka sekund, bez zmartwień o dodatkową konfiguracje czy inne potrzebne elementy. Użycie dockera w praktyce opisuje Ian Miell oraz Aidan Hobson Sayers w dziele pt. „Docker in Practice, Second Edition" [15].

Narzędzie gradle, w znacznym stopniu ułatwia zarządzanie projektem. Do jego zadań należy budowanie aplikacji. Nadzoruje proces budowania programu oraz wszystkich potrzebnych zależności. Dba o aktualizowanie bibliotek do prawidłowych wersji oraz o pobieranie brakujących elementów. Do działania potrzebuje pliku konfiguracyjnego zawierającego wszystkie definicje wraz z instrukcjami odnoszącymi się do budowy aplikacji.

W celu utrwalania logów aplikacji, wykorzystana jest technologia log4j2. Oprócz wielu metod zapisywania danych, umożliwia kategoryzowanie ich na błędy, informacje, debugowanie oraz możliwość zapisania wszystkiego do pliku w zdefiniowanym wcześniej formacie.

Zarządzanie stanem aplikacji może być problemem, dlatego do ułatwienia wybrane jest narzędzie Redux. Pełni rolę kontenera dla aplikacji napisanych w języku JavaScript, który przechowuje jej stan w sposób przejrzysty i przewidywalny, co pozwala na proste testowanie oraz zarządzanie nim. Zmiany stanu działają na podobnej zasadzie co koncept *zwijania* w funkcyjnym paradygmacie programowania. Stan jest dzielony na podstany, które na sam koniec są łączone w całość. Gdy dokonana jest zmiana, zmieniany jest jeden z podstanów, który później łączony jest razem z pozostałymi – niezmienionymi. Szczegóły działania można znaleźć w książce „The Complete Redux Book" [16].

Do pakowania modułów napisanych w języku programowania JavaScript służy Webpack. Narzędzie to tworzy graf wszystkich zależności w aplikacji, a następnie pozwala na ich łatwe użytkowanie. W grafie znajdują się między innymi pliki, które wielokrotnie są wczytywane w grach. Między innymi: zdjęcia, czcionki, definicje stylów CSS.

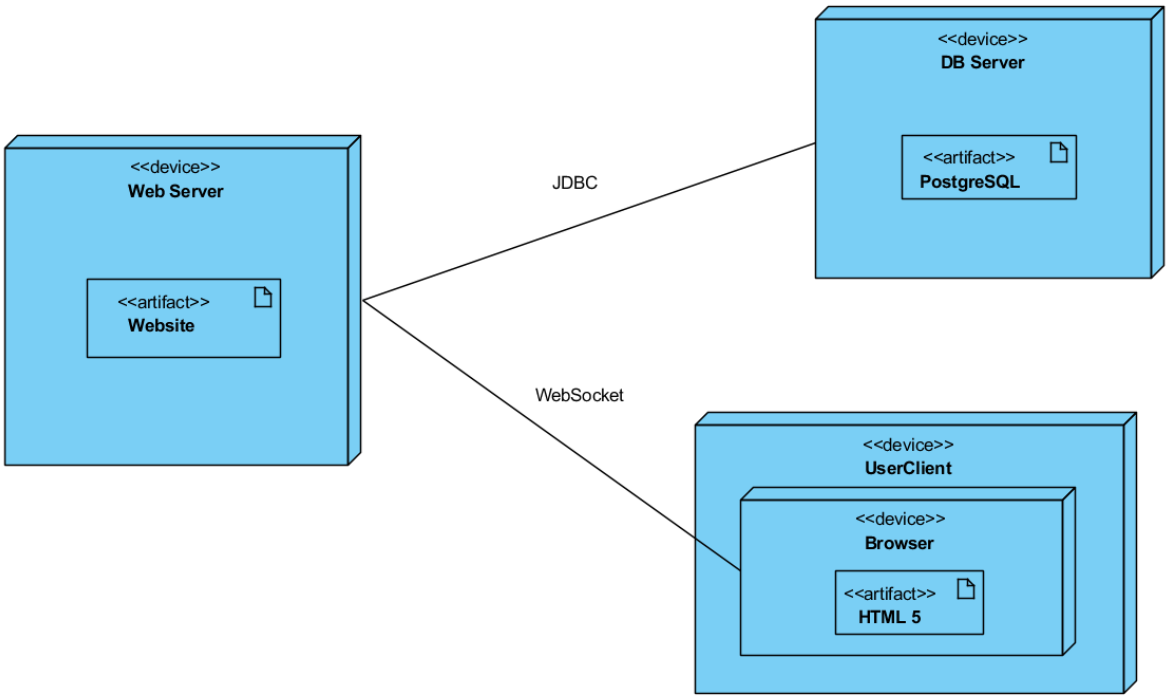
Zawartość wyświetlana na stronie po stronie klienta opisana jest przy pomocy narzędzi:

- HTML5 – Zbiór oprogramowania, który definiuje zachowania i cechy zawartości na stronach internetowych za pomocą znaczników. Jest nierozłączną częścią każdej aplikacji internetowej.
- CSS3 – Język służący do opisywania sposobu prezentowania treści na stronie internetowej oraz pozwalający na formatowanie wyglądu każdego elementu: między innymi tekstów, formularzy, przycisków, wykresów, komponentów na stronie jak i nawigacji. Jest nierozłączną częścią każdej aplikacji internetowej.
- Bootstrap – Framework do języka CSS, zawierający wiele gotowych szablonów dla elementów tworzonych na stronie internetowej. Pozwala na bardzo szybką implementację interfejsu graficznego. Korzysta z języka JavaScript, co pozwala na proste dodanie animacji.

3. Projekt [K.D., J.D.]

3.1. Architektura [K.D., J.D.]

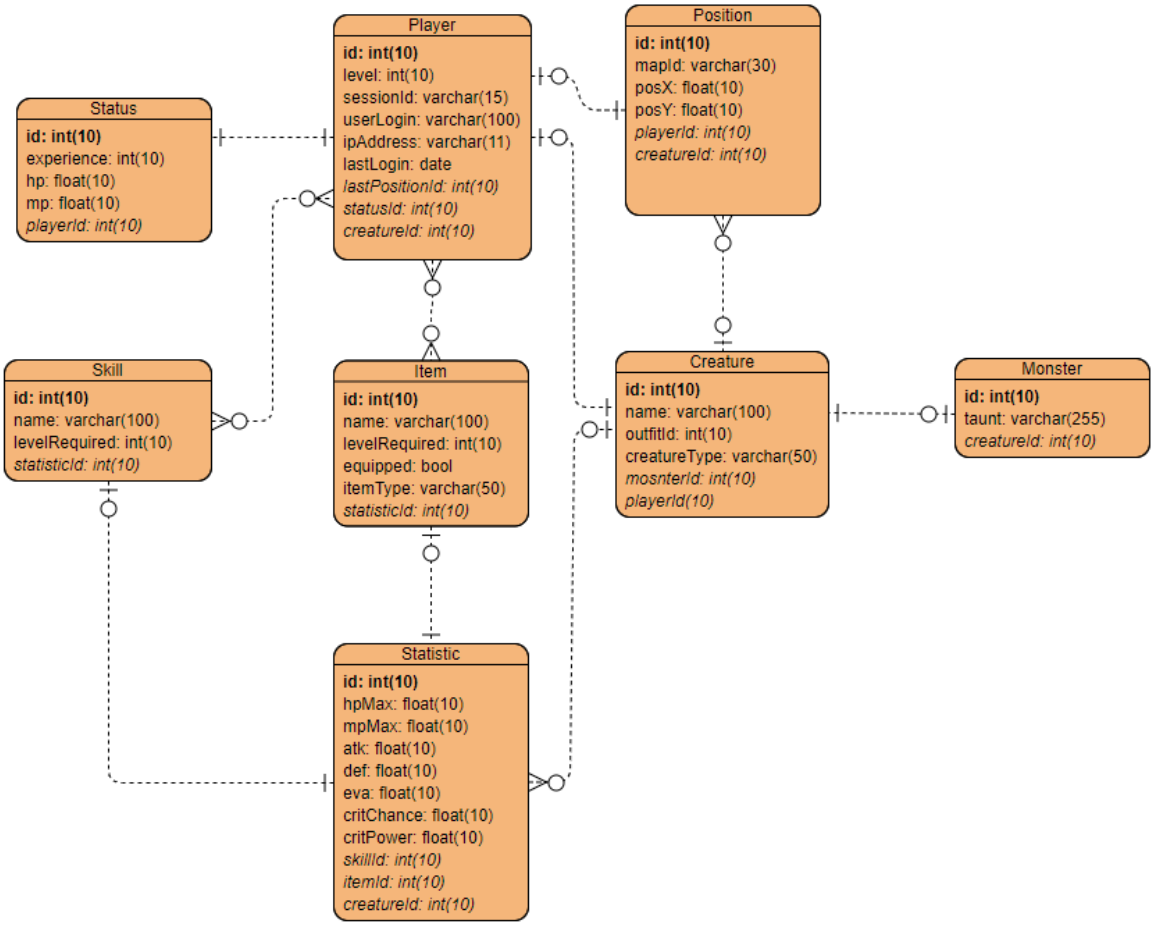
Głównymi składowymi systemu są: serwer aplikacji, serwer bazodanowy oraz urządzenie klienta. Odnoszą się do siebie w takiej relacji jak jest to opisane na diagramie wdrożenia (Rys. 3.1). Serwer aplikacji pełni główną rolę w funkcjonowaniu systemu, gdyż od niego zależy zawartość wyświetlana po stronie klienta. Trzeba również uwzględnić jakie wartości zostaną utrwalone w bazie danych. Jednym z głównych zadań aplikacji po stronie urządzenia klienta jest odbieranie informacji z serwera. W dalszej części wyświetlanie ich w sposób dogodny dla użytkownika w postaci mapy oraz przemieszczających się po nich grafik postaci. Do zadań tej części aplikacji należy wysyłanie każdej aktualizacji stanu użytkownika w celu zapisania go. Umożliwia to odczytanie pozycji gracza przez innych użytkowników.



Rys. 3.1 Diagram wdrożenia

3.2. Struktura danych [K.D., J.D.]

Do konceptualnego opisu relacyjnej bazy danych wykorzystuje się diagram związków encji. Powszechnie znany jest pod nazwą ERD. Z tego powodu wykorzystany jest do opisu struktury danych w omawianej aplikacji (Rys. 3.2).



Rys. 3.2 Diagram ERD

Encje na diagramie można pogrupować ze względu na ich znaczenie. *Creature*, *Player* oraz *Monster*, reprezentują jednostki w grze. Ich znaczenia to odpowiednio kreatura, gracz oraz potwór jako przeciwnik gracza. Encja *Creature*, tak jak jej atrybuty (Tab. 3.1), jest częścią wspólną dwóch pozostałych.

Tab. 3.1 Opis atrybutów encji *Creature*.

Atrybut	Opis
id	Identyfikator kreatury.
name	Nazwa kreatury.
outfitId	Identyfikator wyglądu. Służy do przypisania odpowiedniej grafiki z poziomu gry do danej kreatury.
creatureType	Przypisany typ kreatury. Każdy typ przypisany do kreatury, określa sposób w jaki jednostka walczy z graczem. (Tab. 3.2).
monsterId	Opcjonalny klucz obcy do encji *Monster*, wykorzystywany w przypadku, gdy kreatura reprezentuje potwora.

playerId	Opcjonalny klucz obcy do encji *Player*, wykorzystywany w przypadku, gdy kreatura reprezentuje gracza.

Tab. 3.2 Wyróżnione typy kreatur.

Typ kreatury	Opis
Krótkodystansowa	Walczy na krótki dystans
Długodystansowa	Walczy na odległość
Magiczna	Posługuje się magią

Encja *Creature* jest powiązana w relacji jeden do wielu z encją *Position*, która określa miejsce, w którym może pojawić się dana kreatura. W przypadku gracza, będzie to jedno miejsce jak na przykład lokalizacja startowa. Przeciwnie działa to dla potworów, gdyż zakłada się możliwość pojawienia ich w wielu różnych miejscach. Atrybuty encji oznaczają identyfikator mapy na których może pojawić się kreatura oraz pozycję w kartezjańskim układzie współrzędnych (Tab. 3.3).

Tab. 3.3 Opis atrybutów encji *Position*

Atrybuty	Opis
id	Identyfikator Pozycji.
mapId	Identyfikator mapy, której dotyczy pozycja.
posX	Pierwszy wymiar pozycji.
posY	Drugi wymiar pozycji.
playerId	Opcjonalny klucz obcy encji *Player*, wykorzystywany w przypadku, gdy pozycja jest ostatnim zapisanym położeniem gracza.
creatureId	Opcjonalny klucz obcy encji *Creature*, wykorzystywany w przypadku, gdy pozycja jest miejscem odrodzenia kreatury.

Z encją *Creature* powiązana jest również w relacji jeden do wielu encja *Statistic*, która opisuje atrybuty wykorzystywane do walki (Tab. 3.4). Ostateczna wartość obrażeń jest wyliczana wzorem (patrz (1)), gdzie:

x – ostateczna wartość obrażeń zadanych przeciwnikowi

t – wartość ataku z atrybutu *atk*

c – mnożnik wzmocnionego ataku, wyznaczany za pomocą losowania czy doszło do wzmocnienia wykorzystując wartość procentową z atrybutu *critChance*. W przypadku dojścia do skutku, mnożnik przyjmuje wartość z atrybutu *critPower*, w przeciwnym razie przyjmuje wartość 1.

Tab. 3.4 Opis atrybutów encji *Statistic*.

Atrybut	Opis
id	Identyfikator Statystyk.
hpMax	Maksymalna liczba punktów życia.
mpMax	Maksymalna liczba punktów many.
atk	Wartość obrażeń zadawanych przeciwnikom.
def	Wartość obrażeń ignorowanych w trakcie walki.

eva	Procent szans na uniknięcie ataku.
critChance	Procent szans na wzmocniony atak.
critPower	Mnożnik wzmocnionego ataku.
skillId	Opcjonalny klucz obcy encji *Skill*, wykorzystywany w przypadku, gdy statystyki odnoszą się do umiejętności.
itemId	Opcjonalny klucz obcy encji *Item*, wykorzystywany w przypadku, gdy statystyki odnoszą się do przedmiotu.
creatureId	Opcjonalny klucz obcy encji *Creature*, wykorzystywany w przypadku, gdy statystyki odnoszą się do kreatury.

$$x = t * c \tag{1}$$

Encja *Monster* jest uszczegółowieniem *Creature*. Dodatkowym atrybutem jest pole *taunt*, w którym zapisywany jest ciąg znaków charakterystyczny dla danego potwora. Reprezentuje ono zdanie wypowiadane przez niego w trakcie gry (Tab. 3.5).

Tab. 3.5 Opis atrybutów encji *Monster*.

Atrybut	Opis
id	Identyfikator potwora.
Taunt	Tekst wypowiadany przez potwora w trakcie gry.
creatureId	Obligatoryjny klucz obcy encji *Creature.*

Drugim uszczegółowieniem encji *Creature* jest *Player.* Ze względu na reprezentowanie użytkownika aplikacji, zawiera ona dane związane z połączeniem z serwerem (Tab. 3.6).

Tab. 3.6 Opis atrybutów encji *Player*.

Atrybut	Opis
id	Identyfikator gracza.
Level	Aktualy poziom gracza.
sessionId	Identyfikator sesji.
userLogin	Login użytkownika.
ipAddress	Adres IP użytkownika.
lastLogin	Data ostatniego zalogowania użytkownika
lastPositionId	Obligatoryjny klucz obcy encji *Position,* która reprezentuje ostatnią zapisaną pozycję gracza.
statusId	Obligatoryjny klucz obcy encji *Status.*
creatureId	Obligatoryjny klucz obcy encji *Creature.*

Encja *Player*, ma powiązanie jeden do jednego z *Status*, gdzie przechowywany jest aktualny stan gracza (Tab. 3.7).

Tab. 3.7 Opis atrybutów encji *Status*.

Atrybut	Opis
id	Identyfikator statusu.
Experience	Aktualna liczba punktów doświadczenia.
Hp	Aktualna liczba punktów życia.
Mp	Aktualna liczba punktów many.
playerId	Obligatoryjny klucz obcy encji *Player*.

Encja *Player* ma również powiązania z *Item* (Rys. 3.8) oraz *Skill* (Rys. 3.10), które oznaczają odpowiednio przedmioty oraz umiejętności. Ich rolą jest modyfikacja statystyk postaci, gdzie różnica jest opisana w encji *Statistic*, z którą są powiązane relacją jeden do jednego. Ostateczne statystyki gracza są wyliczane jako suma statystyk przypisanych do kreatury, ekwipowanych przedmiotów oraz zdolności.

Tab. 3.8 Opis atrybutów encji *Item*.

Atrybut	Opis
id	Identyfikator przedmiotu.
Name	Nazwa przedmiotu.
levelRequired	Poziom wymagany do ekwipowania przedmiotu.
Equipped	Informacja czy przedmiot jest ekwipowany.
itemType	Typ przedmiotu (Tab. 3.9).
statisticId	Obligatoryjny klucz obcy encji *Statistic*.

Tab. 3.9 Wyróżnione typy przedmiotów.

Typ przedmiotu	Opis
Oręż	Przedmiot, który gracz może ekwipować na dłonie, zazwyczaj broń lub tarcza.
Pancerz	Przedmiot, który można ekwipować na ciało gracza.
Nakrycie głowy	Przedmiot, który można ekwipować na głowę gracza.
Nakrycie nóg	Przedmiot, który można ekwipować na nogi gracza.
Buty	Przedmiot, który można ekwipować na stopy.
Pozostałe	Pozostałe przedmioty, których nie można ekwipować.

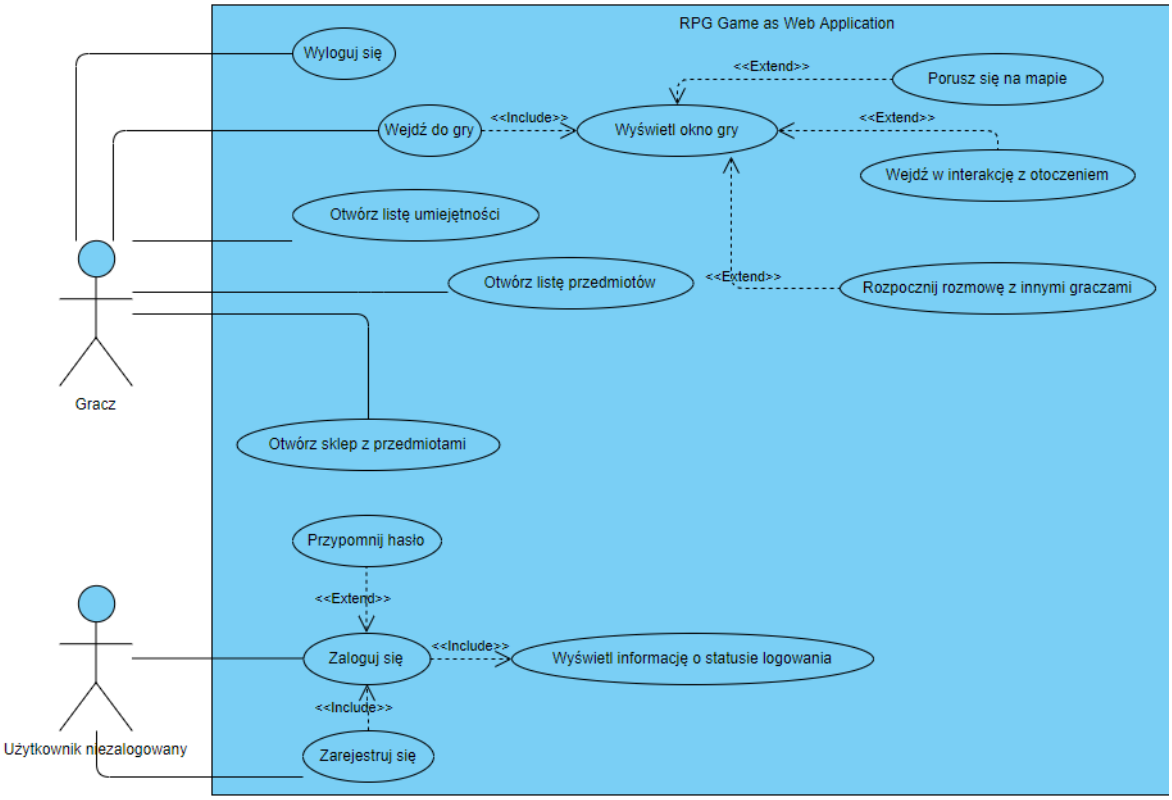
Tab. 3.10 Opis atrybutów encji *Skill*.

Atrybut	Opis
id	Identyfikator
name	Nazwa umiejętności
levelRequired	Minimalny poziom wymagany do posiadania umiejętności
statisticId	Obligatoryjny klucz obcy encji *Statistic*.

Przedmioty można ekwipować tylko wtedy, kiedy ich typ jest inny niż *Pozostałe* (Tab. 3.9). Oprócz tego, nie można ekwipować dwóch przedmiotów tego samego typu.

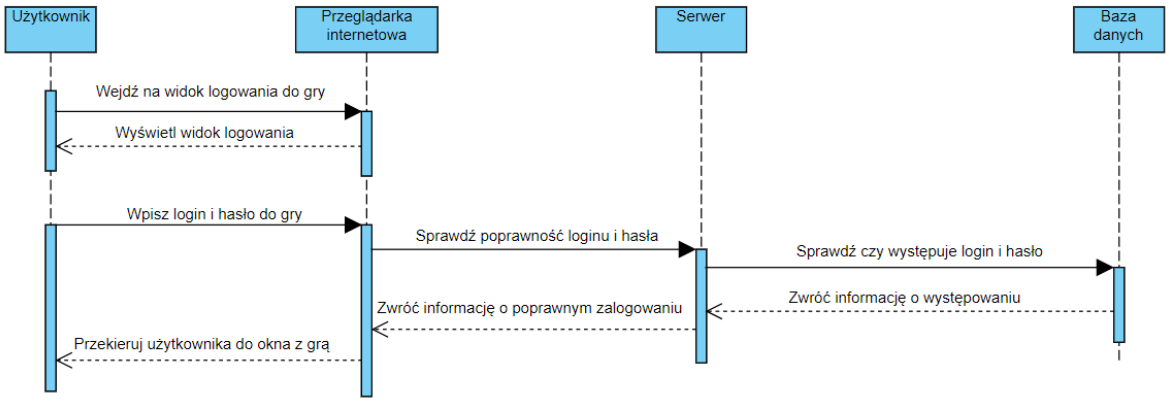
3.3. Przypadki użycia [K.D., J.D.]

W aplikacji rozważane są dwa typy użytkowników. Wyróżniamy niezalogowanego oraz zalogowanego, określanego inaczej mianem gracza. Każda osoba po wejściu do aplikacji, przyjmuje rolę użytkownika niezalogowanego. Jedyne akcje jakie może podjąć to rejestracja, logowanie i przypomnienie hasła. Po prawidłowym przejściu procesu logowania nadawane są uprawnienia gracza. Możliwe staje się otwarcie listy przedmiotów, listy umiejętności czy wejścia do gry. W samej grze można poruszać się po mapie, wejść w interakcje z otoczeniem oraz rozpocząć rozmowę z innymi zalogowanymi użytkownikami. Wszystkie możliwości są opisane diagramem przypadków użycia (Rys. 3.3).



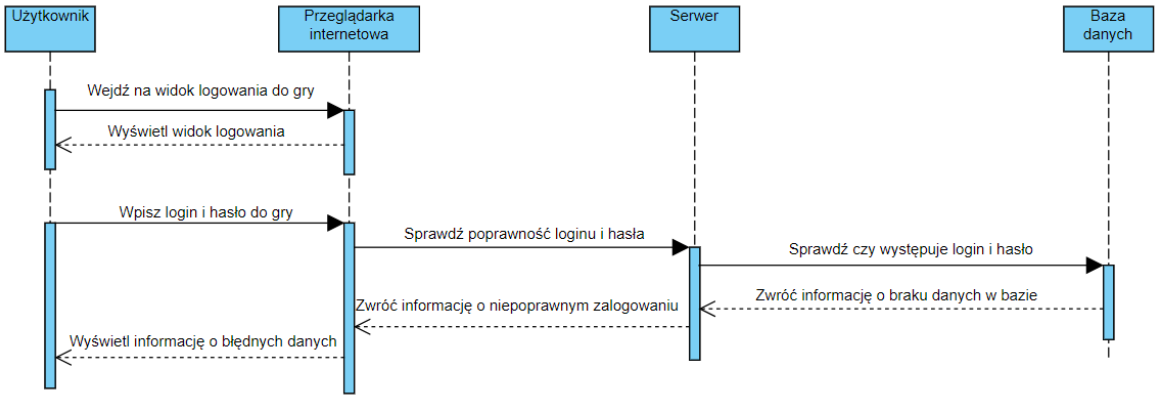
Rys. 3.3 Diagram przypadków użycia

W celu zalogowania do aplikacji, użytkownik otwiera stronę przeznaczoną do logowania gry. Przeglądarka internetowa wyświetla mu odpowiednią formę z miejscem na login i hasło. Gdy użytkownik zatwierdzi swoje dane, przeglądarka wysyła je do serwera w celu weryfikacji. Następnie serwer sprawdza czy wprowadzony login i hasło występują w bazie danych. Gdy okaże się, że takie dane istnieją to serwer zwraca informacje o poprawnym zalogowaniu. W takim wypadku przeglądarka internetowa przekierowuje użytkownika bezpośrednio do okna z grą (Rys. 3.4).



Rys. 3.4 Diagram sekwencji – logowanie zakończone sukcesem

Jeśli dane nie istnieją w bazie danych to serwer zwróci informacje o niepoprawnym zalogowaniu. W takim wypadku przeglądarka internetowa wyświetla użytkownikowi informację o błędnie wprowadzonych danych (Rys. 3.5).



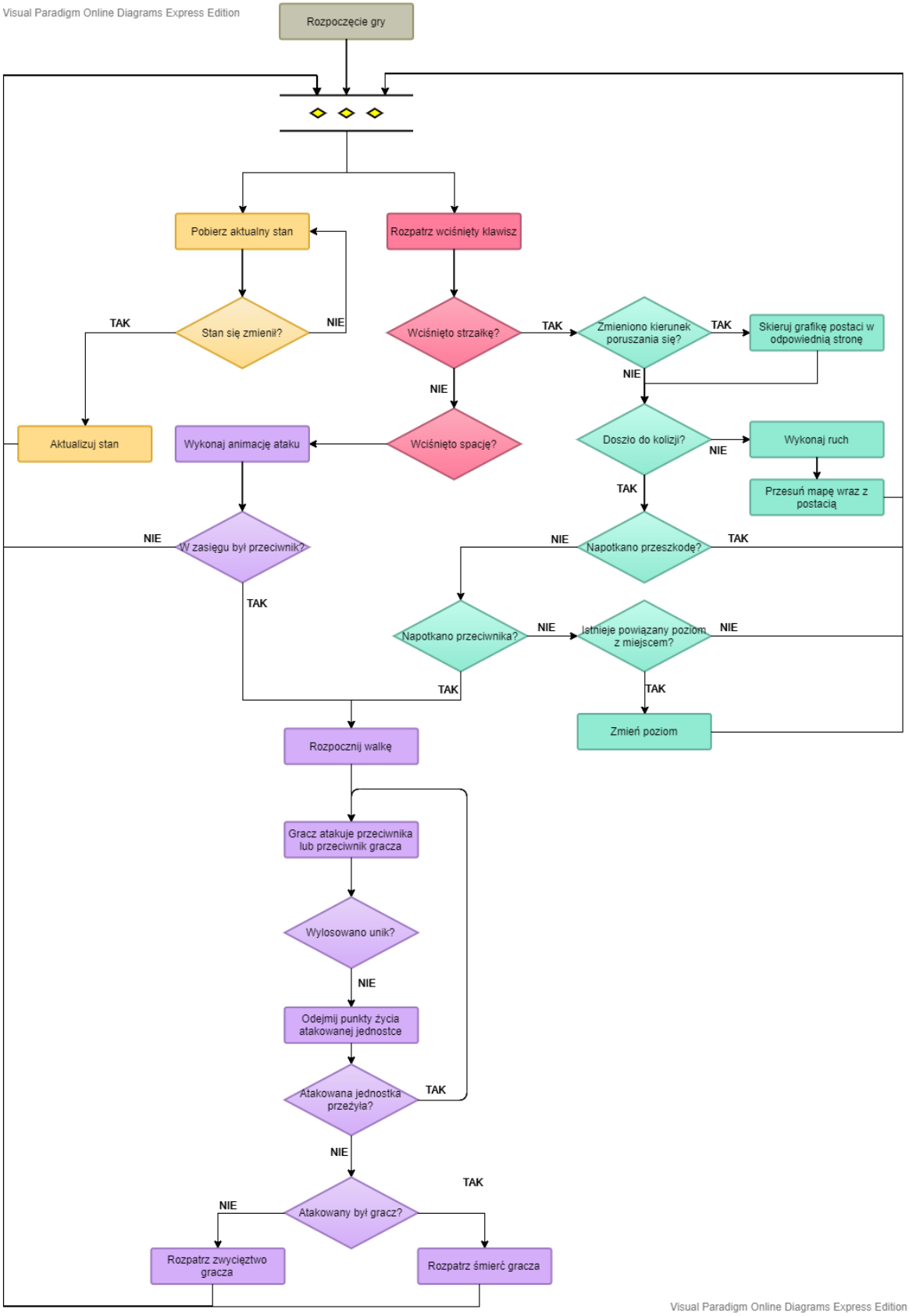
Rys. 3.5 Diagram sekwencji – logowanie zakończone niepowodzeniem

Do opisu przepływu stanu gry, użyto schematu blokowego (Rys. 3.6.). Równolegle są wykonywane dwa fragmenty. Pierwszy odpowiadający za odświeżanie stanu oznaczony żółtym kolorem. Drugi, określający interakcje gry z użytkownikiem opisany czerwonym kolorem.

W przypadku wciśnięcia przycisku odpowiadającemu strzałce, następuje rozpatrzenie ruchu gracza. Część ta jest opisana kolorem zielonym. Jeśli strzałka określa kierunek inny niż ten w jaki postać była skierowana, zostaje on aktualizowany. Gdy napotkano przeszkodę, przez którą nie da się przejść np. mur czy głaz, ruch nie zostaje wykonany i jego rozpatrywanie jest zakończone. W przeciwnym razie szacowane jest, czy napotkano w ten sposób przeciwnika. Jeżeli tak, rozpoczyna się walka. W każdej innej sytuacji sprawdzane jest czy miejsce, na które przemieściła się postać to krawędź, która przenosi postać na inną scenę.

Kiedy zostanie wciśnięta spacja, postać wykonuje animację ataku. Część ta jest opisana kolorem fioletowym. Jeżeli w zasięgu znajduje się przeciwnik, rozpoczynana jest walka. Oznacza to, że uczestnicy atakują się wzajemnie do momentu rozpatrzenia wyniku. Najpierw sprawdzane jest czy akcja gracza wymagała punktów many. Jeżeli tak zostają mu one odjęte. Obrońca ma szansę na uniknięcie ataku. W przypadku nieudanego uniku, odejmowane są punkty życia atakowanemu. Gdy wartość punktów życia gracza osiągnęła

wartość niedodatnią, rozpatrywana jest jego śmierć. Natomiast w sytuacji, gdy przeciwnikowi zabraknie punktów życia, rozpatruje się zwycięstwo gracza.



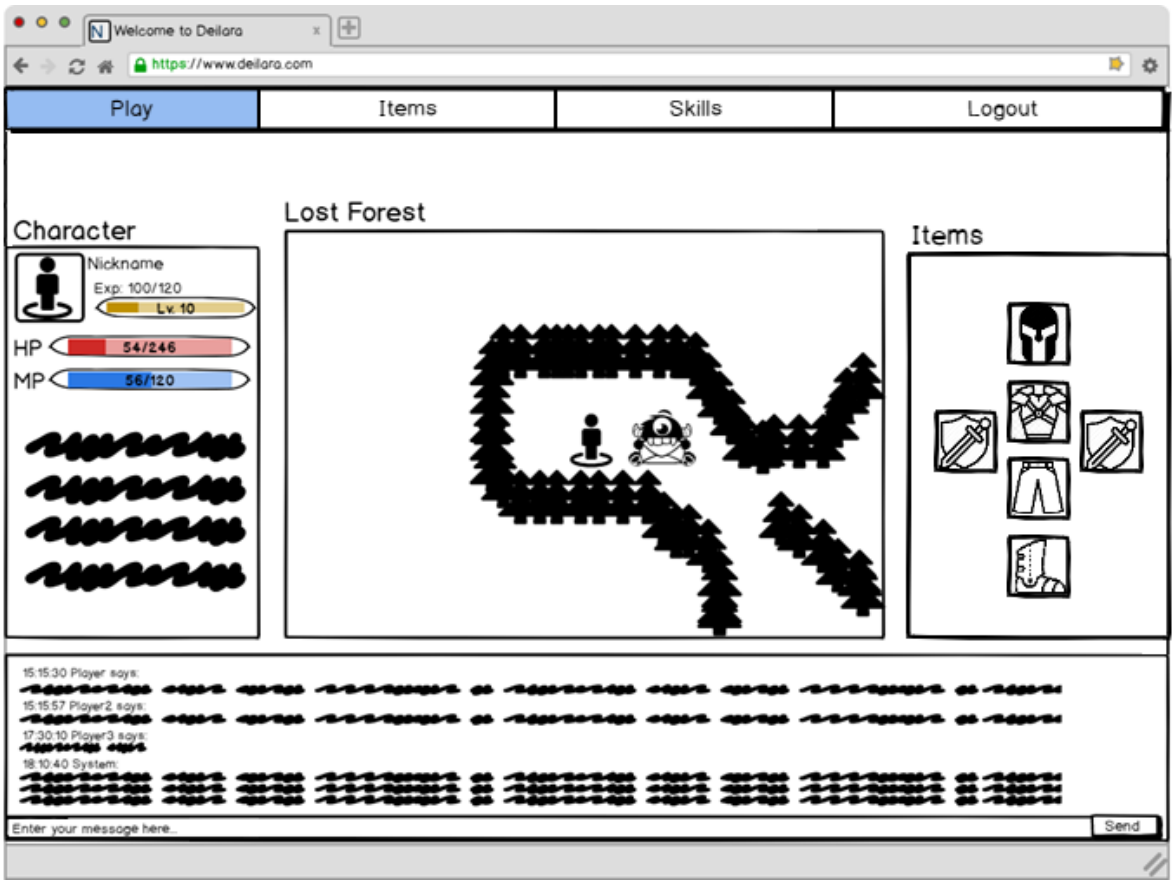
Rys. 3.6 Schemat blokowy działania gry

3.4. Interfejs [K.D., J.D.]

Projekt interfejsu wykonany jest przy pomocy narzędzia balsamiq dostępnego na witrynie https://balsamiq.cloud. Pozwala użytkownikowi na proste dodawanie oraz edycje elementów do makiety. Dostarcza również dużą ilość gotowych rozwiązań.

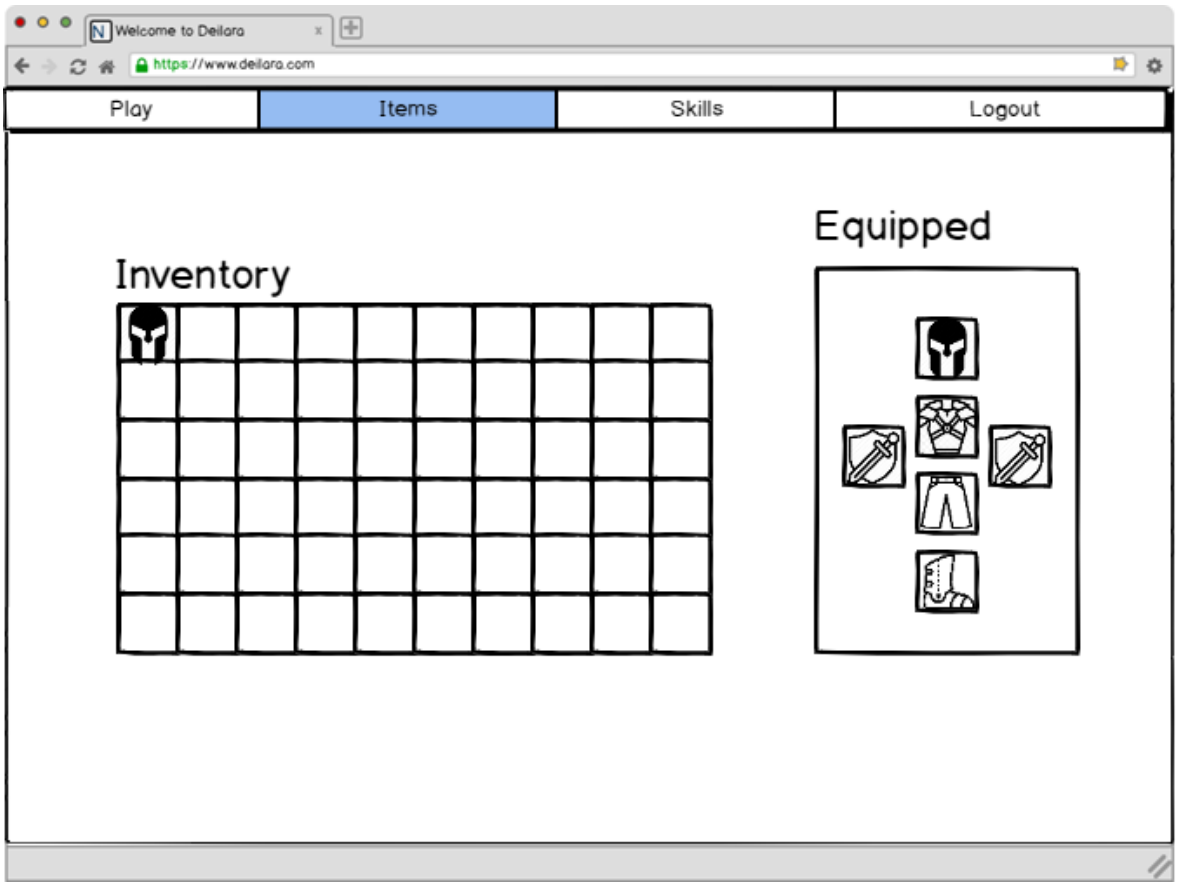
Ze względu na konieczność rozróżnienia użytkowników, wyróżniamy takie widoki jak logowanie, rejestracja czy zapomnienie hasła. Najważniejsze widoki są dostępne dopiero po zalogowaniu. Wyróżniamy widok główny gry, postaci oraz umiejętności

Na głównym widoku aplikacji, znajdują się najważniejsze elementy (Rys. 3.7.). Jest to między innymi mapa, na której odbywają się wszystkie interakcje zarówno z graczami, jak i przeciwnikami. Po lewej stronie znajdują się najważniejsze informacje o postaci, aktualny stan oraz jej statystyki. Z prawej natomiast widnieją ekwipowane przedmioty gracza, których szczegóły można zobaczyć po najechaniu kursorem myszy. Pod mapą świata gry widnieje okno z wiadomościami, które pozwala na dostęp do szybkiej komunikacji z innymi użytkownikami.



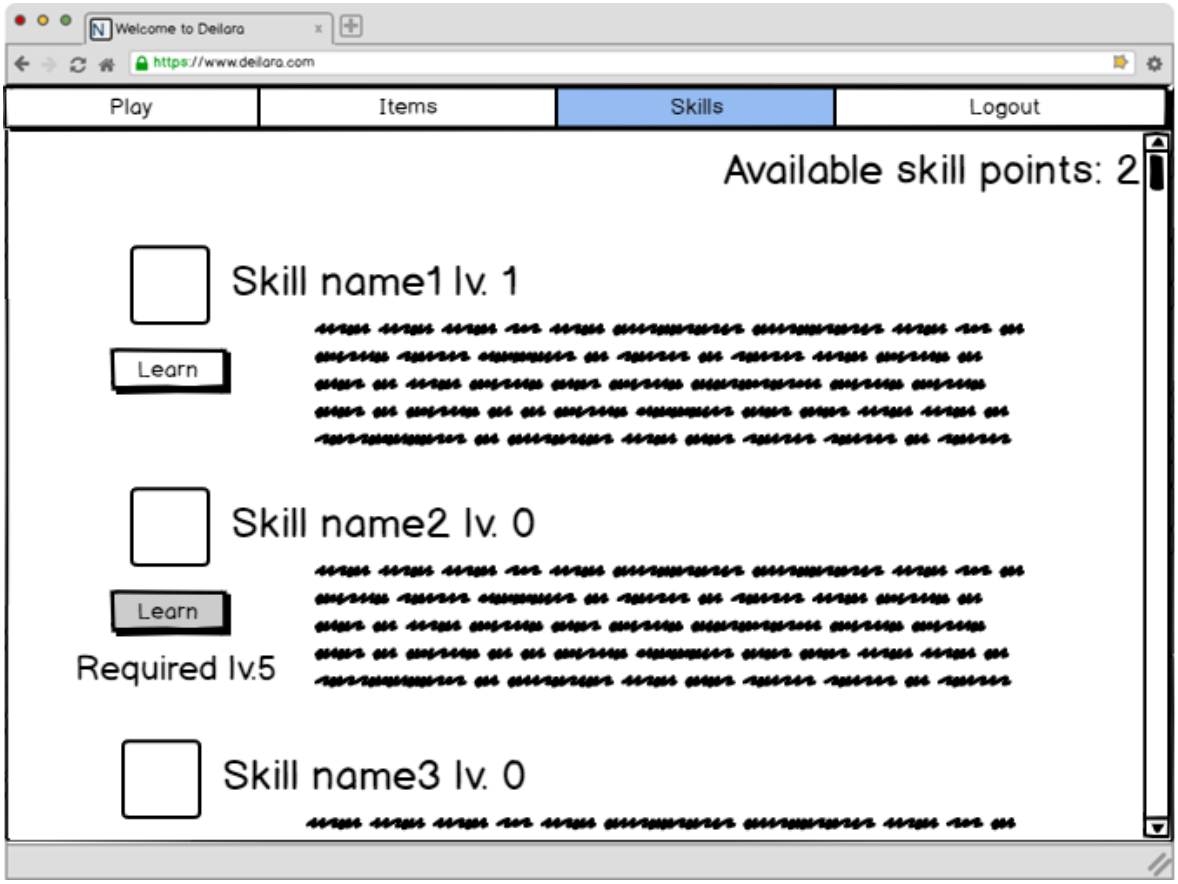
Rys. 3.7 Makieta widoku głównego gry

Zakładka „Items” służy do zarządzania przedmiotami w trakcie gry (Rys. 3.8.). W dowolnym momencie istnieje możliwość zmiany ekwipowanych przedmiotów. By tego dokonać wystarczy kliknąć w odpowiedni przedmiot znajdujący się w sekcji „Inventory”. Podgląd przedmiotów, które są w danej chwili ekwipowane, wyświetlane są po prawej stronie.



Rys. 3.8 Makieta widoku ekwipunku

Zakładka „Skills” służy do zarządzania umiejętnościami w grze (Rys. 3.9). W dowolnym momencie istnieje możliwość ich nauki. By tego dokonać wystarczy kliknąć w przycisk „Learn” przypisany do konkretnej umiejętności. Każda dostępna umiejętność składa się z nazwy oraz opisu. Opis wyświetlany jest bezpośrednio pod nazwą.



Rys. 3.9 Makieta widoku umiejętności

Gdy pierwszy raz użytkownik wejdzie do gry i nie posiada konta, nie będzie mógł skorzystać ze wszystkich możliwości aplikacji. W takiej sytuacji wyświetlona jest strona domowa oraz możliwość założenia konta (Rys. 3.10) lub logowania (Rys. 3.12).

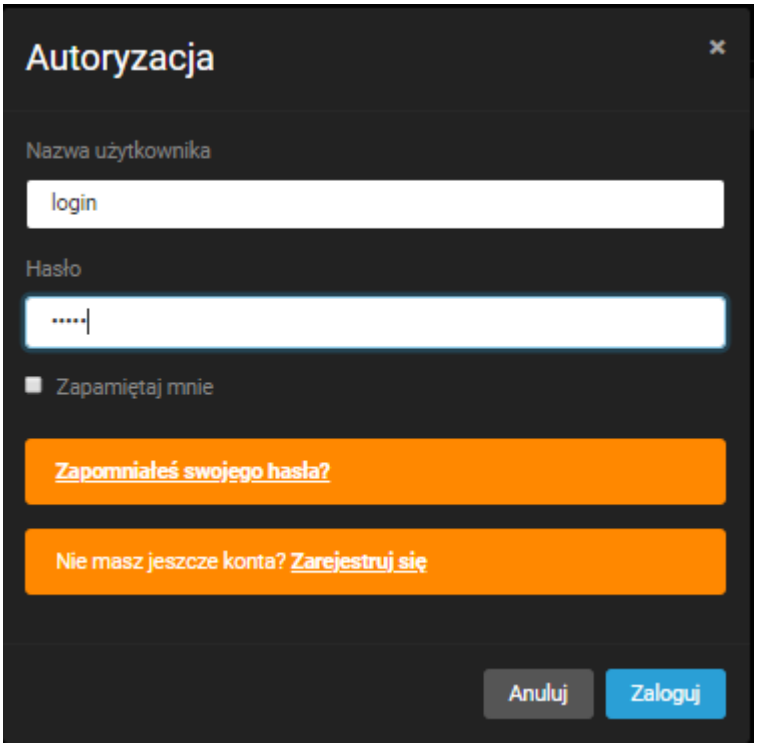
Podczas rejestracji, trzeba podać login, email oraz hasło. W celach informacyjnych wyświetlony jest również pasek, który informuje o stopniu skomplikowania hasła.

Rys. 3.10 Makieta widoku rejestracji

Gdy użytkownik zapomni hasła, ma możliwość zresetowania go poprzez przejście na stronę „Zresetuj hasło” (Rys. 3.11). By tego dokonać wystarczy podać adres email użyty podczas rejestracji. Na ten adres zostanie dostarczona odpowiednia wiadomość pozwalająca na zmianę bieżącego hasła.

Rys. 3.11 Makieta widoku resetowania hasła

Kiedy gracz posiada już założone konto, może się zalogować uzupełniając swój login oraz hasło. Istnieje również możliwość zapamiętania hasła, by uniknąć logowania przy każdym wejściu do aplikacji. Poniżej wyświetlone są pomocnicze przekierowania na widok resetowania hasła oraz rejestracji.



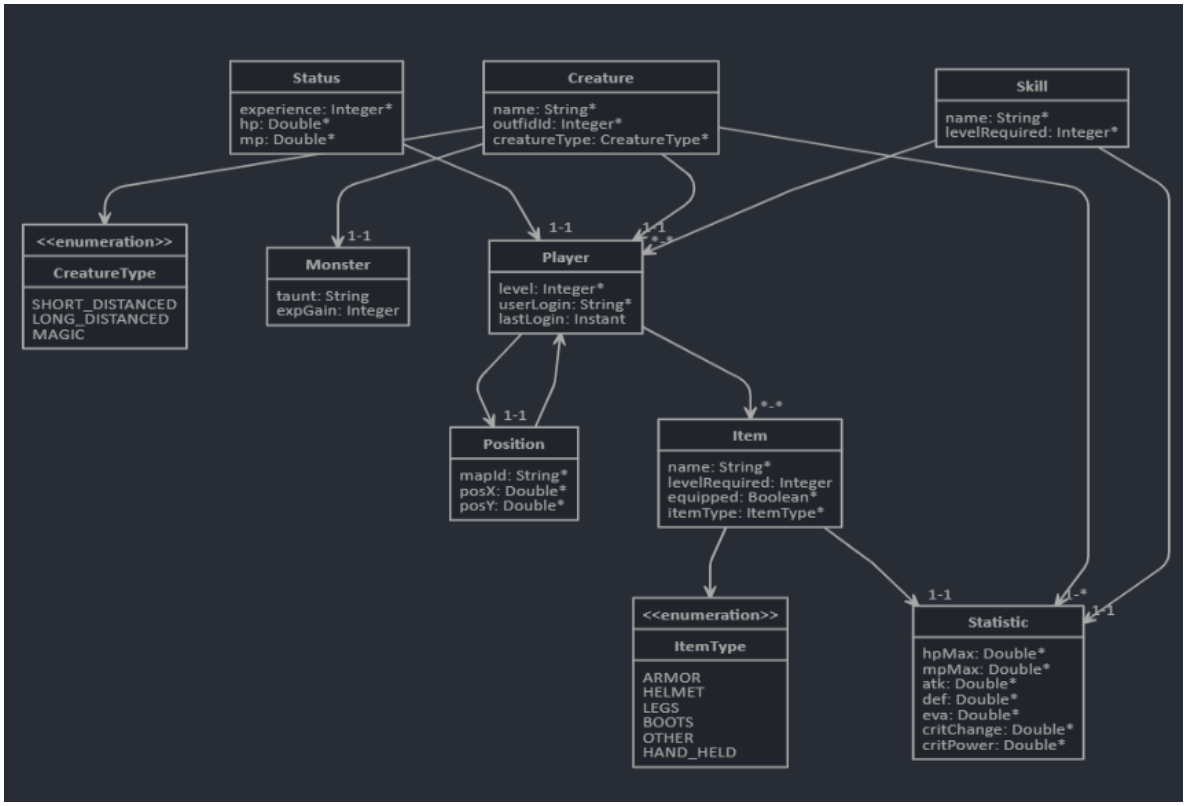
Rys. 3.12 Makieta widoku logowania

4. Implementacja [K.D., J.D.]

4.1. Struktura danych [K.D., J.D.]

W celu uproszczenia implementacji struktury danych, wykorzystane jest narzędzie JDL-Studio, które jest częścią składową technologii JHipster. Służy ono do generowania plików w językach programowania Java oraz JavaScript, które odpowiadają za zarządzanie oraz wyświetlanie danych.

Działania związane z JDL-Studio najlepiej wykonać w oparciu o tekst napisany przez twórcę tej technologii. Matt Raible opisuje w swoim dziele między innymi składnię kodu [17], w którym zdefiniowana jest struktura danych. Tak utworzona struktura następnie zapisywana jest w pliku z rozszerzeniem *.jdl*. JHipster dostarcza narzędzie, za pomocą którego można go zaimportować. Takie działanie generuje kod ostatecznej aplikacji. W celu uproszczenia procesu, można wykorzystać aplikację internetową, która po każdym dopisanym fragmencie, narysuje diagram odpowiadający opisanej strukturze (Rys. 4.1). Autor opisuje również sposób w jaki należy korzystać z tego narzędzia.



Rys. 4.1 Diagram encji w JDL-Studio.

Domeny struktury definiowane są po słowie *entity* (Rys. 4.2). W dalszej części należy uzupełnić nazwę domeny oraz w nawiasach klamrowych pola do niej należące. Aby poprawnie zdefiniować pole, należy zapisać nazwę atrybutu, a za nią typ danych. Opcjonalnie można oznaczyć je jako wymagane, zapisując słowo kluczowe *required* na końcu wyrażenia. Definicje atrybutów są oddzielone przecinkami.

```
entity Creature {
name String required,
outfidId Integer required,
creatureType CreatureType required
}
```

Rys. 4.2 Przykładowa definicja encji w JDL-Studio.

W przeciwieństwie do domen, relacje są definiowane za pomocą słowa *relationship* (Rys. 4.3). Za nim należy wpisać jej nazwę. Dopuszczalne są *OneToOne*, *OneToMany*, *ManyToOne* oraz *ManyToMany*.

Nawiązując do słów Lynn'a Beighleya w „Head First SQL: Your Brain on SQL" [18], relacje oznaczają zależność pomiędzy dwiema domenami. Opisują, ile instancji pierwszej odnosi się do drugiej oraz odwrotnie. Wyróżnia się:

- *OneToOne* – każdemu obiektowi pierwszej domeny można przypisać tylko jeden obiekt drugiej i odwrotnie.
- *OneToMany* – każdemu obiektowi pierwszej domeny można przypisać wiele obiektów drugiej, natomiast każdemu obiektowi drugiej domeny możemy przypisać tylko jeden obiekt pierwszej.
- *ManyToOne* – każdemu obiektowi pierwszej domeny można przypisać tylko jeden obiekt drugiej, natomiast każdemu obiektowi drugiej domeny możemy przypisać wiele obiektów pierwszej.
- *ManyToMany* – zakłada wiele przypisanych obiektów powiązanej domeny po obu stronach relacji.

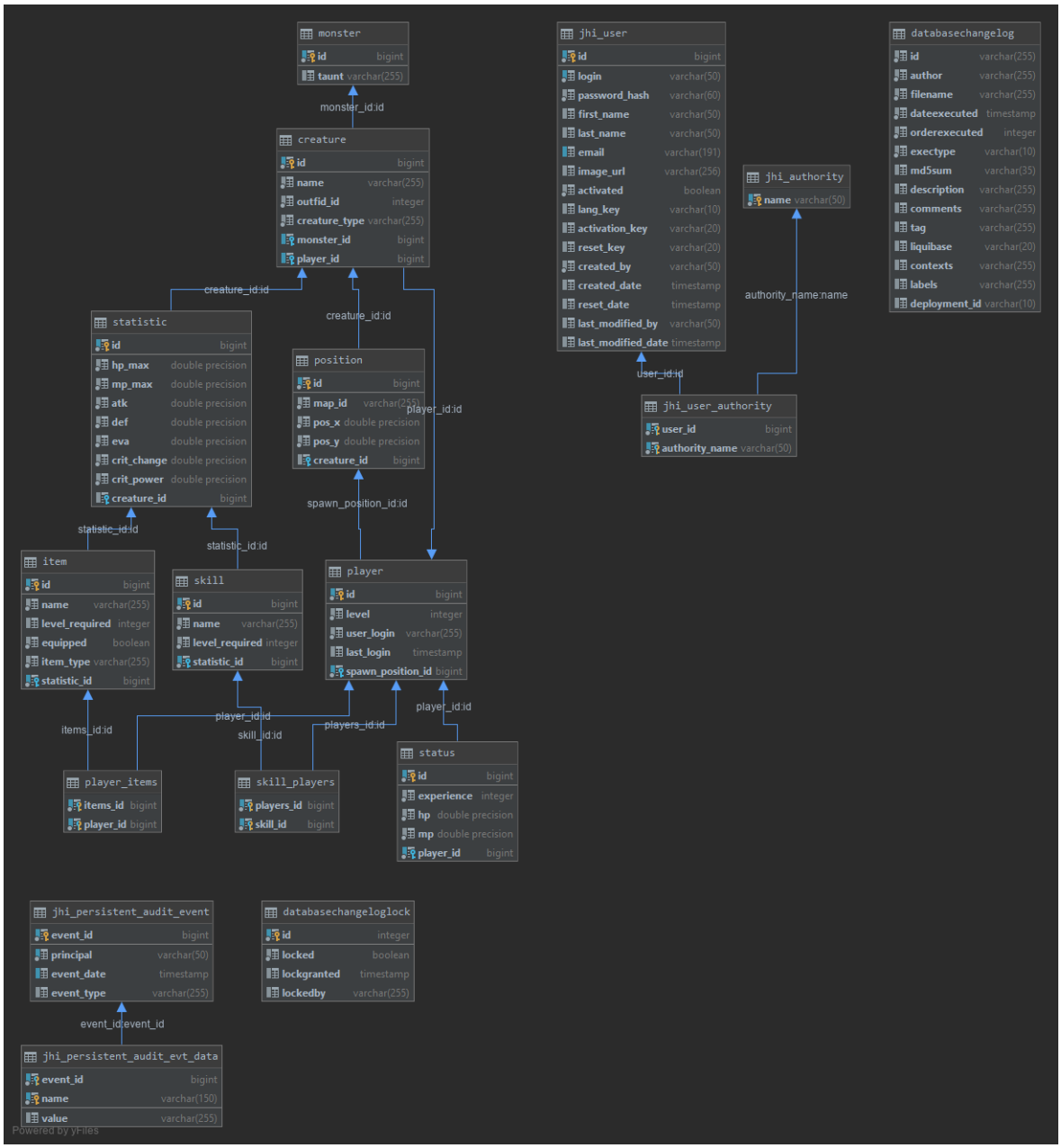
```
relationship OneToOne{
Status {player required} to Player {status required},
Creature {monster} to Monster {creature required},
Creature {player} to Player {creature required},
Player {spawnPosition required} to Position {player},
Item {statistic required} to Statistic {item},
Skill {statistic required} to Statistic {skill}
}

relationship OneToMany {
Creature {positions} to Position {creature},
Creature {statistics} to Statistic {creature}
}

relationship ManyToMany {
Skill {players} to Player {skills},
Player {items} to Item {players}
}
```

Rys. 4.3 Przykładowe definicje relacji w JDL-Studio.

W narzędziu JDL-Studio relacje definiuje się tak samo niezależnie od ich typu (Rys. 4.3). Należy wypisać nazwy dwóch domen i oddzielić je słowem kluczowym *„to”*. Przy ich nazwach można zdefiniować atrybut, będący referencją do drugiej domeny. W przypadku gdy to odniesienie jest wymagane, należy dopisać do niego słowo *required*.

Po wykonaniu skryptu utworzonego w JDL-Studio, wygenerowane są pliki z kodem źródłowym. Za utworzenie struktury w bazie danych, odpowiada narzędzie liquibase. Do jego zadań należy również kontrolowanie wszystkich zmian. Każdy plik konfiguracyjny tego narzędzia reprezentuje inną grupę operacji. Konwencja wymaga by nazywać je w sposób, który pozwala określić czego dotyczą oraz kiedy zostały utworzone, np. *20191114121036_added_entity_Player.xml*. Cyfry oznaczają datę utworzenia pliku, następne części opisują wykonywane operacje oraz domenę, której dotyczą. Zadania wykonywane przy pierwszym uruchomieniu, są skonfigurowane w pliku inicjującym. Odpowiada on za utworzenie bazy danych oraz podstawowych tabel. W tym przypadku w nazwie pliku zamiast daty umieszcza się same zera, np. *00000000000000_initial_schema.xml*.

Akcje liquibase są wykonywane w trakcie uruchamiania aplikacji. Odczytywane są wtedy pliki konfiguracyjne, za pomocą których tworzone są tabele oraz powiązania między nimi. Aby mieć możliwość wykonywania operacji na bazie danych należy mieć uruchomiony serwer narzędzia PostgreSQL. Po uruchomieniu aplikacji, oprócz tabel zdefiniowanych w JDL-Studio, utworzonych jest siedem dodatkowych. Wygnerowaną strukturę można zwizualizować za pomocą narzędzi dostarczanych w środowisku IntellijIDEA (Rys. 4.4.)



Rys. 4.4 Przykładowe definicje relacji w JDL-Studio.

Dwie dodatkowe tabele są związane z audytami. Wszystkie zdarzenia w aplikacji są zapisywane w bazie danych. Tabela *jhi_persistent_audit_event* (Tab. 4.1) przechowuje dane, kiedy zdarzenie się pojawiło oraz co jest jego źródłem. Natomiast *jhi_persistent_audit_evt_data* (Tab. 4.2) posiada informacje jakie to zdarzenie oraz jego opis. Dane te znajdują się w różnych tabelach, aby uniknąć powielania danych.

Tab. 4.1 Opis atrybutów tabeli *jhi_persistent_audit_event.*

Atrybut	Opis
event_id	Klucz główny tabeli.
Principal	Źródło zdarzenia.
Event_date	Data wystąpienia zdarzenia
event_type	Rodzaj zdarzenia.

Tab. 4.2 Opis atrybutów tabeli *jhi_persistent_audit_evt_data.*

Atrybut	Opis
event_id	Klucz obcy pochodzący z tabeli *jhi_persistent_audit_event*. Jest częścią klucza złożonego.
Name	Nazwa wydarzenia będąca częścią klucza złożonego.
Value	Wartość opisująca zdarzenie.

Dwie tabele zostały utworzone bez powiązań. Są to *databasechangelog* (Tab. 4.3) oraz *databasechangeloglock* (Tab. 4.4). Przechowują dane na temat operacji, które zaszły w bazie. Dzięki temu aplikacja jest w stanie zapobiec niekontrolowanym zmianom w strukturze. Funkcjonalność ta jest częścią narzędzia Liquibase. Tabela *databasechangeloglock* służy to umożliwenia edycji bazy danych tylko przez jedną instancję jednocześnie. Druga natomiast przechowuje dane na temat wprowadzonych zmian.

Tab. 4.3 Opis atrybutów tabeli *databasechangeloglock.*

Atrybut	Opis
id	Identyfikator tabeli.
Locked	Informacja czy baza danych jest w tym momencie zablokowana przez inną instancję Liquibase.
Lockgranted	Data umieszczenia blokady.
Lockedby	Założyciel blokady.

Tab. 4.4 Opis atrybutów tabeli *databasechangelog.*

Atrybut	Opis
id	Identyfikator zmiany.
Author	Autor zmian.
Filename	Ścieżka do pliku ze zmianami.
Dateexecuted	Data aplikacji zmian.
Orderexecuted	Kolejność egzekucji.
Exectype	Wynik egzekucji.
Md5sum	Wynik funkcji skrótu w celu wykrycia nieporządanych zmian.
Description	Opis zmian.
Comments	Komentarze zmian.
Tag	Słowa kluczowe, po których można wyszukać zmiany.
Liquibase	Wersja Liquibase.
Contexts	Konteksty użyte przy wprowadzaniu zmian.
Labels	Nagłówki użyte przy wprowadzaniu zmian.
Deployment_id	Identyfikator zbioru zmian.

Pozostałe z dodatkowych tabel, przechowują dane użytkowników. Ogólne informacje znajdują się w *jhi_user* (Tab. 4.5). Nazwy uprawnień są przechowywane w *jhi_authority* jako jedyny atrybut będący kluczem głównym. Ostatnia z dodanych tabel to *jhi_user_authority*. Utworzona jest ze względu na relację *ManyToMany* pomiędzy dwiema

poprzednimi. Przechowywane są w niej powiązania pomiędzy użytkownikiem oraz uprawnieniami.

Tab. 4.5 Opis atrybutów tabeli *jhi_user*.

Atrybut	Opis
id	Klucz główny.
Login	Unikalna nazwa użytkownika.
Password_hash	Enkryptowane hasło użytkownika.
First_name	Imię użytkownika.
Last_name	Nazwisko użytkownika.
Email	Adres email użytkownika.
Image_url	Ścieżka do zdjęcia powiązanego z kontem użytkownika.
Activated	Wyrażenie wskazujące czy konto jest aktywne.
Lang_key	Klucz wskazujący na język używany przez użytkownika.
Activation_key	Klucz aktywacyjny wysyłany na adres email przy aktywacji.
Reset_key	Klucz resetowania hasła wysyłany na adres email przy resetowaniu hasła.
Created_by	Nazwa użytkownika, który utworzył rekord.
Created_date	Data utworzenia użytkownika.
Reset_date	Data ostatniego resetowania hasła.
Last_modified_by	Użytkownik, który dokonał ostatniej modyfikacji.
Last_modified_date	Data ostatniej modyfikacji.

Dla aplikacji istotne jest powiązanie użytkownika z postaciami. Wymagana jest relacja *OneToMany* pomiędzy tabelami *jhi_user* oraz *player*. Powodem jest fakt, iż w przyszłości zakłada się możliwość posiadania wielu postaci przez każdy użytkownika.

Aby poprawnie skonfigurować relacje pomiędzy graczem a użytkownikiem, należy dodać klucz obcy tabeli *jhi_user* w *player* (Rys. 4.5). Zmiany dokonuje się w plikach konfiguracyjnych liquibase.

```
<column name="user_id" type="bigint">
    <constraints unique="false" nullable="false" uniqueConstraintName="ux_player_user_id" />
</column>
```

Rys. 4.5 Konfiguracja klucza obcego w tabeli *player*.

Następnie, w celu utworzenia relacji, należy powiązać utworzony wcześniej klucz obcy z identyfikatorem tabeli *jhi_user* (Rys. 4.6).

```
<addForeignKeyConstraint baseColumnNames="user_id"
                         baseTableName="player"
                         constraintName="fk_player_user_id"
                         referencedColumnNames="id"
                         referencedTableName="jhi_user"/>
```

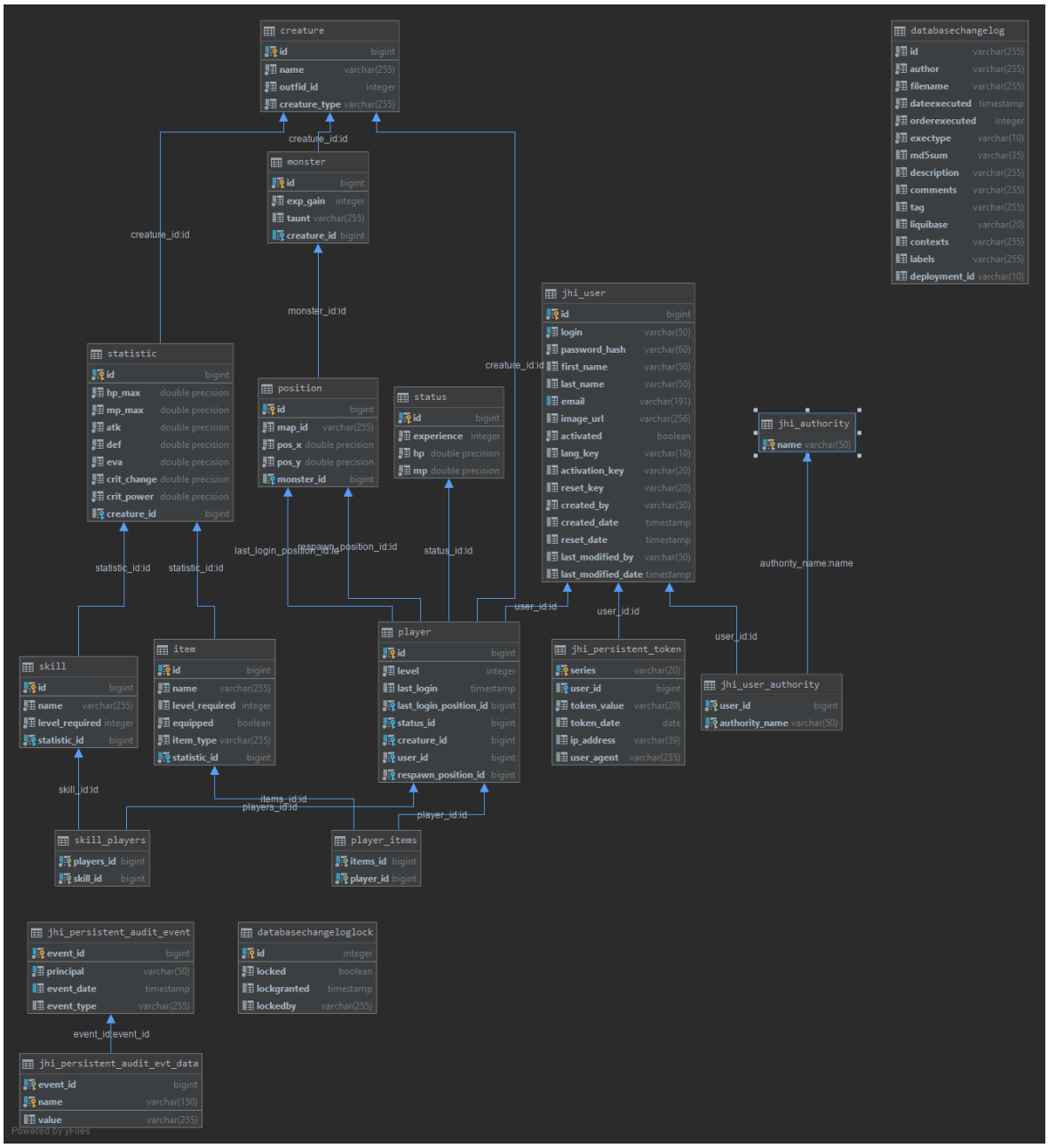
Rys. 4.6 Konfiguracja relacji w *jhi_user* – *player* w liquibase.

Po dokonaniu zmian, trzeba ponownie wygenerować strukturę bazy danych. Ze względu na blokadę zmian w narzędziu liquibase, należy najpierw usunąć schemat a następnie dodać go od nowa wykonując zapytania SQL (Rys. 4.7). Zostało użyte słowo kluczowe *cascade*, gdyż bez niego niemożliwe jest usunięcie schematu. Powodem są istniejące w nim tabele.

```
drop schema public cascade;
create schema public;
```

Rys. 4.7 Zapytania SQL do wyczyszczenia schematu.

Po wykonaniu zapytań, można ponownie uruchomić aplikację. W konsoli widoczne jest powiadomienie informujące, że nie udało się jej uruchomić. Powodem jest brak zmian w konfiguracji ORM. Operacje wykonywane przez liquibase odbywają się wcześniej, dlatego można dokonać wizualizacji wygenerowanej na nowo bazy danych (Rys. 4.8). Widać na niej nowo utworzone powiązanie pomiędzy tabeliami *jhi_user* oraz *player*.



Rys. 4.8 Wizualizacja ostatecznej struktury danych.

Narzędzie jhipster dostarcza również konfiguracji ORM dla utworzonej w JDL-Studio struktury. Aby zdefiniować klasę, która reprezentuje tabele w bazie danych, należy oznaczyć ją adnotacją @Entity. Można dodatkowo określić nazwę odpowiadającej tabeli, nadając wartość polu *name* w adnotacji @Table. Do umożliwienia zapisu w pamięci podręcznej pobieranych danych, oznacza się klasę za pomocą @Cache. W jej parametrze *usage*, można określić strategię działania. Narzędzie jhipster domyślnie wybiera NONSTRICT_READ_WRITE. Jak wskazuje Vlad Mihalcea w „High-Performance Java Persistance” [19], jest to najlepszy wybór dla aplikacji, która głównie wykonuje operacje pobierania danych z bazy. Dane są również przechowywane w Elasticsearch, dlatego klasa oznaczona jest adnotacją Document. W atrybucie indexName, definiuje się nazwę indeksu, do którego zapisywane są dane.

```java
@Entity
@Table(name = "player")
@Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
@org.springframework.data.elasticsearch.annotations.Document(indexName = "player")
public class Player implements Serializable {
```

Rys. 4.9 Oznaczenia przy klasie modelu.

Klasa musi implementować interfejs java.io.Serializable. Informuje on wirtualną maszynę Javy, o możliwości serializowania jej instancji. Bez implementacji tego interfejsu, w takiej sytuacji zostaje wyrzucony wyjątek typu NotSerializableException. Konwencją jest również definicja pola serialVersionUID (Rys. 4.10), która jest informacją na temat aktualnej wersji modelu. W przypadku jej braku, kompilator wygeneruje ją automatycznie.

```java
private static final long serialVersionUID = 1L;
```

Rys. 4.10 Definiowanie UID serializacji.

W dalszej części należy zdefiniować atrybuty, które odpowiadają kolumnom w tabeli. Klucz główny oznacza się adnotacją @Id. W celu umożliwienia automatycznego generowania wartości dodaje się @GeneratedValue. Definiuje się w niej pola *strategy* oraz *generator*. Pierwsze z nich opisuje sposób generowania identyfikatora. W tym przypadku jest to sekwencja. Drugi natomiast wskazuje na narzędzie do tego wykorzystywane, za pomocą wartości atrybutu *generator*. Podaje się w nim nazwę używanego generatora, która jest definiowana w atrybucie *name* adnotacji @SequenceGenerator. Można również nadać początkową wartość oraz rozmiar sekwencji. W tym przypadku nie jest to konieczne. Dodatkowo dodana jest adnotacja @Field, która nadpisuje domyślne ustawienia atrybutu w bazie Elasticsearch. Atrybut *type* z wartością *keyword*, powoduje, że atrybut nie jest dzielony na części w trakcie wyszukiwania.

```java
@Id
@GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "sequenceGenerator")
@SequenceGenerator(name = "sequenceGenerator")
@org.springframework.data.elasticsearch.annotations.Field(type = FieldType.Keyword)
private Long id;
```

Rys. 4.11 Oznaczenia przy kluczu głównym modelu.

Do oznaczenia pozostałych pól, wystarczy nadać im adnotację @Column (Rys. 4.12). Nazwę kolumn definiuje się w atrybucie *name*. Można również określić ograniczenia dla wartości wprowadzanych. Wyróżnia się między innymi:

- NotNull – wartość musi być różna od null.
- NotEmpty – ciąg znaków lub rozmiar kolekcji nie może być pusty.
- Size – długość ciągu znaków musi być wartością pomiędzy minimalną i maksymalną dopuszczalną. Są one definiowane w atrybutach *min* oraz *max*.
- Email – wartość musi być w formacie adresu poczty elektronicznej.

```java
@Column(name = "last_login")
private Instant lastLogin;
```

Rys. 4.12 Oznaczenia przy atrybucie modelu.

Aby zdefiniować relację, dodaje się do atrybutu adnotację:

- @ManyToMany (Rys. 4.13)
- @OneToOne (Rys. 4.14)
- @ManyToOne (Rys. 4.15)
- @OneToMany (Rys. 4.16)

```
@ManyToMany
@Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
@JoinTable(name = "player_items",
           joinColumns = @JoinColumn(name = "player_id", referencedColumnName = "id"),
           inverseJoinColumns = @JoinColumn(name = "items_id", referencedColumnName = "id"))
private Set<Item> items = new HashSet<>();
```

Rys. 4.13 Oznaczenia przy relacji *ManyToMany* w modelu.

```
@OneToOne(optional = false)
@NotNull
@JoinColumn(unique = true)
private Status status;
```

Rys. 4.14 Oznaczenia przy relacji *OneToOne* w modelu.

```
@ManyToOne(optional = false)
@JsonIgnoreProperties("respawnPositionPlayers")
private Position respawnPosition;
```

Rys. 4.15 Oznaczenia przy relacji *ManyToOne* w modelu.

```
@OneToMany(mappedBy = "respawnPosition")
@Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
private Set<Player> respawnPositionPlayers = new HashSet<>();
```

Rys. 4.16 Oznaczenia przy relacji *OneToMany* w modelu.

W przypadku gdy odnosi się do wielu elementów drugiej instancji (Rys. 4.13) (Rys. 4.16), należy dodać adnotację @Cache, by umożliwić zapisywanie pobieranych danych w pamięci podręcznej.

Dla uproszczenia struktury, oznacza się atrybuty adnotacjami @JoinColumn (Rys. 4.14) oraz @JoinTable (Rys. 4.13). Ich zadaniem jest wskazanie referencji tabel. Pierwsza z nich definiuje odniesienia kolumn, natomiast druga tabel.

Korzystając z tej wiedzy, można rozwiązać problem relacji pomiędzy użytkownikiem a graczem. W tym celu należy zdefiniować relację *ManyToOne* w klasie *Player* (Rys. 4.17) oraz *OneToMany* w *User* (Rys. 4.18). Po dokonaniu tych zmian, aplikacja uruchamia się bez błędów.

```
@JsonIgnore
@ManyToOne
private User user;
```

Rys. 4.17 Zmiany w klasie Player przy wiązaniu gracza z użytkownikiem.

```
@OneToMany(cascade = CascadeType.ALL, orphanRemoval = true, mappedBy = "user")
@Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
private Set<Player> players = new HashSet<>();
```

Rys. 4.18 Zmiany w klasie User przy wiązaniu gracza z użytkownikiem.

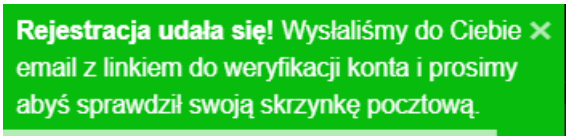
4.2. Użytkownicy w systemie [K.D., J.D.]

Operacje rejestracji, aktywacji konta oraz logowania się nie wymagają wymiany danych w czasie rzeczywistym, dlatego do operacji na użytkownikach w systemie wykorzystywany jest protokół http/1.1. Implementacja podstawowych operacji na użytkownikach wygenerowana jest przez narzędzie jhipster. Aplikacja wykorzystuje Spring Security jako warstwę zabezpieczeń. Jest to biblioteka wchodząca w skład Spring Framework. Do jej najważniejszych zadań należy uwierzytelnianie użytkowników, przechowywanie ich danych i zarządzanie nimi w sesji.

Do łączenia się z serwerem gry przez protokół http/1.1 aplikacja korzysta z biblioteki *axios*. Każdy niezalogowany użytkownik ma możliwość założenia konta w aplikacji poprzez rejestrację (Rys. 4.19).

Rys. 4.19 Widok rejestracji

W tym celu należy podać login, email oraz hasło. Po wciśnięciu przycisku „Zarejestruj”, odpytany jest endpoint */api/register*, a użytkownik zostaje poinformowany o konieczności aktywacji swojego konta (Rys. 4.20). Może tego dokonać poprzez kliknięcie w link aktywacyjny dostarczony pocztą elektroniczną, na adres podany podczas procesu rejestracji. Kliknięcie powoduje odpytanie endpointu */api/activate?key*. W rezultacie nowy użytkownik jest w stanie się zalogować. Jego postać posiada tą samą nazwę co nazwa konta, poziom 1, brak doświadczenia, umiejętności, przedmiotów oraz pozycję startową w świecie gry.



Rys. 4.20 Informacja o udanej rejestracji

By zapewnić prawidłowe działanie rejestracji oraz aktywacji konta, po stronie serwera są wystawione poniższe endpointy:

- */api/register* – przyjmuje zapytania metodą POST. Jako parametr wymaga danych użytych przy rejestracji. Funkcja obsługująca ten endpoint sprawdza poprawność danych, a następnie wysyła wiadomość z linkiem aktywacyjnym. W trakcie procesu walidacji weryfikowane jest czy:

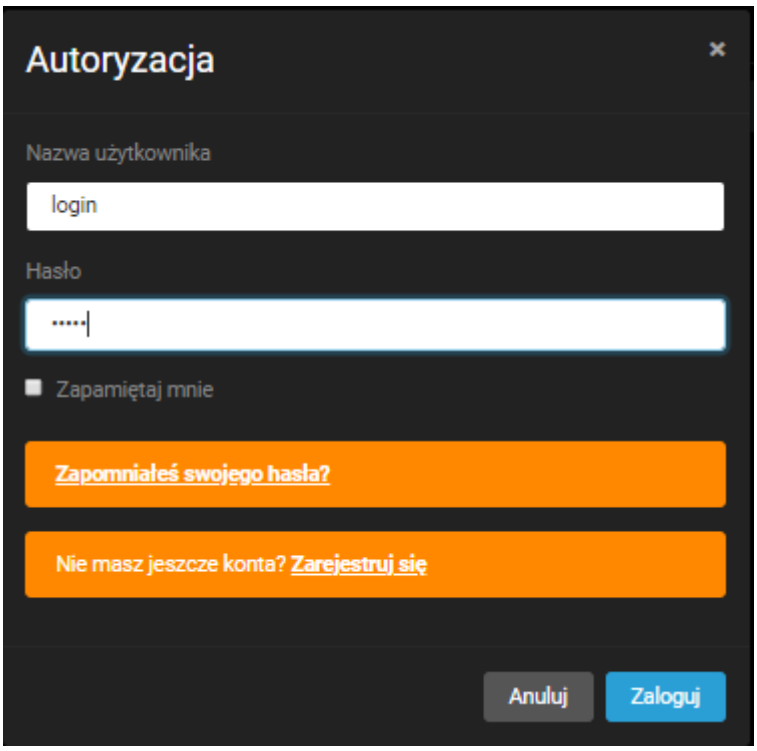
- długość hasła jest poprawna
- login nie został wcześniej użyty
- email nie został wcześniej użyty.

Klucz aktywacyjny generowany jest przy pomocy gotowego narzędzia RandomUtil z biblioteki Apache. Po bezbłędnym przejściu powyższych procesów, nowy użytkownik zapisywany jest w bazie danych z początkowymi statystykami oraz przypisanym kluczem aktywacyjnym. Nie można się jednak wciąż na niego zalogować, dopóki nie zostanie aktywowany.

Przy nieudanej rejestracji, zwracany jest status 400 (Bad Request) w przeciwnym wypadku 200 (OK).

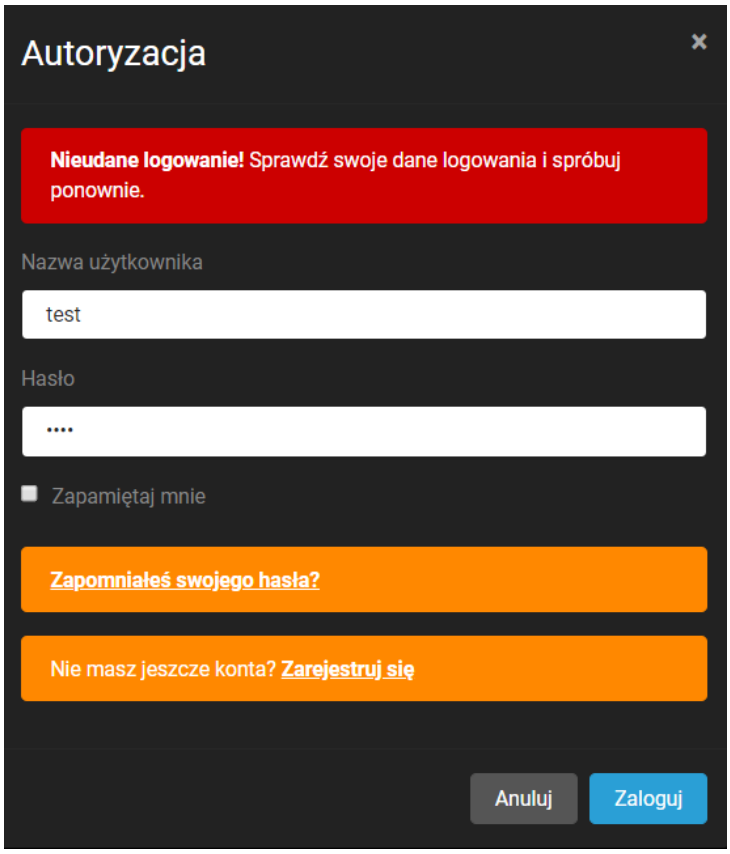
- */api/activate?key=* – przyjmuje zapytania metodą GET. Wymaga wygenerowanego podczas rejestracji klucza, który podaje się w parametrze *key*. Funkcja obsługująca ten endpoint sprawdza czy dany klucz jest przypisany do użytkownika w bazie danych. Jeżeli tak, to aktualizuje go poprzez zmianę wartości atrybutu *isActivated* na *true*. Przy nieudanej aktywacji, zwracany jest status 500 (Internal Server Error) w przeciwnym wypadku 200 (OK).

W celu zalogowania należy przejść na odpowiednio stworzony w tym celu widok (Rys. 4.21)



Rys. 4.21 Widok logowania do gry

Należy podać login oraz hasło użyte podczas rejestracji i kliknąć przycisk „Zaloguj”. W rezultacie odpytany jest endpoint */api/authentication*. Dodatkowo jest możliwość zapamiętania danych logowania, tak by podczas przyszłych wizyt na stronie, nie trzeba było wprowadzać ich za każdym razem. Jeśli logowanie nie powiedzie się, zostanie wyświetlony stosowny komunikat (Rys. 4.22).



Rys. 4.22 Widok niepowodzenia logowania do gry

Przy udanym logowaniu następuje przekierowanie na widok świata gry (Rys. 4.23).

Rys. 4.23 Widok świata gry

By zapewnić prawidłowe działanie logowania, po stronie serwera wystawiony jest poniższy endpoint:

- */api/authentication* – przyjmuje zapytania metodą POST. Jako dane wymagane są login, hasło oraz opcjonalnie token *rememberMe*. Endpoint ten dostarczony jest przez bibliotekę Spring Security. Odpowiada za autoryzację użytkownika. Przy niepowodzeniu zwracany jest status 401 (Unauthorized) w przeciwnym wypadku 200 (OK).

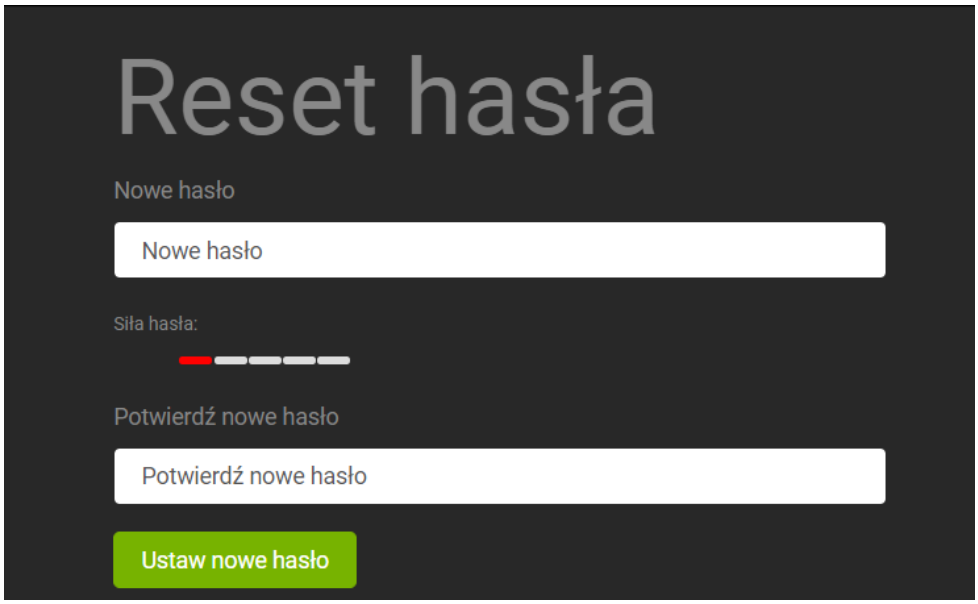
W sytuacji, gdy użytkownik zapomni swojego hasła, może wykorzystać opcję jego zresetowania (Rys. 4.24).

Rys. 4.24 Przycisk „Zapomniałeś swojego hasła?”

Po kliknięciu w przycisk „Zapomniałeś swojego hasła?”, otwiera się formularz, którego wypełnienie pozwala na zresetowanie hasła (Rys. 4.25).

Rys. 4.25 Widok resetowania hasła

W jedynym polu w tym formularzu należy wpisać adres pocztowy użyty podczas procesu rejestracji, a następnie kliknąć przycisk „Reset hasła”. Skutkiem tego działania jest odpytanie endpointu */api/account/reset-password-init* oraz wysłanie wiadomości z linkiem pozwalającym na zresetowanie zapomnianego hasła. Po kliknięciu, otwierany jest widok z formularzem pozwalającym na zmianę hasła (Rys. 4.26). Należy wprowadzić nowe, a następnie zatwierdzić klikając w „Ustaw nowe hasło”. W rezultacie odpytany jest endpoint */api/account/reset-password-finish.*



Rys. 4.26 Formularz resetowania hasła

By zapewnić prawidłowe działanie operacji resetu hasła, po stronie serwera są wystawione poniższe endpointy:

- */api/account/reset-password-init* – przyjmuje zapytania metodą POST. Wymaga adresu email użytego podczas rejestracji. Funkcja obsługująca ten endpoint sprawdza czy adres pocztowy występuje w bazie danych, a następnie wysyła wiadomość z linkiem oraz kluczem resetującym hasło. Jest on generowany przy pomocy gotowego

narzędzia RandomUtil z biblioteki Apache, a następnie przypisywany do użytkownika.
Przy niepowodzeniu, zwracany jest status 400 (Bad Request) w przeciwnym wypadku 200 (OK).

- */api/account/reset-password-finish* – przyjmuje zapytania metodą POST. Wymaga nowego hasła oraz klucza weryfikującego. Funkcja obsługująca ten endpoint sprawdza, czy dane są poprawne. W pierwszej kolejności weryfikuje czy hasło jest w odpowiednim formacie. Jeżeli dane są poprawne, zmienia hasło użytkownikowi, do którego przypisany jest klucz weryfikujący przesłany w zapytaniu.
  Przy niepowodzeniu, zwracany jest status 400 (Bad Request) w przeciwnym wypadku 200 (OK).

Do poprawnego działania systemu uwierzytelniania potrzebne są role. W aplikacji istnieją dwie podstawowe:

- *ROLE_USER* – podstawowa rola nadawana wszystkim użytkownikom systemu od razu po poprawnym przejściu procesu rejestracji.
- *ROLE_ADMIN* – rola wskazująca na administratora systemu. Pozwala na dodatkowe działania w aplikacji

By system uwierzytelniania działał poprawnie, wymagane jest by każdy użytkownik w aplikacji miał przypisaną przynajmniej jedną rolę.
Takie działanie pozwala na proste zabezpieczanie endpointów przed operacjami ze strony niepożądanych podmiotów (Rys. 4.27).

```
.antMatchers( …antPatterns: "/management/**").hasAuthority(AuthoritiesConstants.ADMIN);
```

Rys. 4.27 Przykład zabezpieczenia endpointu

Przy pierwszym uruchomieniu aplikacji, do bazy danych dodawani są użytkownicy. W tym celu aplikacja korzysta z narzędzia liquibase, które automatyzuje operacje bazodanowe. Takie operacje są zaimplementowane w celu przyspieszenia wstępnej konfiguracji. Dzięki nim nie ma potrzeby dodawania użytkowników ręcznie.

W bazie danych istnieją następujący użytkownicy:

- System – użytkownik posiadający rolę zarówno administratora oraz zalogowanego użytkownika. Jest wykorzystywany do operacji, wykonywanych przez aplikację.
- Anonymoususer – użytkownik nie posiadający żadnych ról. Jest nim każda niezalogowana osoba w aplikacji. Jest wykorzystywany w celu rozróżnienia użytkowników zalogowanych od niezalogowanych.
- Administratorzy – użytkownicy posiadający rolę administratora oraz zalogowanego użytkownika.

4.3. Konfiguracja Phaser [K.D., J.D.]

Phaser jest dodany do aplikacji jako komponent o nazwie *Game*. Wczytywany jest na stronie głównej w komponencie *Home* (Rys. 4.28). Jest uruchamiany przy wejściu na url „/” (Rys. 4.29). Metoda *render* zwraca widok reprezentujący komponent gry.

```
render() {
  return (
    <Row>
      <Col>
        {this.props.isAuthenticated && <Game />}
        {!this.props.isAuthenticated && <BestPlayer />}
      </Col>
      <Col>{this.props.isAuthenticated && <Chat />}</Col>
    </Row>
  );
}
```

Rys. 4.28 Użycie komponentu *Game*.

```
const Routes = () => (
  <div className="view-routes">
    <Switch>
      <ErrorBoundaryRoute path="/login" component={Login} />
      <ErrorBoundaryRoute path="/logout" component={Logout} />
      <ErrorBoundaryRoute path="/register" component={Register} />
      <ErrorBoundaryRoute path="/activate/:key?" component={Activate} />
      <ErrorBoundaryRoute path="/reset/request" component={PasswordResetInit} />
      <ErrorBoundaryRoute path="/reset/finish/:key?" component={PasswordResetFinish} />
      <PrivateRoute path="/admin" component={Admin} hasAnyAuthorities={[AUTHORITIES.ADMIN]} />
      <PrivateRoute path="/account" component={Account} hasAnyAuthorities={[AUTHORITIES.ADMIN, AUT
      <PrivateRoute path="/entity" component={Entities} hasAnyAuthorities={[AUTHORITIES.USER]} />
      <ErrorBoundaryRoute path="/" exact component={Home} />
      <ErrorBoundaryRoute component={PageNotFound} />
    </Switch>
  </div>
);
```

Rys. 4.29 Definicja ścieżek na stronie.

Pole *this.props.isAuthenticated*, jest pobierane z narzędzia redux. Przyjmuje wartość *true*, gdy użytkownik jest zautoryzowany. Jego wartość jest ustawiana w trakcie procesu logowania.

Komponent *Game* zarządza stanem oraz korzysta z niego, dlatego jest połączony z narzędziem redux metodą *connect* (Rys. 4.30). W argumentach przyjmuje funkcję *mapStateToProps* oraz obiekt *mapDispatchToProps*. Pierwszy odpowiada za wartości pobierane z reduxa, natomiast drugi za akcje, które są na nim wykonywane.

```
import React from 'react';
import { connect } from 'react-redux';
import { initGame } from 'app/modules/game/game.reducer';

export interface IGameProps extends DispatchProps, StateProps {}

export class Game extends React.Component<IGameProps> {
  componentDidMount() {
    this.props.initGame();
  }

  render() {
    return <div id="board" />;
  }
}

const mapStateToProps = storeState => ({
  config: storeState.game.config,
  instance: storeState.game.instance
});

const mapDispatchToProps = { initGame };

type StateProps = ReturnType<typeof mapStateToProps>;
type DispatchProps = typeof mapDispatchToProps;

export default connect(
  mapStateToProps,
  mapDispatchToProps
)(Game);
```

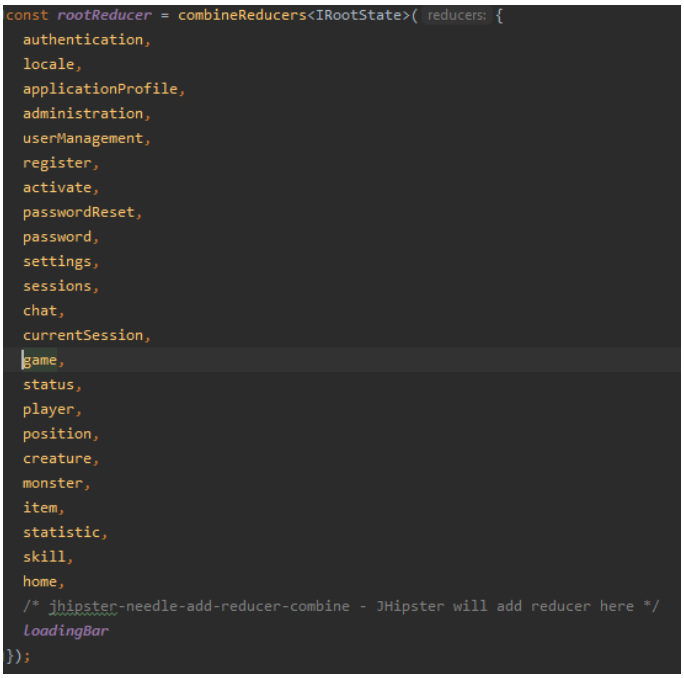
Rys. 4.30 Definicja ścieżek na stronie.

Game ma zdefiniowaną metodę *componentDidMount.* Jest ona wykonywana w momencie, gdy komponent jest załadowany. Po wywołaniu inicjuje grę funkcją *initGame*, która wysyła akcję INIT_GAME do redux (Rys. 4.31).

```
export const initGame = () => ({
  type: ACTION_TYPES.INIT_GAME
});
```

Rys. 4.31 Definicja metody *initGame*.

Akcje są rozpoznawane w elemencie nazywanym reduktorem. Każdy komponent zarządzający stanem definiuje własny w postaci funkcji, która rozpoznaje akcje i edytuje stan aplikacji na jej podstawie. Są one łączone w konfiguracji narzędzia redux (Rys. 4.32).

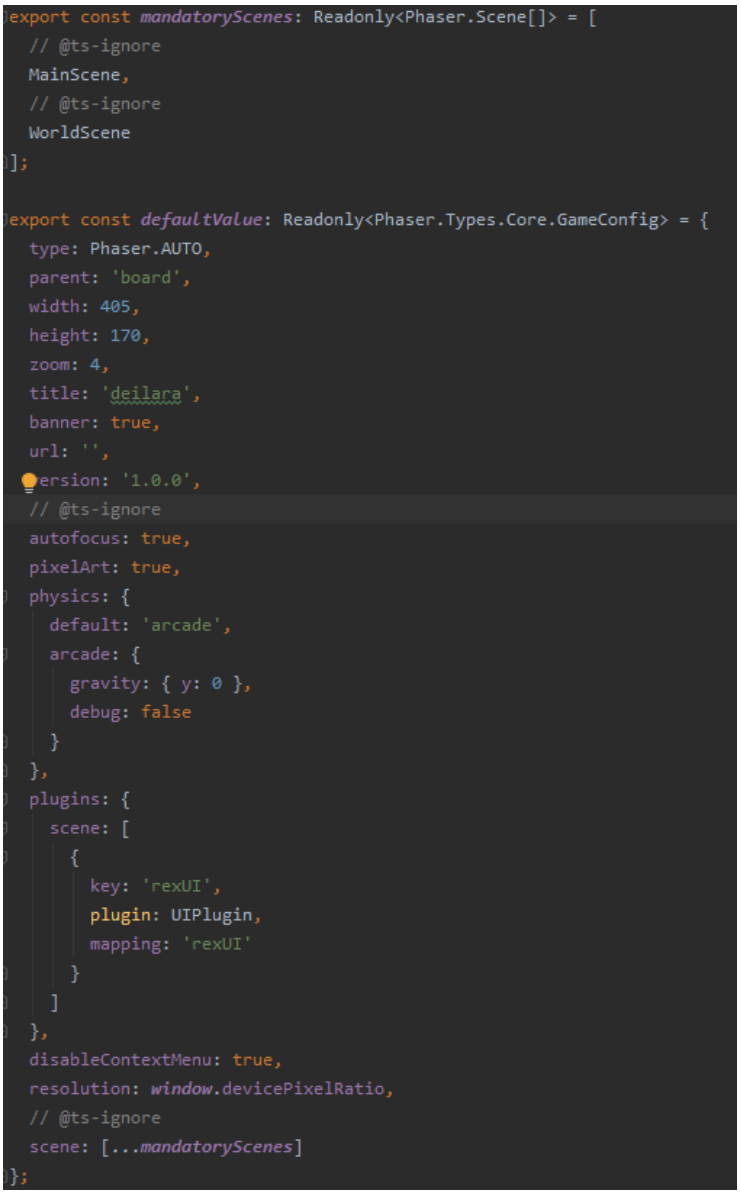


Rys. 4.32 Importowanie reduktorów.

Reduktor *Game* rozpoznaje tylko akcję INIT_GAME (Rys. 4.33.). Gdy zostanie wywołana, tworzona jest nowa instancja gry wraz z przekazaną konfiguracją (Rys. 4.34.).

```
export default (state: GameState = initialState, action): GameState => {
  switch (action.type) {
    case ACTION_TYPES.INIT_GAME:
      return {
        ...state,
        instance: new Phaser.Game(state.config)
      };
    default:
      return state;
  }
};
```

Rys. 4.33 Reduktor komponentu *Game*.



```
export const mandatoryScenes: Readonly<Phaser.Scene[]> = [
  // @ts-ignore
  MainScene,
  // @ts-ignore
  WorldScene
];

export const defaultValue: Readonly<Phaser.Types.Core.GameConfig> = {
  type: Phaser.AUTO,
  parent: 'board',
  width: 405,
  height: 170,
  zoom: 4,
  title: 'deilara',
  banner: true,
  url: '',
  version: '1.0.0',
  // @ts-ignore
  autofocus: true,
  pixelArt: true,
  physics: {
    default: 'arcade',
    arcade: {
      gravity: { y: 0 },
      debug: false
    }
  },
  plugins: {
    scene: [
      {
        key: 'rexUI',
        plugin: UIPlugin,
        mapping: 'rexUI'
      }
    ]
  },
  disableContextMenu: true,
  resolution: window.devicePixelRatio,
  // @ts-ignore
  scene: [...mandatoryScenes]
};
```

Rys. 4.34 Konfiguracja gry.

Konfiguracja obejmuje:

- *type* – typ renderowania, domyślnie Canvas.
- parent – identyfikator elementu na widoku, do którego zostanie dodany widok gry
- width – szerokość widoku gry
- height – długość widoku gry
- zoom – przybliżenie widoku gry
- title – tytuł gry
- banner – wykorzystanie banneru podczas uruchamiania gry
- url – adres URL gry
- version – wersja gry
- autofocus – automatyczne wskazywanie na oknie gry
- pixelArt – korzystanie z tekstur niskiej rozdzielczości
- physics – ustawienia fizyki
- plugins – dodatkowe wtyczki

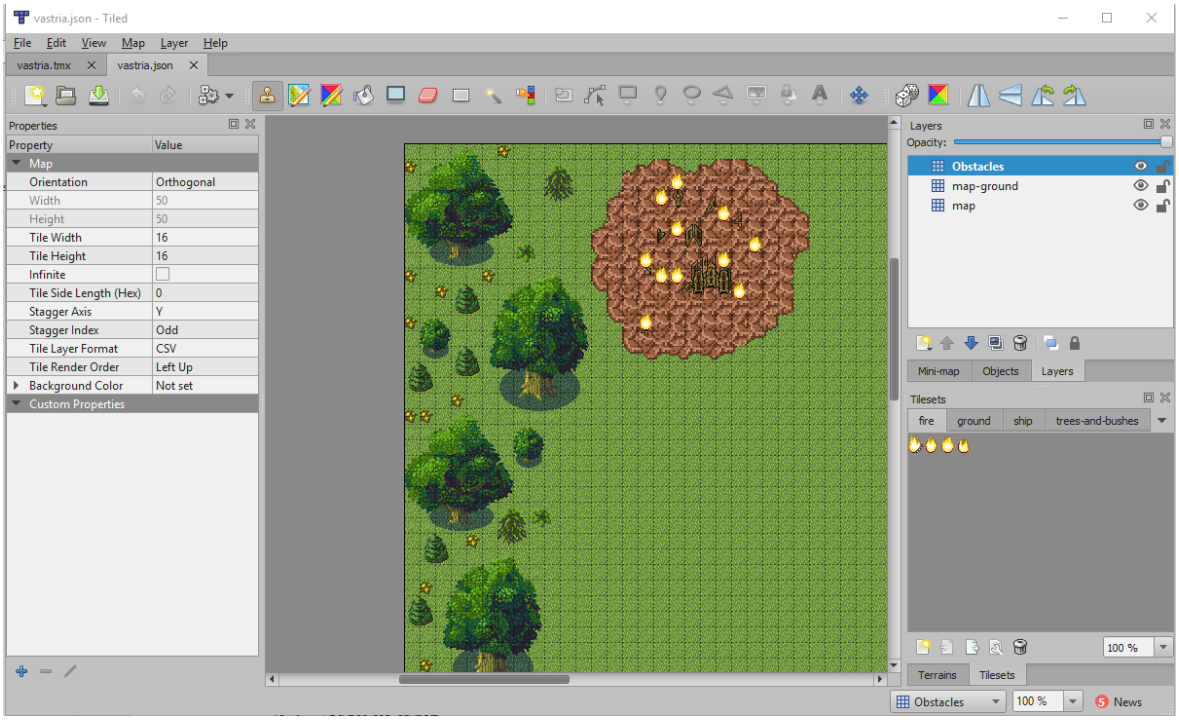
- disableContextMenu – wyłączenie zdarzenia *contextMenu*
- resolution – ustawienia rozdzielczości
- scene – wykorzystywane sceny

Każda scena reprezentuję inną mapę. Wyjątkiem jest *MainScene*, która odpowiada za załadowanie zawartości oraz uruchomienie odpowiedniej mapy danemu użytkownikowi (Rys. 4.35). Wysyła żądanie do serwera odnośnie ostatniej pozycji gracza, a następnie uruchamia scenę, na której ostatnio się znajdował. Ta natomiast korzystając z informacji jakie otrzymała, inicjuje grę. Aby zarządzać odpowiednio zawartością, implementuje się metody:

- preload – odpowiada za akcje przed załadowaniem sceny.
- create – odpowiada za akcje przy tworzeniu sceny.
- update – odpowiada za akcje w pętli gry.

4.4. Odczytywanie mapy [K.D., J.D.]

Znacznym ułatwieniem przy tworzeniu map do gry, jest narzędzie *Tiled* (Rys. 4.35). Korzystając z importowanych plików z grafiką, można umieszczać elementy na siatce reprezentującej mapę. We właściwościach definiuje się wszystkie rozmiary oraz format pliku.



Rys. 4.35 Widok narzędzia Tiled

Tworzona mapa jest podzielona na wiele warstw. Działanie to ma na celu oddzielenie elementów, dla których przewiduje się te same zachowania. Przykładem jest warstwa przeszkód. Dzięki temu można zabronić przejścia postaci przez wszystkie jej elementy, zamiast każdego z osobna.

Wykorzystywane pliki graficzne to tak zwane tilemapy (Rys. 4.36). Korzysta się z nich w podobny sposób do palety. Grafika podzielona jest na fragmenty równej długości.

Rys. 4.36 Przykład tilemapy

W narzędziu *Tiled*, można importować tilemapę korzystając z opcji import. W trakcie nadawana jest jej nazwa oraz rozmiar siatki. W rezultacie tilemapa jest gotowa do użytku (Rys. 4.37).

Rys. 4.37 Zaimportowana tilemapa o nazwie *trees-and-brushes*

Dzięki narzędziu *Tiled*, można zaznaczyć obszar na tilemapie, a następnie umieścić go wielokrotnie na aktualnie wybranej warstwie mapy. Po jej stworzeniu, można ją wyeksportować do pliku o rozszerzeniu *json*. W grze, wczytuje się go przy pomocy biblioteki Phaser po stronie klienta. Dzieje się to podczas ładowania elementów w *MainScene* (Rys. 4.38). Wykorzystywana do tego metoda *tilemapTiledJSON*, w pierwszym argumencie przyjmuje identyfikujący ciąg znaków, w drugim natomiast ścieżkę do pliku.

```
this.load.tilemapTiledJSON('vastria', '../../../content/tilesets/vastria.json');
```

Rys. 4.38 Importowanie mapy

Należy również zaimportować pliki z grafiką wykorzystywane w mapie (Rys. 4.39). Służy do tego metoda image, która przyjmuje identyczne argumenty co *tilemapTiledJSON*.

```
this.load.image('map_tiles', '../../../content/tilesets/tilemaps/map.png');
this.load.image('map_fire', '../../../content/tilesets/tilemaps/fire.png');
this.load.image('map_ground', '../../../content/tilesets/tilemaps/ground.png');
this.load.image('map_ship', '../../../content/tilesets/tilemaps/ship.png');
this.load.image('map_trees-and-bushes', '../../../content/tilesets/tilemaps/trees-and-bushes.png');
```

Rys. 4.39 Importowanie grafik

Wszystkie powyższe zmiany powinny być umieszczone w metodzie *preload*, ze względu na fakt, iż wczytywanie musi odbywać się w pierwszej kolejności (Rys. 4.40).

```
preload() {
  // Runs once, loads up assets like images and audio
  this.load.image('map_tiles', '../../../content/tilesets/tilemaps/map.png');
  this.load.image('map_fire', '../../../content/tilesets/tilemaps/fire.png');
  this.load.image('map_ground', '../../../content/tilesets/tilemaps/ground.png');
  this.load.image('map_ship', '../../../content/tilesets/tilemaps/ship.png');
  this.load.image('map_trees-and-bushes', '../../../content/tilesets/tilemaps/trees-and-bushes.png');
  this.load.image('wolf', '../../../content/images/wolf.png');
  this.load.image('demon', '../../../content/images/demon.png');
  this.load.image('ent', '../../../content/images/dark-ent.png');
  this.load.image('golem', '../../../content/images/coppergolem.png');
  this.load.image('worm', '../../../content/images/giant-worm.png');
  this.load.image('sword', '../../../content/images/attack-icon.png');
  this.load.tilemapTiledJSON('vastria', '../../../content/tilesets/vastria.json');
  this.load.spritesheet('player', '../../../content/characters/RPG_assets.png', { frameWidth: 16, frameHeight: 16 });
  this.stompFactory = StompFactory.getInstance();
  this.stompFactory.getStompClientObservable().subscribe( next: val => {
    this.addChatListeners();
    const worldName = axios.get<string>( url: `api/getLastPosition`);
    worldName
      .then(res => {
        this.scene.start(res.data.mapId,  offset: { lastPosition: res.data });
      })
      .catch(err => {
        alert('Brak obsługi działania na wielu kartach!');
      });
  });
}
```

Rys. 4.40 Metoda preload głównej sceny

Po wczytaniu komponentów są gotowe do użycia. Wykorzystuje się je podczas tworzenia sceny, dla której mapa powstała (Rys. 4.41). W pierwszej kolejności inicjuje się ją wywołując metodę *tilemap*. Argumentem jest obiekt konfiguracyjny. Na potrzebę gry wystarczy uzupełnić atrybut *key*, którego wartością jest identyfikator wczytanej wcześniej mapy. Następnie inicjuje się zdjęcia odpowiadające tilemapom wykorzystanym na niej, za pomocą metody *addTilesetImage*. W pierwszym argumencie przyjmuje identyfikator wczytanego obrazu, natomiast w drugim nazwę wpisaną w narzędziu Tiled podczas importowania odpowiadającej tilemapy. Aby ułatwić zarządzanie, przypisuje się je do tablicy. W dalszym ciągu inicjowane są warstwy za pomocą metody *createStaticLayer*. W pierwszym argumencie przyjmuje nazwę, w drugim użyte tilemapy, a w ostatnich dwóch przesunięcie ich względem początku układu współrzędnych. Przeszkody są traktowane w sposób wyjątkowy. Przypisuje się je do zmiennej, która następnie jest przekazywana do serwisów odpowiadających za zarządzanie grą.

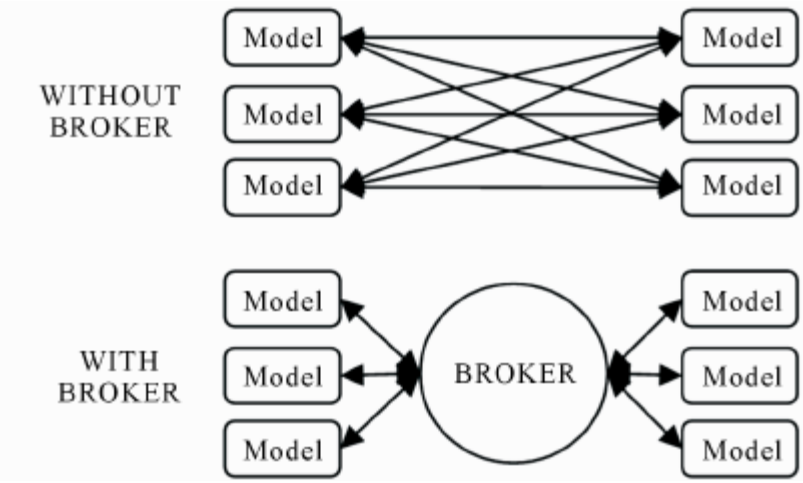
```
create(initParams) {
  // Creating map
  this.map = this.make.tilemap({ key: 'vastria' });
  // adding tilesets to map, first parameter is the name in JSON file of tileset, second is name of timeset fro
  const fire_tiles = this.map.addTilesetImage('fire', 'map_fire', 15, 20);
  const ground_tiles = this.map.addTilesetImage('ground', 'map_ground', 16, 16);
  const ship_tiles = this.map.addTilesetImage('ship', 'map_ship', 16, 16);
  const trees_and_bushes_tiles = this.map.addTilesetImage('trees-and-bushes', 'map_trees-and-bushes', 16, 16);
  const tilests = [fire_tiles, ground_tiles, ship_tiles, trees_and_bushes_tiles];

  // Creating layers where player cannot move
  // First parameter is layer from layer JSON file, second are tilesets which we want to include with that laye
  const obstacles_level = this.map.createStaticLayer('Obstacles', tilests, 0, 0);
  // Creating layers where player can move
  this.map.createStaticLayer('map', tilests, 0, 0);
  this.map.createStaticLayer('map-ground', tilests, 0, 0);

  // make obstacles unable to collide
  obstacles_level.setCollisionByExclusion([-1]);

  const configuration = defaultValue;
  configuration.obstacles.push(obstacles_level);
  configuration.lastPosition = initParams.lastPosition;
  this.manager = new InitService(this, configuration);
  this.manager.initWorld();
  this.physics.world.bounds.width = this.map.widthInPixels;
  this.physics.world.bounds.height = this.map.heightInPixels;
}
```

Rys. 4.41 Metoda create pierwszej sceny gry

4.5. Wyświetlanie danych w czasie rzeczywistym [K.D., J.D.]

Protokół websocket jest przeznaczony do przesyłania komunikatów w czasie rzeczywistym. W związku z tym wykorzystany jest w projekcie do operacji współdzielenia graczy i potworów. Pakiet spring framework dostarcza technologię stomp, która rozszerza podstawowe możliwości websocket. Najważniejszą jest wykorzystanie tzw. Message Broker. Technologia ta wspiera wymianę komunikatów między klientem a serwerem. Serwer informuje o pojawieniu się nowej wiadomości wszystkim, którzy są nią zainteresowani. W modelu, który używałby protokołu http/1.1, to klienci musieliby odpytywać serwer co jakiś czas o nowe wiadomości (Rys. 4.42). Takie zachowanie generowałoby niepożądany ruch w sieci, który w konsekwencji mógłby znacząco spowolnić serwer.

Aplikacja została skonfigurowana w taki sposób, by każde zapytanie wysyłane na endpoint zawierający w swoim adresie */topic/* było przetwarzane przez Message Brokera.



Rys. 4.42 Przepływ danych z brokerem oraz bez [20]

W celu skonfigurowania przepływu wiadomości na serwerze wystarczy umieścić dwie adnotacje przed definicją metody znajdującej się w kontrolerze (Rys. 4.43).

```
@MessageMapping("/topic/position/connect")
@SendTo("/topic/players/current")
```

Rys. 4.43 Konfiguracja przepływu wiadomości po stronie serwera

Adnotacja MessageMapping przyjmuje w argumencie adres, na którym serwer nasłuchuje wiadomości przychodzących. Natomiast argument adnotacji SendTo wskazuje adres, na który zostaną przesłane dane zwrócone w metodzie. Takie oznaczenia są wystarczające by właściwie skonfigurować przepływ wiadomości po stronie serwera.

Analogicznie do MessageMapping w języku javascript wywołuje się funkcję subscribe z biblioteki stomp (Rys. 4.44). W pierwszym argumencie definiowany jest adres, na którym klient nasłuchuje wiadomości przychodzących. Drugi argument to funkcja przetwarzająca wspomniane wiadomości.

```
this._client.subscribe( options: '/topic/players/current', players => {
```

Rys. 4.44 Konfiguracja przepływu wiadomości po stronie klienta

Analogicznie do adnotacji SendTo, w języku javascript jest wywołuje się funkcję send z biblioteki stomp (Rys. 4.45). W pierwszym argumencie definiowany jest adres, na który zostaną przesłane dane z drugiego argumentu.

```
this._client.send(
  '/topic/position/change',
  JSON.stringify( value: {
    playerId: null,
    x,
    y,
    flipX,
    attacking: this._isAttacking
  }),
  {}
);
```

Rys. 4.45 Aktualizacja pozycji gracza z poziomu klienta

Każdy użytkownik w aplikacji łączący się z serwerem przez protokół websocket otrzymuje unikalny identyfikator sesji. Takie połączenie następuje od razu po zalogowaniu się do gry. Użytkownik ten wysyła zapytanie do serwera na endpoint */topic/position/connect/* w celu ustanowienia połączenia oraz pobrania statystyk postaci. Jest ona dołączana do kolekcji aktualnych graczy w grze. Następnie zostaje wysłana na endpoint */topic/players/current/*, który jest nasłuchiwany przez wszystkich zalogowanych użytkowników (Rys. 4.46). Odebraną listę następnie przetwarza się w celu wyświetlenia nowego gracza.

```
this._client.subscribe( options: '/topic/players/current', players => {
  const sessionId = StompFactory.getInstance()
    .getSockClient()
    ._transport.url.split( separator: '/')[6];
  JSON.parse(players.body).forEach(player => {
    if (player.sessionId === sessionId && !this.playerService.isPlayerCreated()) {
      player.x = this._config.lastPosition.posX;
      player.y = this._config.lastPosition.posY;
      this.createLevelStatus(player);
      this.playerService.createPlayer(player);
      this.enemyService.configureEnemies();
    } else if (player.sessionId !== sessionId) {
      this.playerService.createOtherPlayer(player);
    }
  });
```

Rys. 4.46 Kod odpowiadający za inicjalizację postaci

Funkcja tworząca nowego gracza na mapie analizuje otrzymaną z serwera kolekcję wszystkich graczy. Sprawdza każdego z nich i w zależności od spełnionych warunków podejmuje określone działania. Kiedy identyfikator sesji gracza aktualnie analizowanego w kolekcji zgadza się z identyfikatorem klienta wykonującego skrypt:

- grafika postaci zostaje dodana na mapę
- potwory zostają pobrane
- konfigurowany jest status poziomu gracza

W sytuacji, kiedy to inny użytkownik się połączy, dodana zostaje grafika jego postaci na mapę gry (Rys. 4.47).

Rys. 4.47 Załadowana mapa gry

Grafika postaci jest kontenerem. W jego skład wchodzą obrazy, które są łączone w jedną całość. Skutkiem tych działań jest obserwowana przez gracza postać wojownika z mieczem i nazwą nad głową. W rzeczywistości są to oddzielne obrazy oraz tekst odpowiednio umieszczony w kontenerze. Tworzeniem postaci zajmuje się funkcja *createPlayer* (Rys. 4.48).

```
public createPlayer(playerInfo: any) {
  this._playerSprite = this._world.physics.add.sprite(0, 0, 'player', 6);
  let playerName = this._world.add.text(-20, -20, playerInfo.characterName, { font: '8px roboto', align: 'center' });
  PlayerService._playerContainer = this._world.add.container(playerInfo.x, playerInfo.y);
  PlayerService._playerContainer.setSize(16, 16);
  PlayerService._playerContainer.add(this._playerSprite);
  PlayerService._playerContainer.add(playerName);
  this._world.physics.world.enable(PlayerService._playerContainer);
  this.addPlayerSword();
  this.configureCamera();
  PlayerService._playerContainer.weapon.x = 0;
  PlayerService._playerContainer.weapon.y = 7;
  PlayerService._playerContainer.weapon.flipY = true;
  PlayerService._playerContainer.weapon.flipX = true;
  PlayerService._playerContainer.body.setCollideWorldBounds(true);
  this._isPlayerCreated = true;
  this.configurePlayerAnimation();
  this._playerCursors = this._world.input.keyboard.createCursorKeys();
  this._config.obstacles.forEach(layer => this._world.physics.add.collider(PlayerService._playerContainer, layer));
}
```

Rys. 4.48 Funkcja createPlayer

Po zalogowaniu do gry, konfigurowany jest również status poziomu gracza (Rys. 4.49).

Rys. 4.49 Status poziomu gracza

Za jego konfigurację odpowiedzialna jest funkcja *createLevelStatus* (Rys. 4.50)

```
private createLevelStatus(player: any) {
  let label = this._world.add.text(325, 0, 'Level: ' + player.level, { font: '17px roboto', align: 'center' });
  label.setScrollFactor(0);
  this._client.subscribe( options: '/user/queue/player/currentLevel', frame => {
    if (label.text !== 'Level: ' + frame.body) {
      let toast = this._world.rexUI.add
        .toast({
          x: PlayerService._playerContainer.x,
          y: PlayerService._playerContainer.y,

          background: this._world.rexUI.add.roundRectangle(0, 0, 2, 2, 10, COLOR_PRIMARY),
          text: this._world.add.text(0, 0, '', {
            fontSize: '12px'
          }),
          space: {
            left: 8,
            right: 8,
            top: 6,
            bottom: 6
          }
        })
        .show('You advanced to level ' + frame.body);
      label.setText('Level: ' + frame.body);
    }
  });
```

Rys. 4.50 Funkcja *createLevelStatus*

Oprócz samego dodawania statusu poziomu gracza, klient nasłuchuje również na */user/queue/player/currentLevel/*. Serwer wysyła wiadomość do gracza na podany adres w momencie, kiedy awansuje on na nowy poziom. Gdy to następuje, oczom gracza ukazuje się animowane powiadomienie oraz zmienia się numer poziomu w statusie (Rys. 4.51). Do wyświetlenia tego komunikatu użyta została funkcja dostarczona z wtyczki rexUI kompatybilnego z silnikiem phaserJS.

Rys. 4.51 Powiadomienie o awansie

Każdy ruch użytkownika powoduje wysłanie komunikatu na endpoint serwera */topic/position/change* informującego o zmianie aktualnej pozycji (Rys. 4.45).

Po odebraniu informacji, serwer wyszukuje postać nadawcy w kolekcji aktywnych graczy (Rys. 4.52)

```
PlayerDTO oldPlayer = gameService.getPlayers() List<PlayerDTO>
    .stream() Stream<PlayerDTO>
    .filter(el -> Objects.equals(headers.getSessionId(), el.getSessionId())) Stream<PlayerDTO>
    .findAny() Optional<PlayerDTO>
    .orElse( other: null);
```

Rys. 4.52 Fragment kodu wyszukującego nadawcę

Następnie stara pozycja gracza jest nadpisywana przez nową (Rys. 4.53) i wysyłana na endpoint */topic/players/position/change*, który jest nasłuchiwany przez wszystkich zalogowanych użytkowników.

```
if (oldPlayer != null) {
    oldPlayer.setFlipX(positionDTO.getFlipX());
    oldPlayer.setX(positionDTO.getX());
    oldPlayer.setY(positionDTO.getY());
    oldPlayer.setAttacking(positionDTO.isAttacking());
    return oldPlayer;
}
```

Rys. 4.53 Fragment kodu nadpisujący pozycję gracza

Odebrane informacje o zmienionej pozycji gracza są wizualizowane jako animacja grafiki postaci będącej inicjatorem operacji (Rys. 4.54).

```
this._client.subscribe( options: '/topic/players/position/change', frame => {
  this.playerService
    .getOtherPlayers()
    .getChildren()
    .forEach(player => {
      const playerInfo = JSON.parse(frame.body);
      if (player.sessionId === playerInfo.sessionId) {
        player.sprite.flipX = playerInfo.flipX;
        if (playerInfo.x > player.x) {
          player.sprite.anims.play('right', true);
          this.playerService.setPositionRight(player.sprite, player.weapon);
        } else if (playerInfo.x < player.x) {
          player.sprite.anims.play('left', true);
          this.playerService.setPositionLeft(player.sprite, player.weapon);
        } else if (playerInfo.y > player.y) {
          player.sprite.anims.play('down', true);
          this.playerService.setPositionDown(player.sprite, player.weapon);
        } else if (playerInfo.y < player.y) {
          player.sprite.anims.play('up', true);
          this.playerService.setPositionRight(player.sprite, player.weapon);
        }
        player.attacking = playerInfo.attacking;
        player.x = playerInfo.x;
        player.y = playerInfo.y;
        player.lastupdate = new Date().getMilliseconds();
      }
    });
});
```

Rys. 4.54 Fragment kodu odpowiedzialny za animacje

Inicjator operacji jest znajdowany za pomocą identyfikatora sesji. Tak znaleziona postać jest animowana w odpowiednim kierunku.

Po połączeniu gracza do gry, pobierane są również bieżące pozycje potworów na mapie. W tym celu klient wysyła żądanie do serwera w celu pobrania aktualnej ich kolekcji (Rys. 4.55).

```
private spawnEnemies() {
  this._client.send('/topic/monsters/load', res => {
    this.createEnemies(res);
  });
}
```

Rys. 4.55 Funkcja wysyłająca żądanie pobrania danych potworów

Serwer przesyła żądaną kolekcję na endpoint */topic/monsters/create/update/*, który jest nasłuchiwany przez wszystkich zalogowanych graczy (Rys. 4.56).

```
this._client.subscribe( options: '/topic/monsters/create/update', res => {
  this.createEnemies(res);
});
```

Rys. 4.56 Fragment kodu pobierający pozycje potworów

Odebrane dane potworów są przetwarzane u każdego użytkownika w celu ich wyświetlenia. W przypadku gdy któryś nie istnieje, zostaje dodany i zwizualizowany w postaci odpowiedniej grafiki na mapie (Rys. 4.57).

```
if (!existingEnemy) {
  const enemyContainer = this._world.add.container(createdEnemyModel.x, createdEnemyModel.y);
  const enemySprite = this._world.add.sprite(0, 0, createdEnemyModel.name, 6);
  let enemyName = this._world.add.text(-15, -20, createdEnemyModel.name, { font: '8px roboto', align: 'center' });
  enemyContainer.setSize(16, 16);
  enemyContainer.add(enemySprite);
  enemyContainer.add(enemyName);
```

Rys. 4.57 Fragment kodu wyświetlający potwory

Potwór podobnie jak postać gracza jest kontenerem. Oznacza to, że na jego grafikę składać się może wiele obrazów. Jest to potwór oraz nazwa nad jego głową.

Ruch potworów odbywa się losowo co 3 sekundy. Pakiet Spring Framework dostarcza gotową funkcjonalność wykonywania metody co stały okres czasu. W tym celu należy umieścić adnotację @Scheduled z argumentem fixedDelay przed definicją metody, którą chcemy powtarzać (Rys. 4.58). Argument przyjmuje czas mierzony w milisekundach.

```
@Scheduled(fixedDelay = 3000, initialDelay = 7500)
```

Rys. 4.58 Adnotacja Scheduled

W celu aktywacji tej funkcjonalności, wymagane jest umieszczenie dodatkowej adnotacji @EnableScheduling nad nazwą klasy odpowiadającej za konfigurację lub nad kontrolerem (Rys. 4.59).

```
@Controller
@EnableScheduling
public class CreatureService {
```

Rys. 4.59 Adnotacja EnableScheduling

Metoda jest wykonywana dla każdego potwora w grze. Rozpoczyna się od losowania kierunku ruchu. Następnie pozycja każdego potwora aktualizowana jest o 20 jednostek i wysyłana na endpoint */topic/monsters/move/update/*, nasłuchiwany przez wszystkich zalogowanych graczy (Rys.4.60).

```
private configureEnemyMovement() {
  this._client.subscribe( options: '/topic/monsters/move/update', res => {
    const response = JSON.parse(res.body);
    for (const updatedEnemy of response) {
      this._spawns.getChildren().forEach(enemy => {
        // @ts-ignore
        if (enemy.id === updatedEnemy.id) {
          const monsterToMove = enemy;
          // @ts-ignore
          const currentX = monsterToMove.x;
          // @ts-ignore
          const currentY = monsterToMove.y;

          const deltaX = Math.abs( x: updatedEnemy.x - currentX);
          const deltaY = Math.abs( x: updatedEnemy.y - currentY);

          if (updatedEnemy.x < currentX) {
            // @ts-ignore
            monsterToMove.body.setVelocityX(-this.V);
          } else if (updatedEnemy.x > currentX) {
            // @ts-ignore
            monsterToMove.body.setVelocityX(this.V);
          }
          if (updatedEnemy.y < currentY) {
            // @ts-ignore
            monsterToMove.body.setVelocityY(-this.V);
          } else if (updatedEnemy.y > currentY) {
            // @ts-ignore
            monsterToMove.body.setVelocityY(this.V);
          }
          setTimeout( handler: () => {
            // @ts-ignore
            enemy.body.setVelocityX(0);
          }, timeout: (deltaX / this.V) * 1000);
          setTimeout( handler: () => {
            // @ts-ignore
            enemy.body.setVelocityY(0);
          }, timeout: (deltaY / this.V) * 1000);
        }
      });
```

Rys. 4.60 Fragment kodu odpowiedzialny za ruch potworów

Każdy potwór na mapie po stronie klienta zostaje poruszony w kierunku, w którym został zaktualizowany na serwerze. W tym celu należy wyliczyć moduł z różnicy zmienionej pozycji względem pierwotnej (patrz (2) (3)).

$$deltaX = abs(updatedEnemyX - currentX) \quad (2)$$
$$deltaY = abs(updatedEnemyY - currentY) \quad (3)$$

gdzie:
- updatedEnemyX oznacza nową pozycję X potwora
- updatedEnemyY oznacza nową pozycję Y potwora
- currentX oznacza obecną pozycję X potwora
- currentY oznacza obecną pozycję Y potwora

Gdy jest już wyliczona różnica, rozpoczyna się animacja potwora. Nadana zostaje mu stała prędkość w odpowiednim kierunku. Czas, przez który postać kreatury pozostaje w ruchu jest wyliczany ze wzoru (patrz (4)).

$$t = \frac{delta}{V} * 1000 \tag{4}$$

gdzie:
- delta oznacza różnicę pomiędzy nową a starą pozycją potwora
- V stała prędkość równa 40

Czas jest mnożony przez 1000, ponieważ oczekiwany jest ruch w sekundach, natomiast bazowa jednostka dla funkcji setTimeout to milisekundy.
Po czasie t, prędkość potwora zostaje ponownie zmieniana na wartość zerową, wskutek czego postać zatrzymuje się.

Ruch potwora jest wizualizowany w postaci animacji jego grafiki. (Rys. 4.61).

Rys. 4.61 Potwór wyświetlony na planszy

4.6. Zapisywanie aktualnego stanu gry [K.D., J.D.]

Zapisywanie stanu gry do bazy danych odbywa się tylko przy zakończeniu sesji użytkownika. Sesja kończy się, gdy gracz się rozłączy lub aplikacja zostanie wyłączona. Do obsługi tego zdarzenia zdefiniowana jest metoda *onApplicationEvent* (Rys. 4.62) dostarczona z interfejsu *org.springframework.context.ApplicationListener*.

```java
@Override
@Transactional
public void onApplicationEvent(SessionDisconnectEvent event) {
    log.debug("disconnected");
    ActivityDTO activityDTO = new ActivityDTO();
    activityDTO.setSessionId(event.getSessionId());
    for(PlayerDTO el : gameService.getPlayers()){
        if(event.getSessionId().equals(el.getSessionId())){
            gameService.getPlayers().remove(el);
            Player currentPlayer = playerRepository.findOneByCharacterName(el.getCharacterName());
            currentPlayer.getLastLoginPosition().setPosX(el.getX());
            currentPlayer.getLastLoginPosition().setPosY(el.getY());
            currentPlayer.getLastLoginPosition().setMapId(el.getLandName());
            currentPlayer.setLevel(el.getLevel());
            currentPlayer.getStatus().setExperience(el.getExp());
            playerRepository.save(currentPlayer);
            break;
        }
    }
    messagingTemplate.convertAndSend( destination: "/topic/position/disconnect", activityDTO);
}
```

Rys. 4.62 Metoda obsługująca rozłączenie użytkownika

Spring Framework wykrywa dodaną implementację. Jest to wystarczające by zapewnić wykonywanie działań przy zakończeniu sesji. Zapisywane dane to:

- Pozycja postaci – Współrzędne x i y wraz z nazwą mapy, której dotyczą
- Poziom postaci – Aktualny poziom postaci
- Doświadczenie postaci – Aktualna liczba punktów doświadczenia postaci

Po zapisaniu danych, użytkownik jest usunięty z kolekcji aktualnych postaci w grze.

4.7. Komunikacja między graczami [K.D., J.D.]

Do komunikacji między graczami wykorzystywany jest mechanizm czatu. Funkcjonalność ta zaimplementowana jest z wykorzystaniem technologii websocket. Powodem jest konieczność komunikacji w czasie rzeczywistym.

Wszystkie wiadomości są zapisywane w pamięci tymczasowej. Trzymanie ich w pamięci trwałej w szybkim tempie wyczerpałaby dostępne miejsce. Za pobieranie odpowiada metoda *getMessages* (Rys. 4.63). Po otrzymaniu komunikatu na endpoincie */topic/chat/message/get*, serwer wysyła listę wiadomości do wszystkich użytkowników nasłuchujących na */topic/player/chat/messages.*

```java
@MessageMapping("/topic/chat/message/get")
@SendTo("/topic/players/chat/messages")
public List<ChatMessage> getMessages() { return this.chatMessageList; }
```

Rys. 4.63 Metoda obsługująca pobranie wszystkich wiadomości po stronie serwera

W analogiczny sposób stworzona jest metoda *addMessage* (Rys. 4.64), odpowiadająca za dodawanie nowych wiadomości. Otrzymuje komunikaty na endpoincie */topic/chat/message/add.* Przyjęta wiadomość jest dodawana do listy, która następnie jest wysyłana wszystkim użytkownikom nasłuchującym na */topic/players/chat/message.*

```
@MessageMapping("/topic/chat/message/add")
@SendTo("/topic/players/chat/messages")
public List<ChatMessage> addMessage(@Payload ChatMessage message, Principal principal){
    message.setUserName(principal.getName());
    message.setDate(Instant.now());
    this.chatMessageList.add(message);
    return this.chatMessageList;
}
```

Rys. 4.64 Metoda obsługująca dodanie wiadomości po stronie serwera

Po stronie klienta, użytkownicy nasłuchują na endpoincie */topic/players/chat/message* (Rys. 4.65). Lista jest aktualizowana zarówno przy każdej nowej wiadomości, jak i pierwszym załadowaniu. Odpowiedź z serwera jest zamieniana na obiekt JSON, z którego pobierane są wiadomości dodawane do stanu w redux.

```
this.stompFactory.getStompClientObservable().subscribe( next: val => {
  val.client.subscribe( options: '/topic/players/chat/messages', v => {
    this.props.updateMessages( messageData: {
      messageData: JSON.parse(v.body)
    });
    const objDiv = document.getElementById( elementId: 'messageList');
    objDiv.scrollTop = objDiv.scrollHeight;
  });
```

Rys. 4.65 Nasłuchiwanie na endpoincie czatu

Gdy użytkownik wpisze wiadomość i wciśnie przycisk „Send”, zostaje ona wysłana do serwera (Rys. 4.66).

```
this.stompFactory.getStompClient().send(
  '/topic/chat/message/add',
  JSON.stringify( value: {
    message: msgValue
  }),
  {}
);
```

Rys. 4.66 Metoda obsługująca dodanie wiadomości po stronie klienta

W celu pobrania wszystkich wiadomości z serwera, wysyłane jest żądanie na endpoint */topic/chat/message/get* (Rys. 4.67).

```
this.stompFactory.getConnection().then(() => {
  this.stompFactory.getStompClient().send('/topic/chat/message/get', {}, {});
});
```

Rys. 4.67 Metoda obsługująca pobranie wszystkich wiadomości po stronie klienta

Do wizualizacji czatu został stworzony komponent *Chat* (Rys. 4.68). W jego skład wchodzi lista wiadomości oraz forma do ich wysyłania.

```
export const Chat = (props: IChatProps) => {
  const { messages } = props;
  return (
    <div className="chat">
      <p className="welcome-msg">Witamy w DEILARA!</p>
      <MessageList />
      <SendMessageForm />
    </div>
  );
};
```

Rys. 4.68 Komponent czatu

W rezultacie otrzymany jest widok przyjazny użytkownikowi (Rys. 4.69).

Rys. 4.69 Widok czatu

4.8. System walk [K.D., J.D.]

W grze zaimplementowany jest system walki. Kiedy potwór znajduje się w zasięgu gracza, ten ma możliwość zlikwidowania go. Dzieje się to, gdy przeciwnik koliduje z mieczem. Gdy użytkownik wciśnie klawisz spacji, potwór zostaje zniszczony. Klient wysyła informację o zdarzeniu na endpoint */topic/monsters/kill* (Rys. 4.70).

```
this._client.send('/topic/monsters/kill', enemy.id);
```

Rys. 4.70 Wysłanie informacji o zabitym potworze

Serwer otrzymuje identyfikator potwora oraz użytkownika biorącego udział w wydarzeniu (Rys. 4.71).

```java
@MessageMapping("/topic/monsters/kill")
@SendTo("/topic/monsters/kill/update")
public List<MonsterDTO> killMonster(@Payload String uuid, SimpMessageHeaderAccessor headers, Principal principal) throws NotFoundException {
    PlayerDTO player = gameService.getPlayers().stream().filter(el -> Objects.equals(headers.getSessionId(), el.getSessionId()))
        .findAny().orElseThrow(() -> new NotFoundException("Player not found in database"));
    MonsterDTO monsterDTO = gameService.getMonsters().stream().filter(m -> m.getId().toString().equals(uuid))
        .findAny().orElseThrow(() -> new NotFoundException("Monster not found in database"));
    countPlayerExperience(player, monsterDTO.getExpGain());
    this.gameService.getMonsters().remove(monsterDTO);
    messagingTemplate.convertAndSendToUser(principal.getName(), destination: "/queue/player/currentLevel", player.getLevel());
    return this.gameService.getMonsters();
}
```

Rys. 4.71 Metoda odpowiedzialna za śmierć potwora

Gracz otrzymuje punkty doświadczenia w ilości odpowiadającej danemu przeciwnikowi. Szczegóły opisane są w podrozdziale 4.9. Zabity potwór zostaje usunięty z kolekcji aktywnych. Następnie serwer informuje użytkownika inicjującego operację o obecnym stanie jego poziomu. Informacja ta wysłana jest na endpoint */queue/player/currentLevel*. Wszyscy użytkownicy są poinformowani o śmierci potwora w celu usunięcia go z mapy. Gracze odbierają tę informację, nasłuchując endpoint */topic/monsters/kill/update*.

Potwory są odradzane co określoną, stałą wartość czasu. W tym celu metoda *respawnMonsters* oznaczona jest adnotacją @Scheduled (Rys. 4.72).

```java
@Scheduled(fixedDelay = 10000)
public void respawnMonsters() {
    List<MonsterDTO> respawnedEnemies = gameService.getMonsterSpawns().stream()
        .filter(m -> !gameService.getMonsters().contains(m))
        .collect(Collectors.toList());
    gameService.getMonsters().addAll(respawnedEnemies);
    messagingTemplate.convertAndSend( destination: "/topic/monsters/create/update", respawnedEnemies);
}
```

Rys. 4.72 Metoda odpowiedzialna za odradzanie potworów

Kod odpowiedzialny za odradzanie potworów, wyszukuje takie, które nie znajdują się w kolekcji aktywnych. Wynik jest do niej dodawany i wysyłany na endpoint */topic/monsters/craete/update*, który jest również wykorzystywany przy tworzeniu potworów (Rys. 4.56).

4.9. Rozwój postaci [K.D., J.D.]

Za rozwój postaci rozumie się zdobywanie punktów doświadczenia. Te z kolei zamieniane są na poziomy w momencie przekroczenia ustalonego progu. Do aktualizacji dochodzi po pokonaniu potwora (Rys. 4.73).

```java
private void countPlayerExperience(PlayerDTO player, int expGain) {
    // needed exp for next level
    player.setExp(player.getExp() + expGain);
    int expNeeded = (int) (Math.pow(10, player.getLevel() - 1D) * GameConfiguration.INTIAL_LEVEL_EXP_REQ);
    while (expNeeded <= player.getExp()) {
        player.setLevel(player.getLevel() + 1);
        expNeeded *= 10;
    }
}
```

Rys. 4.73 Funkcja wyliczająca poziom gracza.

Próg doświadczenia potrzebny do zdobycia kolejnego poziomu jest wyliczany ze wzoru (5).

$$x = 10^{c-1} * p \tag{5}$$

gdzie:
x – Próg doświadczenia potrzebny do zdobycia kolejnego poziomu.
c – Aktualny poziom gracza
p – Doświadczenie potrzebne do zdobycia pierwszego poziomu równe 50.

Zakończenie

Celem pracy był projekt i implementacja gry RPG, która jest aplikacją webową. Ze względu na specyfikę dziedziny, pobocznymi celami były:

- Poznanie technologii webowych
- Wzbogacenie wiedzy dotyczącej tworzenia gier komputerowych

Cel pracy został osiągnięty, gdyż w rezultacie otrzymano aplikację spełniającą jego założenia. Przy implementacji, wykorzystano najnowsze technologie webowe. Poznano przy tym rozwiązania problemów związanych z komunikacją w czasie rzeczywistym pomiędzy klientem a serwerem. Implementacja gier komputerowych odbywa się na innej zasadzie niż aplikacji biznesowych. Wiedza ta, tak jak w poprzednim przypadku, również została nabyta.

W ramach pracy część funkcjonalności nie została zaimplementowana ze względu na krótki przedział czasowy. Po mimo tego aplikacja zachowuje opisane w celu założenia. Gracze rywalizują ze sobą na mapie, aby zdobyć najwyższy poziom. Osiągają to poprzez pokonywanie potworów. Mogą również komunikować się ze sobą za pomocą okna czatu. Pominięte zostały funkcjonalności o najmniejszym priorytecie. Są to wykorzystanie przedmiotów, umiejętności oraz mechanizmy związane z przeliczaniem punktów życia i many.

Dobrze wykonany przegląd dziedzinowy, pozwolił na znaczne uproszczenie pracy. Znalezienie odpowiednich technologii takich jak Phaser, czy też JHipster, w dużym stopniu skróciło czas implementacji. Znajomość gotowych rozwiązań wykorzystywanych w dziedzinie tworzenia gier komputerowych również się do tego przyczyniło.

Komunikacja w czasie rzeczywistym powinna odbywać się za pomocą protokołu websocket, ze względu na potrzebę ciągłej wymiany informacji. Po mimo tego, nie zawsze powinien być wykorzystywany, gdyż zwiększa poziom złożoności architektury. Ciągła łączność i tak nie rozwiązuje wszystkich problemów. Nawet przy wykorzystaniu najlepszych rozwiązań należy założyć, że komunikacja może być w jakimś stopniu opóźniona, a nawet zakończona.

Implementacja jak i projekt gry komputerowej odbywa się na innej zasadzie niż większość aplikacji webowych. Zamiast wielu różnych źródeł przepływu danych istnieje jedna pętla gry, w której zachodzą wszystkie zmiany. W związku z tym podejście do projektu jak i implementacji podczas pracy było inne.

Dodanie testów wspomogło szybkie rozpoznawanie potencjalnych jak i istniejących błędów. Skróciło to czas rozwiązywania problemów oraz uprościło zarządzanie grą.

W przypadku braku niektórych funkcjonalności, dopasowanie implementacji interfejsu do opisanych założeń było skomplikowane. Podjęto więc decyzję o jego modyfikacji.

Reasumując, implementacja gry RPG jako aplikacji webowej nie jest łatwym zadaniem i wymaga dogłębnej analizy na wielu płaszczyznach. Potrzebny jest pełny przegląd dziedzinowy oraz wiedza techniczna.

Gra wraz z instrukcją instalacji jest dostępna do pobrania pod adresem https://drozdowskidwojak.itch.io/deilara.

Bibliografia

[1] Hootsuite, WeAreSocial, „Datareportal,” January 2019. [Online]. Available: https://datareportal.com/reports/digital-2019-global-digital-overview.

[2] J. Schell, The Art of Game Design, Taylor & Francis Ltd, 2019.

[3] R. C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship, Prentice Hall, 2009.

[4] B. Eckel, Thinking in Java, Prentice Hall, 2006.

[5] C. Walls, Spring in Action, Manning, 2014.

[6] D. Kurata, Doing Web Development: Client-Side Techniques, Apress, 2014.

[7] . A. Accomazzo , N. Murray i A. Lerner, Fullstack React: The Complete Guide to ReactJS and Friends, Fullstack.io, 2017.

[8] E. Feronato, HTML5 Cross Platform Game Development Using Phaser 3, Emanuele Feronato, 2019.

[9] C. Bauer i G. King, Hibernate w akcji, Helion, 2005.

[10] G. Smith, PostgreSQL 9.0 High Performance, Packt Publishing, 2010.

[11] M. Johns, Getting Started with Hazelcast, Packt Publishing, 2013.

[12] A. Lombardi, WebSocket: Lightweight Client-Server Communications, O'Reilly Media, 2015.

[13] R. Osherove, The Art of Unit Testing, 2009.

[14] P. P. Papapetrou i G. A. Campbell, SonarQube in Action, 2013.

[15] I. Miell i A. H. Sayers, Docker in Practice, Second Edition, 2019.

[16] L. Gelman i B. Dinkevich, The Complete Redux Book, Leanpub, 2017.

[17] M. Raible, The JHipster Mini-Book, InfoQ, 2016.

[18] L. Beighleya, Head First SQL: Your Brain on SQL, O'Reilly Media, 2007.

[19] V. Mihalcea, High-Performance Java Persistance, VLAD MIHALCEA, 2016.

[20] D. Salas, X. Liang i Y. Liang, „Integration using a message broker,” Researchgate, 2012.

Spis rysunków

Spis tabel