WYŻSZA SZKOŁA TECHNOLOGII INFORMATYCZNYCH W KATOWICACH
WYDZIAŁ INFORMATYKI
KIERUNEK: INFROMATYKA

Leon Raczyński
Nr albumu 8233
Studia stacjonarne

PROJEKT I IMPLEMENTACJA GRY PRZEGLĄDARKOWEJ Z WYKORZYSTANIEM TECHNOLOGII WEBOWYCH

Praca dyplomowa napisana
pod kierunkiem
mgr inż. Łukasza Iwanickiego
w roku akademickim 2023/2024

Katowice 2024

Spis treści

WSTĘP

Celem niniejszej pracy jest zaprojektowanie i implementacja gry przeglądarkowej w technologiach webowych.

W dobie rozwijających się różnego rodzaju silników do gier komputerowych, konsolowych i mobilnych, gry przeglądarkowe stają się coraz mniej popularne. Motywacją do zrealizowania tematu pracy jest stworzenie gry, która nie wymaga od użytkownika instalacji jakiegokolwiek dodatkowego oprogramowania, a dostępna jest bezpośrednio z poziomu przeglądarki internetowej. Dodatkowo oparta jest tylko i wyłącznie o technologie takie jak HTML5[1], CSS3[2], JavaScript[3], Node.js[4] i SQL[5]. Aspekt ten daje możliwość twórcy swobodnej realizacji zamysłów projektowych bez konieczności trzymania się ściśle określonych i narzuconych szablonów oraz opcji przez silniki do tworzenia gier.

Ponadto, rozwijanie gry przeglądarkowej pozwala na praktyczne zastosowanie wielu aspektów inżynierii oprogramowania, takich jak projektowanie interfejsu użytkownika, zarządzanie bazą danych, czy też implementacji logiki gry.

W ramach pracy zostanie przedstawiony proces tworzenia gry, od projektowania interfejsu użytkownika, przez zarządzanie bazą danych, aż po implementację całości.

[1] https://pl.wikipedia.org/wiki/HTML5
[2] https://pl.wikipedia.org/wiki/Kaskadowe_arkusze_styl%C3%B3w
[3] https://pl.wikipedia.org/wiki/JavaScript
[4] https://nodejs.org/en
[5] https://pl.wikipedia.org/wiki/SQL

ROŹDZIAŁ 1
ANALIZA WYMAGAŃ

1.1. Określenie celów projektu

W celu uzyskania efektywniejszej realizacji zamysłu projektowego zostały nakreślone następujące cele projektowe.

1.1.1. Zaprojektowanie i implementacja

Głównym celem projektu jest stworzenie gry przeglądarkowej, która będzie atrakcyjna dla potencjalnego gracza.

1.1.2. Wykorzystanie technologii webowych

Projekt wykorzystuje takie technologie jak HTML5, CSS3, JavaScript[1] oraz Node.js.

1.1.3. Tworzenie grafiki i dźwięku

Projekt zawiera atrakcyjne elementy graficzne typu *pixel art* w perspektywie 2D oraz dźwiękowe, które przyciągną użytkownika i zwiększą jego zainteresowanie.

1.1.4. Dostępność i użyteczność

Finalnie, gra powinna być łatwa w obsłudze i dostępna dla szerokiego grona użytkowników, niezależnie od ich doświadczenia z grami przeglądarkowymi.

1.2. Wymagania funkcjonalne i niefunkcjonalne

W projekcie powinny być zrealizowane następujące wymagania funkcjonalne oraz niefunkcjonalne.

1.2.1. Wymagania funkcjonalne

***Tabela 1**. Wymaganie funkcjonalne: Rejestracja*

Nr. wymagania	1
Typ wymagania	Funkcjonalne
Przypadek użycia	Rejestracja
Opis	Użytkownik chcąc zacząć nową grę zakłada konto wpisując imię dla swojego bohatera oraz hasło.
Uzasadnienie	Użytkownik powinien mieć możliwość założenia nowego konta w celu rozpoczęcia rozgrywki od nowa.

Źródło: Opracowanie własne

***Tabela 2.** Wymaganie funkcjonalne: Logowanie*

Nr. wymagania	2
Typ wymagania	Funkcjonalne
Przypadek użycia	Logowanie
Opis	Użytkownik chcąc zalogować się na swoje wcześniej utworzone konto wpisuje imię swojego bohatera i hasło.
Uzasadnienie	Użytkownik powinien mieć możliwość zalogowania na swoje konto w celu kontynuacji gry od ostatniego etapu w którym zostały zapisane jego postępy.

Źródło: Opracowanie własne

Tabela 3. Wymaganie funkcjonalne: Sterowanie

Nr. wymagania	3
Typ wymagania	Funkcjonalne
Przypadek użycia	Sterowanie
Opis	Gracz ma możliwość sterowania swoim bohaterem (tj. poruszanie się oraz atak).
Uzasadnienie	W celu poruszania się po mapie i interakcji z przeciwnikiem, gracz powinien mieć możliwość sterowania postacią za pomocą odpowiednich klawiszy.

Źródło: Opracowanie własne

Tabela 4. Wymaganie funkcjonalne: Zarządzanie ekwipunkiem

Nr. wymagania	4
Typ wymagania	Funkcjonalne
Przypadek użycia	Zarządzanie ekwipunkiem
Opis	Gracz ma możliwość zarządzania przedmiotami w znajdującymi się w ekwipunku swojego bohatera. Przekładania ich z miejsca na miejsce, ubierania oraz zdejmowania z ciała postaci.
Uzasadnienie	W celu polepszenia statystyk bohatera, gracz powinien mieć możliwość zakładania przedmiotów na swoją postać. Otwarcie ekwipunku powinno odbywać się za pomocą wciśnięcia odpowiedniego klawisza.

Źródło: Opracowanie własne

Tabela 5. *Wymaganie funkcjonalne: Interakcja z bohaterem niezależnym*

Nr. Wymagania	5
Typ wymagania	Funkcjonalne
Przypadek użycia	Interakcja z bohaterem niezależnym
Opis	Gracz ma możliwość interakcji z postaciami niezależnymi w celu prowadzenia dialogów.
Uzasadnienie	Gracz powinien mieć możliwość prowadzenia dialogu z postaciami niezależnymi co skutkuje przywróceniem życia głównego bohatera, czy też zakończeniem etapu zadania w zależności od konkretnego *NPC*. Interakcja powinna odbywać się poprzez wciśnięcie odpowiedniego klawisza.

Źródło: Opracowanie własne

Tabela 6. *Wymaganie funkcjonalne: Pominięcie dialogu*

Nr. Wymagania	6
Typ wymagania	Funkcjonalne
Przypadek użycia	Pominięcie dialogu
Opis	Gracz ma możliwość pominięcia aktualnie wyświetlanego dialogu prowadzonego z bohaterem niezależnym.
Uzasadnienie	W momencie gdy gracz przeczyta aktualny dialog lub wyrazi chęć jego pominięcia, powinien mieć możliwość wykonania tej czynności. Pominięcie dialogu powinno odbywać się poprzez wciśnięcie odpowiedniego klawisza.

Źródło: Opracowanie własne

Tabela 7. Wymaganie funkcjonalne: Otwarcie okna menu

Nr. wymagania	7
Typ wymagania	Funkcjonalne
Przypadek użycia	Otwarcie okna menu
Opis	Gracz ma możliwość otwarcia okna menu z opcjami „Wczytaj ostatni zapis” i „Wyloguj się” w celu zrealizowania którejś z tych pozycji.
Uzasadnienie	W celu uzyskania dostępu do opcji „Wczytaj ostatni zapis” i „Wyloguj się” gracz powinien mieć możliwość otwarcia okna menu.

Źródło: Opracowanie własne

Tabela 8. Wymaganie funkcjonalne: Wczytanie ostatniego zapisu

Nr. wymagania	8
Typ wymagania	Funkcjonalne
Przypadek użycia	Wczytanie ostatniego zapisu
Opis	Gracz ma możliwość wczytania stanu gry z ostatniego punktu kontrolnego w którym gra została zapisana.
Uzasadnienie	Gracz powinien mieć możliwość cofnięcia się do stanu gry z ostatniego zapisu. Wybranie tej opcji powinno odbywać się poprzez wybranie LPM odpowiedniej pozycji w menu.

Źródło: Opracowanie własne

Tabela 9. Wymaganie funkcjonalne: Wylogowywanie

Nr. wymagania	9
Typ wymagania	Funkcjonalne
Przypadek użycia	Wylogowywanie
Opis	Gracz ma możliwość wylogowania się z poziomu gry. Czynność ta jest zwieńczona powrotem do ekranu logowania.
Uzasadnienie	Gracz powinien mieć możliwość wylogowanie się w celu zmiany konta.

Źródło: Opracowanie własne

1.2.2. Wymagania niefunkcjonalne

Tabela 10. *Wymaganie niefunkcjonalne: Wspierany system operacyjny*

Nr. Wymagania	10
Typ wymagania	Niefunkcjonalne
Przypadek użycia	Wspierany system operacyjny
Opis	Serwer na którym działa gra uruchamia się na urządzeniach z systemem Windows[6].
Uzasadnienie	System Windows jest najpopularniejszym systemem operacyjnym na komputery personalne. Serwer powinien pracować z tym systemem.

Źródło: Opracowanie własne

Tabela 11. *Wymaganie niefunkcjonalne: Wspierane przeglądarki internetowe*

Nr. Wymagania	11
Typ wymagania	Niefunkcjonalne
Przypadek użycia	Wspierane przeglądarki internetowe
Opis	Gra uruchamia się w popularnych przeglądarkach internetowych tj. Microsoft Edge[7], Google Chrome[8] i Opera[9].
Uzasadnienie	Gra powinna się uruchamiać w najbardziej popularnych przeglądarkach internetowych, aby zwiększyć liczbę odbiorców.

Źródło: Opracowanie własne

[6] https://www.microsoft.com/pl-pl/windows?r=1
[7] https://www.microsoft.com/pl-pl/edge?form=MA13FJ&ch=1
[8] https://www.google.com/intl/pl_pl/chrome/
[9] https://www.opera.com/pl

Tabela 12. *Wymaganie niefunkcjonalne: Wygodny interfejs*

Nr. wymagania	12
Typ wymagania	Niefunkcjonalne
Przypadek użycia	Wygodny interfejs
Opis	Interfejs gry jest intuicyjny, zrozumiały i łatwy w użyciu dla nowych graczy.
Uzasadnienie	Interfejs gry powinien być zbudowany w taki sposób aby nowi gracze oraz osoby które nie są biegłe w grach mogły z łatwością się po nim poruszać.

Źródło: Opracowanie własne

ROŹDZIAŁ 2
PROJEKTOWANIE GRY

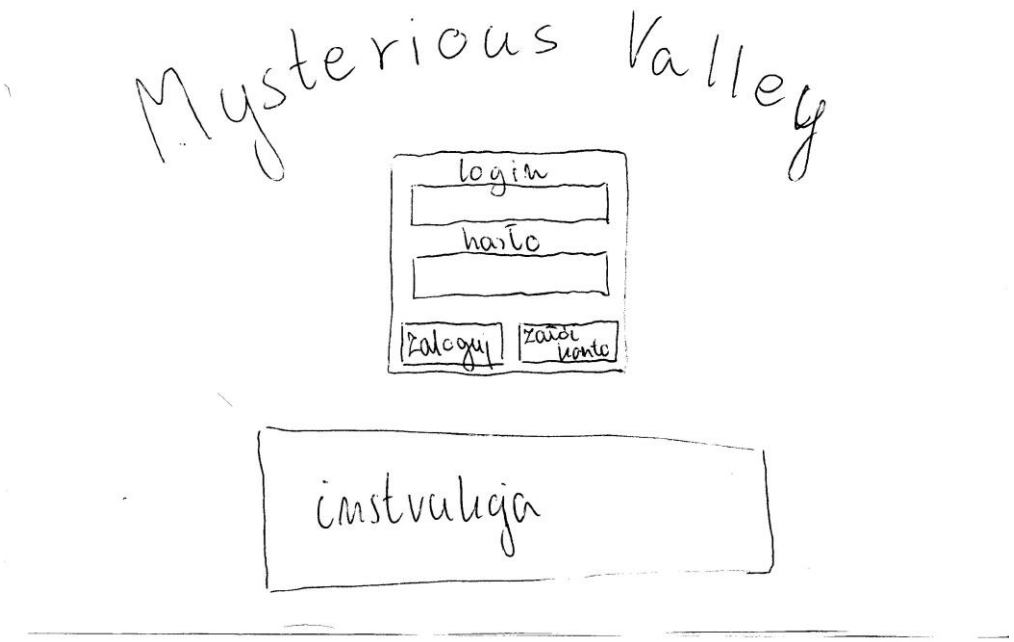
2.1. Koncepcja gry

W toku pracy nad grą pojawił pomysł na jej tytuł „Mysteroius Valley”, co w wolnym tłumaczeniu znaczy „Tajemnicza Dolina”. Jest to gra przeglądarkowa o charakterze *RPG*. Gracz przenosi się do świata fantasy w którym musi sprostać wyzwaniu jakim jest pomoc mieszkańcom miasta. W osiągnięciu celu przygody, przeszkadzają mu magiczne stworzenia, które po dostrzeżeniu głównego bohatera natychmiast go atakują. W razie potrzeby na pomoc bohaterowi przyjdą mieszkańcy, którzy pomogą mu wyleczyć poniesione rany, czy też użyczą mu dodatkowego ekwipunku.

2.2. Projekt interfejsu użytkownika

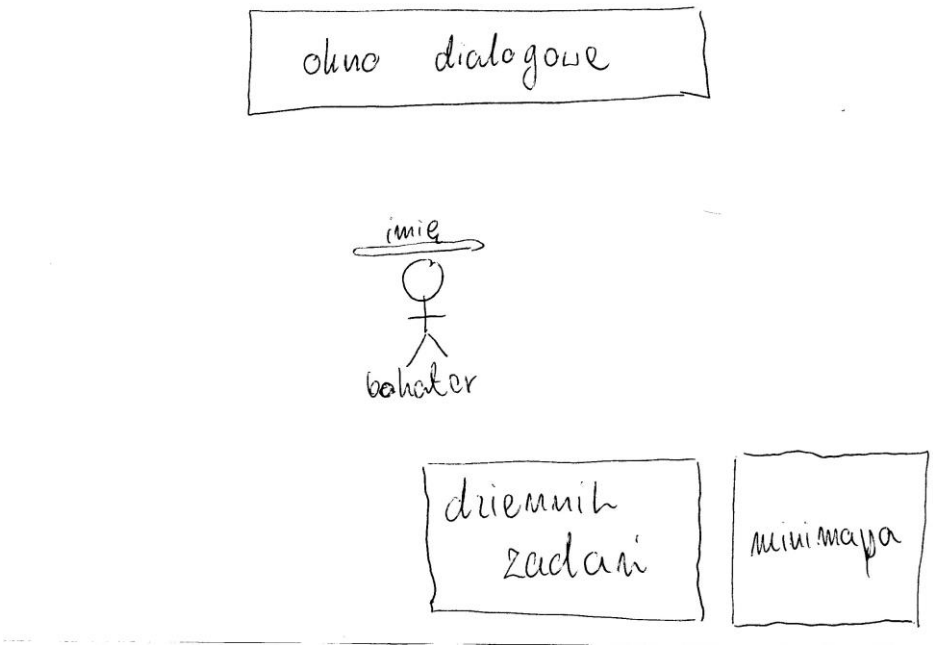
Proces tworzenia gry został poprzedzony stworzeniem projektu wizualnego poszczególnych elementów interfejsu użytkownika.

2.2.1. Ekran logowania



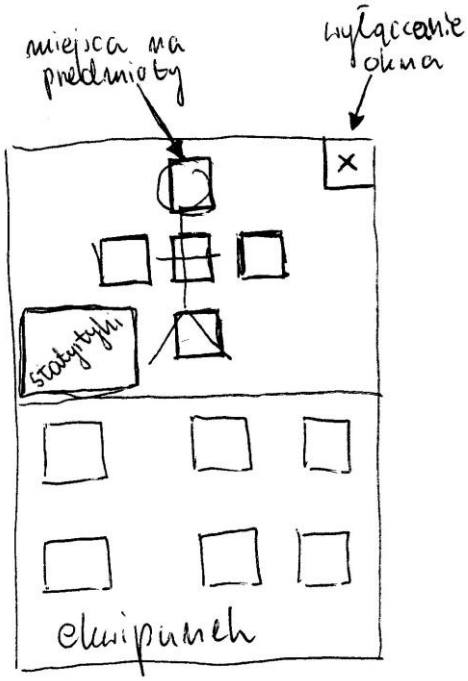
Rysunek 1. *Projekt: Ekran logowania*
Źródło: Opracowanie własne

2.2.2. Ekran główny gry



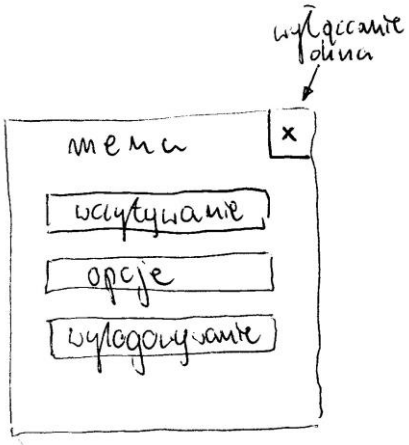
Rysunek 2. *Projekt: Ekran główny gry*
Źródło: Opracowanie własne

2.2.3. Ekwipunek



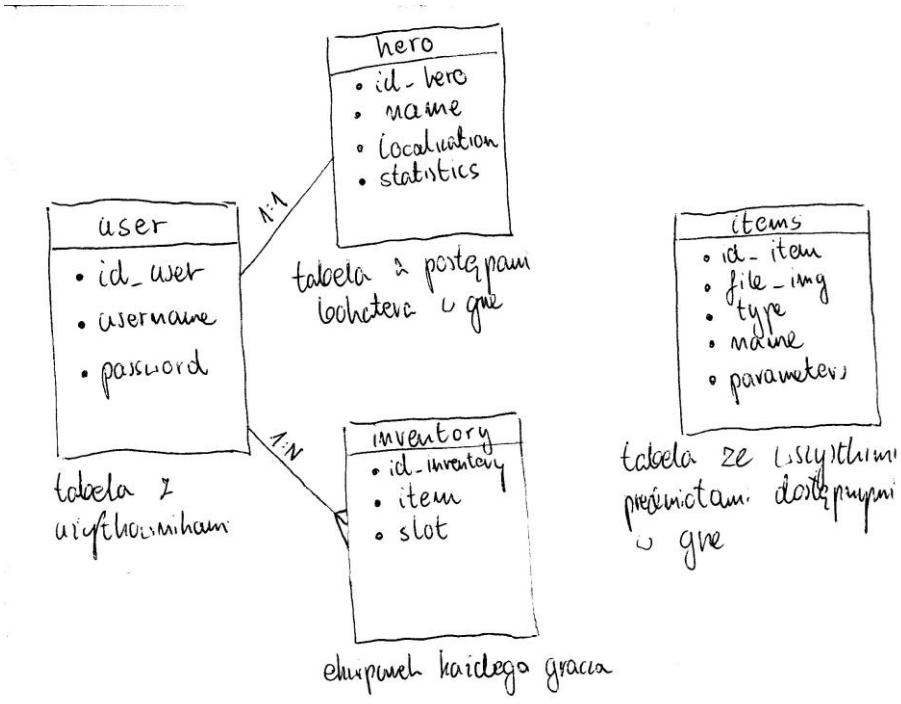
Rysunek 3. *Projekt: Ekwipunek*
Źródło: Opracowanie własne

2.2.4. Menu gry



Rysunek 4. *Projekt: Menu gry*
Źródło: Opracowanie własne

2.3. Projekt tabel bazy danych wraz z relacjami



Rysunek 5. *Projekt: Baza danych*
Źródło: Opracowanie własne

ROŹDZIAŁ 3
WYBÓR NARZĘDZI PRACY

3.1. Środowisko programistyczne

Do zrealizowania projektu, jako środowisko programistyczne został wybrany program Visual Studio Code[10] firmy Microsoft. Program nie zajmuje dużo miejsca na dysku i nie pobiera wielu zasobów komputera podczas pracy. Ponadto użytkownik ma możliwość zainstalowania wielu dodatkowych rozszerzeń bezpośrednio z poziomu aplikacji.

3.2. Środowisko uruchomieniowe

Aby umożliwić funkcjonowanie bazy danych w projekcie został użyty Node.js – środowisko uruchomieniowe. Pozwoliło to również instalację modułu odpowiadającego za obsługę serwera.

3.3. Serwer

JavaScript sam w sobie nie obsługuje operacji na zewnętrznych bazach danych, dlatego w projekcie został użyty serwer Express[11] wraz z modułem Express Session[12] w celu zastosowania sesji użytkownika.

3.4. System zarządzania bazą danych

W celu przechowywania danych użytkownika oraz przedmiotów dostępnych w grze została zastosowana baza danych SQLite[13].

SQLite jest systemem zarządzania bazą danych który charakteryzuje się zajmowaniem małej ilości przestrzeni dyskowej i łatwością przenoszenia całej bazy danych. Taka baza zawarta jest tylko w jednym pliku.

[10] https://code.visualstudio.com/
[11] https://expressjs.com/
[12] https://expressjs.com/en/resources/middleware/session.html
[13] https://www.sqlite.org/

Baza danych SQLite może być tworzona i modyfikowana poprzez okno konsolowe systemu operacyjnego na którym pracuje użytkownik. Jednakże w celu wizualizacji całego procesu w projekcie zostały użyte programy SQLiteStudio[14] oraz DB Browser for SQLite[15].

3.5. Tworzenie i edycja elementów graficznych

Jako że projekt ma na celu stworzenie gry, to elementy graficzne są równie ważne co kod obsługujący całą mechanikę. W tym celu zostały wykorzystane programy Aseprite[16], Adobe Photoshop[17] i Tiled[18].

- **Aseprite** jest programem idealnie nadającym się do tworzenia i edycji grafiki typu *pixel art*. W tym właśnie stylu zaprojektowana jest gra.
- **Photoshop** jest programem graficznym przeznaczonym głównie do tworzenia i obróbki grafiki rastrowej. Posłużył on do stworzenia grafiki znajdującej się na ekranie logowania.
- **Tiled** posłuży do tworzenia mapy i definiowania lokalizacji do umieszczenia obiektów *NPC*, lokacji i obszarów w których tworzą się potwory.

3.6. Edycja elementów dźwiękowych

Dźwięki są bardzo ważnym aspektem we wszystkich rodzajach gier. Powodują one integralność czynności wykonywanych przez gracza z elementami w grze, poprzez emisje ich w odpowiednich momentach i czasie.

Jako że w celach projektowych nie zostało uwzględnione tworzenie dźwięków, zostały wykorzystane gotowe darmowe już ścieżki dźwiękowe pobrane z internetu [7][8][9]. W celu dopasowania ich do konkretnych zdarzeń w grze poddane zostały

[14] https://sqlitestudio.pl/
[15] https://sqlitebrowser.org/
[16] https://www.aseprite.org/
[17] https://www.adobe.com/pl/products/photoshop.html
[18] https://www.mapeditor.org/

edycji (tj. czas trwania, edycja głośności w odpowiednich miejscach) w programie Audacity[19].

[19] https://www.audacityteam.org/

ROŹDZIAŁ 4

STRUKTURA BAZY DANYCH

4.1. Przeznaczenie bazy danych

W procesie tworzenia gry bierze udział baza danych. Ma ona na celu przechowywanie danych w odpowiednich tabelach o danych logowania każdego gracza, aktualnych postępach w grze, aktualnego ekwipunku z przedmiotami oraz informacji o wszystkich przedmiotach dostępnych w grze wraz z ich parametrami i obrazami.

4.2. Tabele bazy danych

Cała baza danych składa się łącznie z 4 tabel. Tabele **user**, **hero** i **inventory** są połączone ze sobą odpowiednimi relacjami.

4.2.1. Tabela user

W tabeli tej przechowywane są dane o wszystkich zarejestrowany użytkownikach. Opis atrybutów:

- **id_user** (klucz główny) – przechowywane jest tu unikalne id konta każdego użytkownika.
- **username** – nazwa użytkownika i zarazem imię bohatera w grze.
- **password** – hasło służące do logowania się na własne konto.

Table: user		
Column	*Data type*	*Constraints*
id_user	*INTEGER*	•*PRIMARY KEY ASC AUTOINCREMENT* •*NOT NULL*
username	*TEXT*	•*NOT NULL*
password	*TEXT*	•*NOT NULL*

Rysunek 6. *Tabela: user*
Źródło: Opracowanie własne

4.2.2. Tabela hero

Tabela zawiera informacje bohaterze głównym i jego statystykach. Opis atrybutów:

- **id_hero** (klucz główny) – Unikalne id każdego bohatera głównego.
- **hero_name** – Imię bohatera nadane przez użytkownika podczas rejestracji.
- **position_x** – Ostatnia zapisana pozycja bohatera na osi X. Podczas tworzenia nowego konta bazowo jest ustawiona wartość -430.
- **position_y** – Ostatnia zapisana pozycja bohatera na osi Y. Podczas tworzenia nowego konta bazowo jest ustawiona wartość -6750.
- **current_stage** – Aktualnie wykonywane przez bohatera zadanie.
- **killed_monsters** – Zlikwidowane potwory.
- **founded_places** – Lista dotychczasowo odkrytych przez gracza miejsc na mapie.
- **id_user** (klucz obcy) – Komórka stworzona, aby połączyć relacjami ze sobą tabele.

Table: hero		
Column	Data type	Constraints
id_hero	INTEGER	•PRIMARY KEY AUTOINCREMENT
hero_name	TEXT	•NOT NULL
position_x	INTEGER	•DEFAULT (-430)
position_y	INTEGER	•DEFAULT (-6750)
current_stage	INTEGER	•DEFAULT (1)
killed_monsters	INTEGER	•DEFAULT (0)
founded_places	TEXT	
id_user	INTEGER	•NOT NULL
Global table constraints		
•FOREIGN KEY (id_user) REFERENCES user (id_user)		

Rysunek 7. *Tabela: hero*
Źródło: Opracowanie własne

4.2.3. Tabela inventory

W tej tabeli zawarte są informację o aktualnie noszonym przez bohatera ekwipunku. Podczas tworzenia nowego konta automatycznie tworzone są elementy tej tabeli w ilości odpowiadającej ilości dostępnego miejsca w ekwipunku. Podczas rozgrywki, gdy bohater otrzyma nowy przedmiot, element tabeli odpowiadający konkretnej komórce w ekwipunku jest aktualizowany. Opis atrybutów:

- **id_inv_item** (klucz główny) – Unikalne id każdego elementu tabeli.
- **item_name** – Nazwa przedmiotu umiejscowionego w tej komórce w ekwipunku.
- **frame** – Konkretna komórka w ekwipunku.
- **id_user** (klucz obcy) – Komórka stworzona, aby połączyć relacjami ze sobą tabele.

Table: inventory		
Column	*Data type*	*Constraints*
id_inv_item	*INTEGER*	*•NOT NULL* *•PRIMARY KEY ASC AUTOINCREMENT*
item_name	*TEXT*	
frame	*TEXT*	
id_user	*INTEGER*	*•NOT NULL*
Global table constraints		
•FOREIGN KEY (id_user) REFERENCES user (id_user)		

Rysunek 8. *Tabela: invenotry*
Źródło: Opracowanie własne

4.2.4. Tabela items

Tabela przechowuje informacje o wszystkich przedmiotach, które bohater może zdobyć podczas przygody. Zawarte są również tutaj parametry każdego przedmiotu wraz z jego plikiem graficznym zapisanym w postaci kodu Base64.

- **id_ite_item** (klucz główny) – Unikalne id każdego przedmiotu.

- **item_image** – Atrybut przechowuje obraz przedmiotu zakodowany w Base64[20].
- **item_type** – Typ przedmiotu. W grze prawie każdy przedmiot można założyć na bohatera. Typ przedmiotu pozwala uniknąć sytuacji w której gracz np. wkłada miecz w miejsce które przeznaczone jest do noszenia hełmu.
- **item_name** – Nazwa danego przedmiotu.
- **item_damage** – Parametr, który określa ile obrażeń zadaje dany przedmiot. Przedmioty, które nie są broniami nie maja w tym miejscu wartość 0.
- **item_defense** – Określa liczbę punktów obrony, którą bohater otrzymuje po założeniu przedmiotu z tym parametrem. Analogicznie do poprzedniej pozycji – przedmioty, które są broniami mają tą wartość ustawioną na 0.
- **item_hitpoints** – W zależności o wartości tego parametru, bohater po założeniu przedmiotu z tym atrybutem otrzymuje konkretną wartość dodatkowych punktów życia.
- **item_cost** – Informacja o wartości danego przedmiotu. Parametr utworzony w celu ewentualnej, dalszej ewaluacji projektu.

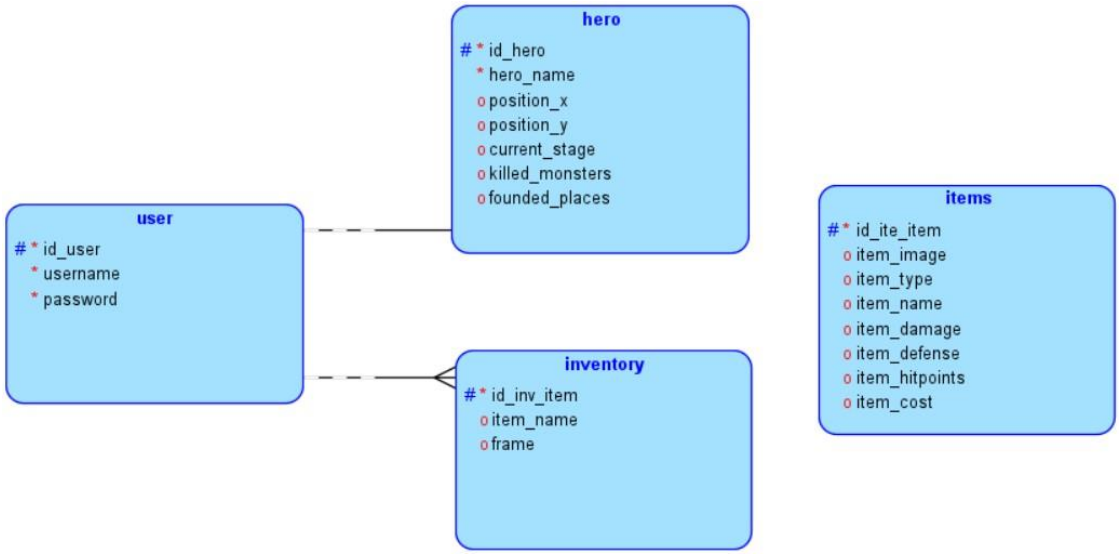
[20] https://pl.wikipedia.org/wiki/Base64

Table: items		
Column	*Data type*	*Constraints*
id_ite_item	*INTEGER*	*•NOT NULL*
item_image	*TEXT*	*•NOT NULL*
item_type	*TEXT*	*•NOT NULL*
item_name	*TEXT*	*•NOT NULL*
item_damage	*INTEGER*	*•NOT NULL*
item_defense	*INTEGER*	*•NOT NULL*
item_hitpoints	*INTEGER*	*•NOT NULL*
item_cost	*INTEGER*	*•NOT NULL*
Global table constraints		
•PRIMARY KEY (id_ite_item)		

Rysunek 9. *Tabela: items*
Źródło: Opracowanie własne

4.3. Model logiczny

Model logiczny bazy danych stworzony w programie SQL Developer Data Modeler[21] firmy Oracle.

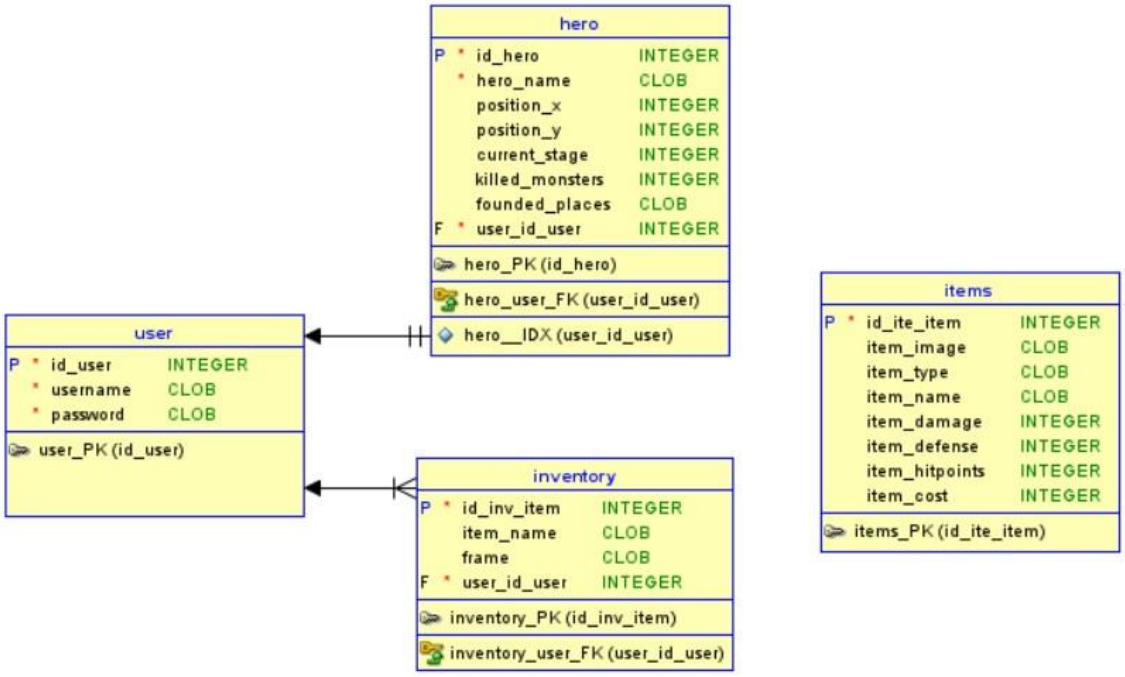


Rysunek 10. *Model logiczny bazy danych*
Źródło: Opracowanie własne

[21] https://www.oracle.com/database/sqldeveloper/technologies/sql-data-modeler/

4.4. Model relacyjny

Model relacyjny bazy danych stworzony w programie SQL Developer Data Modeler firmy Oracle.



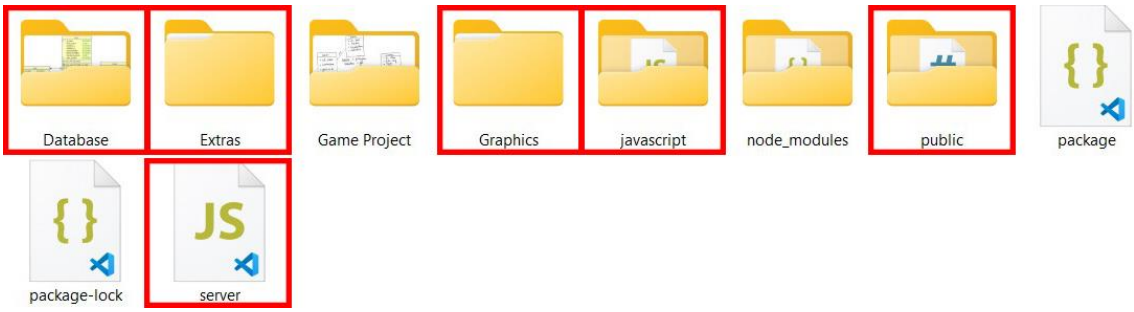
Rysunek 11. *Model relacyjny bazy danych*
Źródło: Opracowanie własne

ROŹDZIAŁ 5

IMPLEMENTACJA GRY

5.1. Struktura projektu

Projekt oparty jest o środowisko uruchomieniowe oraz serwer w którym są ściśle określone ścieżki dostępu do poszczególnych folderów w lokalizacji. Zachowanie więc odpowiedniej struktury i nazewnictwa folderów i plików zaznaczonych na poniższym zdjęciu jest niezbędne.



Rysunek 12. *Struktura folderu głównego projektu*
Źródło: Opracowanie własne

5.2. Implementacja serwera

Cały proces implementacji serwera składa się z kilku etapów. Każdy z nich jest opisany poniżej.

5.2.1. Instalacja modułów

Korzystanie z serwera Express w projekcie JavaScript wymaga instalacji odpowiedniego modułu. Proces ten odbywa się poprzez otworzenie konsoli/terminala systemowego w głównym folderze projektu i wykonanie polecenia:

- ***npm install express***

W rezultacie menadżer pakietów środowiska Node.js pobierze i zainstaluje żądany moduł w lokalizacji z projektem.

Oprócz samego serwera potrzebna jest również obsługa sesji użytkownika, ścieżek dostępu do plików i folderów oraz bazy danych SQLite. Niezbędna jest więc instalacja odpowiednich pakietów za to odpowiadających:

- ***npm install express-session***
- ***npm install url***
- ***npm install path***
- ***npm install sqlite3***

5.2.2. Budowa serwera

Kod serwera można podzielić na kilka części. W pierwszej znajdują się polecenia importujące potrzebne do działania moduły.

```
import express from 'express';
import session from 'express-session';
import { fileURLToPath } from 'url';
import path from 'path';
import sqlite3 from 'sqlite3';
```

Blok kodu 1. *Budowa serwera: Importowanie modułów*
Źródło: Opracowanie własne

Kolejna część zawiera polecenia pobierające nazwę pliku i ścieżkę do lokalizacji w której element się znajduje. Zdefiniowany jest tu również serwer Express, wraz z jego konfiguracją, port na którym ma być uruchamiany oraz zmienna służąca w późniejszym etapie do przechowywania bazy danych.

```
const __filename = fileURLToPath(import.meta.url);
const __dirname = path.dirname(__filename);
const databasePath = path.join(__dirname,'Database','gamebase.db');
const app = express();
const port = process.env.PORT || 3000;
let db = null;
```

Blok kodu 2. *Budowa serwera: Konfiguracja serwera*
Źródło: Opracowanie własne

Mając zdefiniowany serwer, należy skonfigurować sesję dla użytkownika w następujący sposób.

```
app.use(session({
  secret: 'session-secret-key',
  resave: false,
  saveUninitialized: true
}));
```

Blok kodu 3. *Budowa serwera: Konfiguracja sesji użytkownika*
Źródło: Opracowanie własne

Aby serwer brał pod uwagę zawartość folderów w projekcie, niezbędnym jest zdefiniowanie ścieżek do ich lokalizacji.

```
app.use(express.json());
app.use(express.static(path.join(__dirname, 'public')));
app.use('/Extras/Cursors', express.static(path.join(__dirname,
'/Extras/Cursors')));
app.use('/Extras/Fonts', express.static(path.join(__dirname,
'/Extras/Fonts')));
app.use('/Extras/Sounds', express.static(path.join(__dirname,
'/Extras/Sounds')));
app.use('/Graphics/GUI', express.static(path.join(__dirname,
'/Graphics/GUI')));
app.use('/Graphics/Heroes', express.static(path.join(__dirname,
'/Graphics/Heroes')));
app.use('/Graphics/NPCs', express.static(path.join(__dirname,
'/Graphics/NPCs')));
app.use('/Graphics/Maps', express.static(path.join(__dirname,
'/Graphics/Maps')));
app.use('/Graphics/Monsters', express.static(path.join(__dirname,
'/Graphics/Monsters')));
app.use('/Graphics/Weapons', express.static(path.join(__dirname,
'/Graphics/Weapons')));
```

Blok kodu 4. *Budowa serwera: Ustawienie ścieżek do plików statycznych*
Źródło: Opracowanie własne

W środkowej części pliku wykonywane są metody *HTTP*[22] dostarczone przez interfejs *REST API*. Poniższy rysunek ukazuje niektóre z żądań *GET*, które umożliwiają wykonywanie plików JavaScript przez serwer.

```
app.get('/', (req, res) => {
  res.sendFile(path.join(__dirname, 'public', 'login.html'));
});

app.get('/login', (req, res) => {
  res.sendFile(path.join(__dirname, 'public', 'login.html'));
});

app.get('/game', (req, res) => {
  res.sendFile(path.join(__dirname, 'public', 'game.html'));
});

app.get('/login.mjs', (req, res) => {
  res.sendFile(path.join(__dirname, 'javascript', 'login.mjs'));
});
```

[22] https://pl.wikipedia.org/wiki/Hypertext_Transfer_Protocol

```
app.get('/skrypt.mjs', (req, res) => {
  res.sendFile(path.join(__dirname, 'javascript', 'skrypt.mjs'));
});
app.get('/progress.mjs', (req, res) => {
  res.sendFile(path.join(__dirname, 'javascript', 'progress.mjs'));
});
```

Blok kodu 5. *Budowa serwera: Metody HTTP*
Źródło: Opracowanie własne

W celu uruchomienia komunikacji z serwerem niezbędne jest ustawienie jego *IP* wraz portem. W efekcie dodany jest również komunikat informujący użytkownika o działaniu serwera.

```
app.listen(port, '0.0.0.0', () => {
  console.log(`Serwer działa na porcie ${port}`);
});
```

Blok kodu 6. *Budowa serwera: Uruchomienie serwera*
Źródło: Opracowanie własne

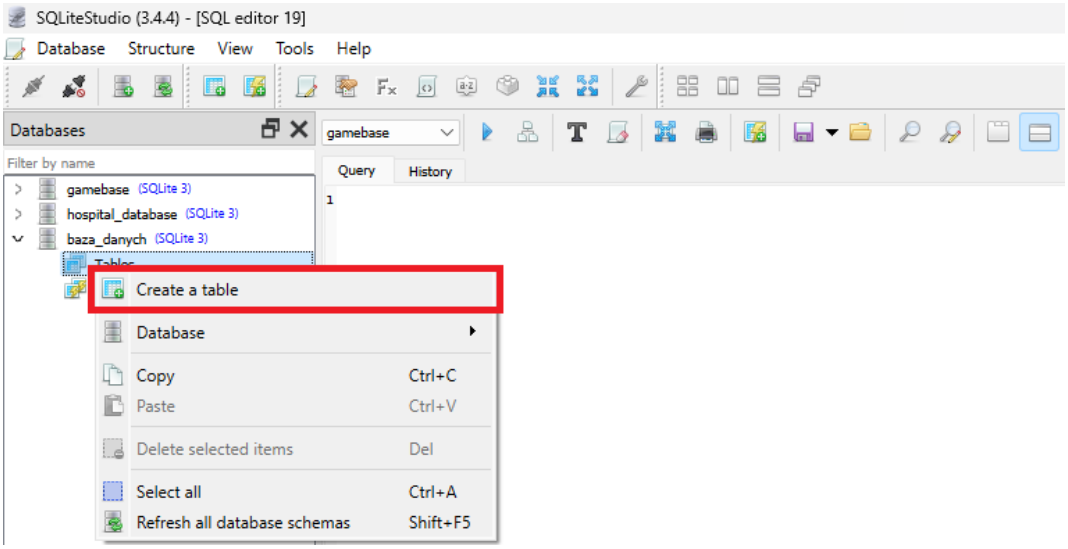
5.3. Implementacja bazy danych

5.3.1. Instalacja oprogramowania

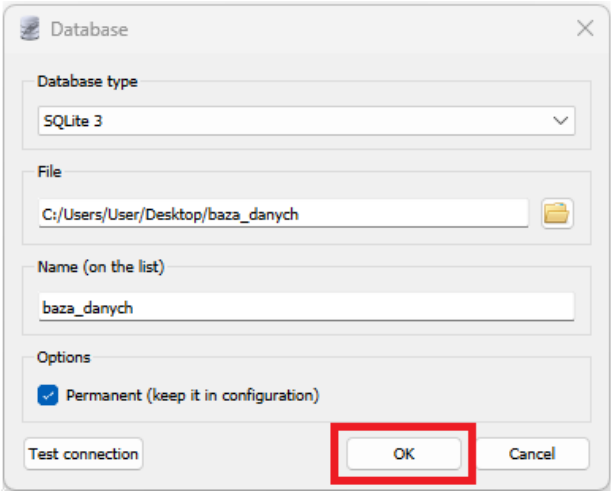
Pierwszym krokiem podczas procesu implementacji bazy danych jest zainstalowanie programów SQLiteStudio i DB Browser for SQLite.

5.3.2. Utworzenie bazy danych wraz z tabelami

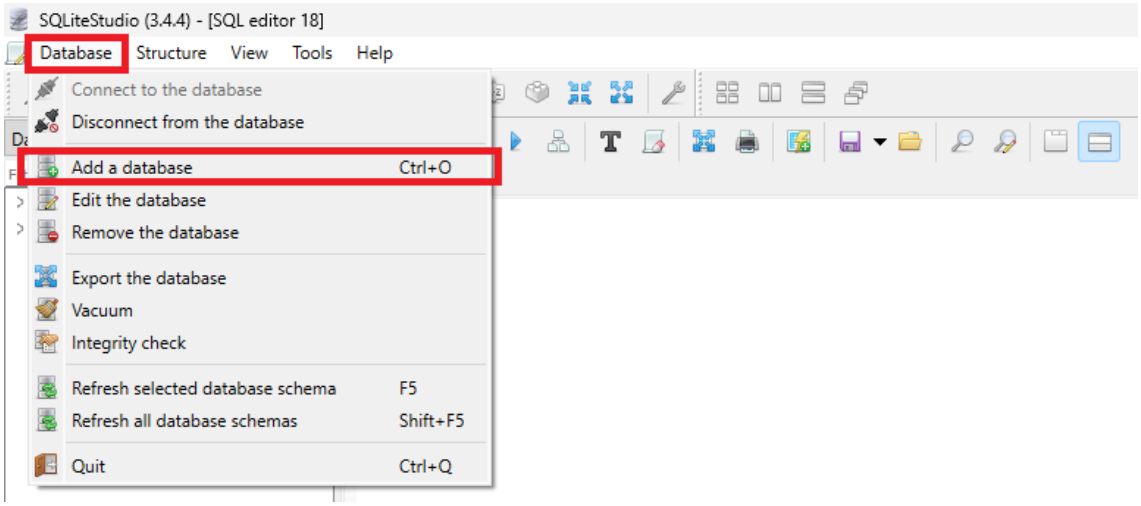
Tworzenie bazy danych wraz z tabelami odbywa się w programie SQLiteStudio. Cały proces pokazany jest po kolei etapami na zdjęciach poniżej.



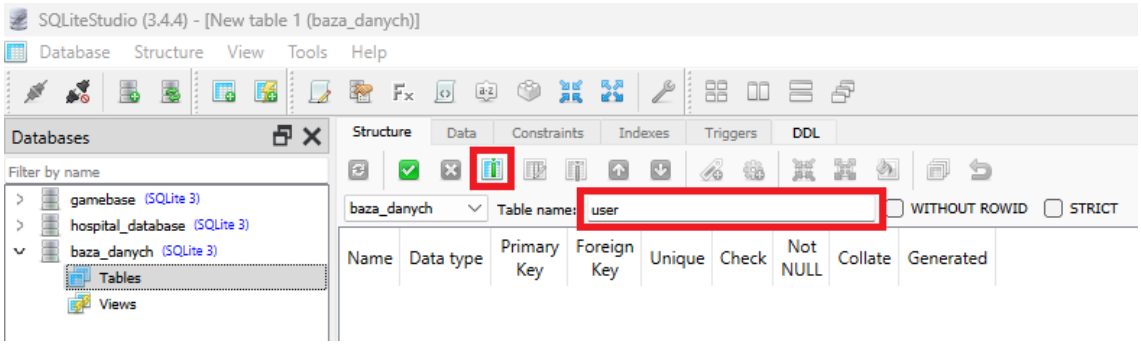
Rysunek 13. *Proces tworzenia bazy danych: Dodanie nowej bazy danych 1*
Źródło: Opracowanie własne



Rysunek 14. *Proces tworzenia bazy danych: Dodanie nowej bazy danych 2*
Źródło: Opracowanie własne

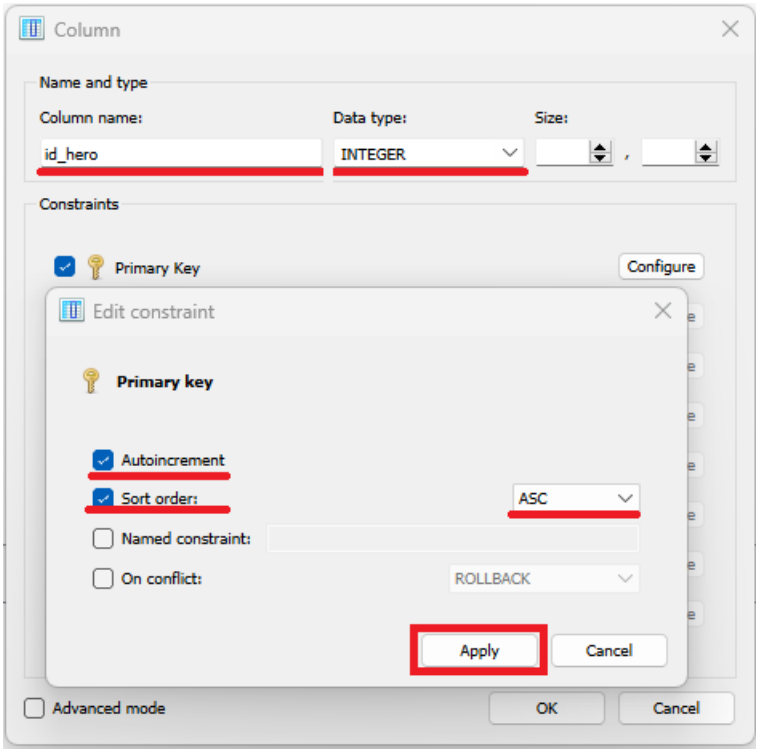


Rysunek 15. *Proces tworzenia bazy danych: Dodanie nowej bazy danych 3*
Źródło: Opracowanie własne

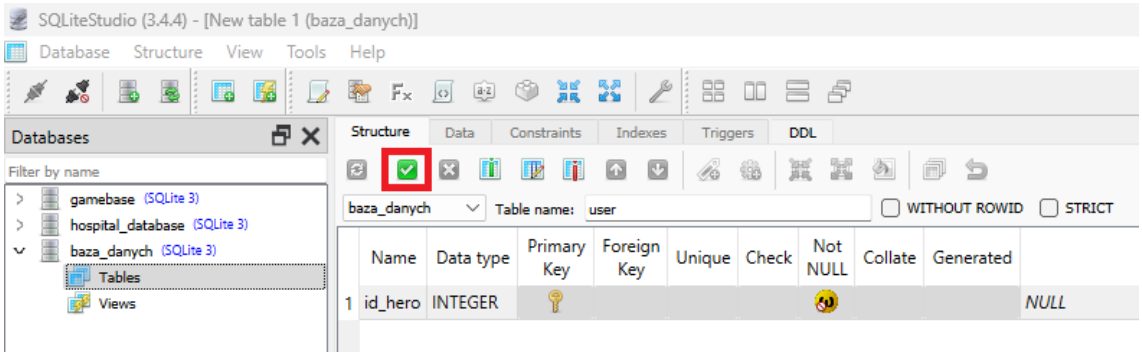


Rysunek 16. *Proces tworzenia bazy danych: Dodanie nowej bazy danych 4*
Źródło: Opracowanie własne

Rysunek 17. *Proces tworzenia bazy danych: Dodanie nowej bazy danych 5*
Źródło: Opracowanie własne



Rysunek 18. *Proces tworzenia bazy danych: Dodanie nowej bazy danych 6*
Źródło: Opracowanie własne



Rysunek 19. *Proces tworzenia bazy danych: Dodanie nowej bazy danych 6*
Źródło: Opracowanie własne

Powyższe kroki pokazują proces tworzenia nowej bazy danych i tabeli wraz z jedną kolumną o nazwie **id_hero**. Analogicznie etapy te zostały powtórzone w celu utworzenia wszystkich tabel i kolumn tak jak zostało to przedstawione w projekcie bazy danych (Rys. 5). Kod SQL[2] potrzebny do utworzenia tabel dla projektu:

```
CREATE TABLE hero (
    id_hero         INTEGER PRIMARY KEY AUTOINCREMENT,
    hero_name       TEXT    NOT NULL,
    position_x      INTEGER DEFAULT ( -430),
    position_y      INTEGER DEFAULT ( -6750),
    current_stage   INTEGER DEFAULT (1),
    killed_monsters INTEGER DEFAULT (0),
    founded_places  TEXT,
    id_user         INTEGER NOT NULL,
    FOREIGN KEY (
        id_user
    )
    REFERENCES user (id_user)
);

CREATE TABLE inventory (
    id_inv_item INTEGER   NOT NULL
                          PRIMARY KEY ASC AUTOINCREMENT,
    item_name   TEXT (25),
    frame       TEXT (10),
    id_user     INTEGER   NOT NULL,
    FOREIGN KEY (
        id_user
    )
    REFERENCES user (id_user)
);

CREATE TABLE items (
    id_ite_item    INTEGER   NOT NULL,
    item_image     TEXT (15) NOT NULL,
    item_type      TEXT (10) NOT NULL,
    item_name      TEXT (25) NOT NULL,
    item_damage    INTEGER   NOT NULL,
    item_defense   INTEGER   NOT NULL,
    item_hitpoints INTEGER   NOT NULL,
    item_cost      INTEGER   NOT NULL,
    CONSTRAINT id_ite_item PRIMARY KEY (
        id_ite_item
    )
);
```

```
CREATE TABLE user (
     id_user  INTEGER   PRIMARY KEY ASC AUTOINCREMENT
                        NOT NULL,
     username TEXT (15) NOT NULL,
     password TEXT (15) NOT NULL
);
```

Blok kodu 7. *Tworzenie tabel*
Źródło: Opracowanie własne

5.3.3. Wdrożenie do serwera

W momencie, gdy baza danych została stworzona i znajduje się w odpowiednim katalogu głównej lokalizacji projektu, następuje proces podłączenia jej do serwera. Operacja odbywa się poprzez dodanie nowego żądania z połączeniem z bazą danych. Wykonanie żądania skutkuje również otrzymaniem informacji zwrotnej o pomyślnym (lub nie) połączeniu z bazą. Żądanie wykonywane jest zawsze podczas wczytywania strony z logowaniem i samą grą.

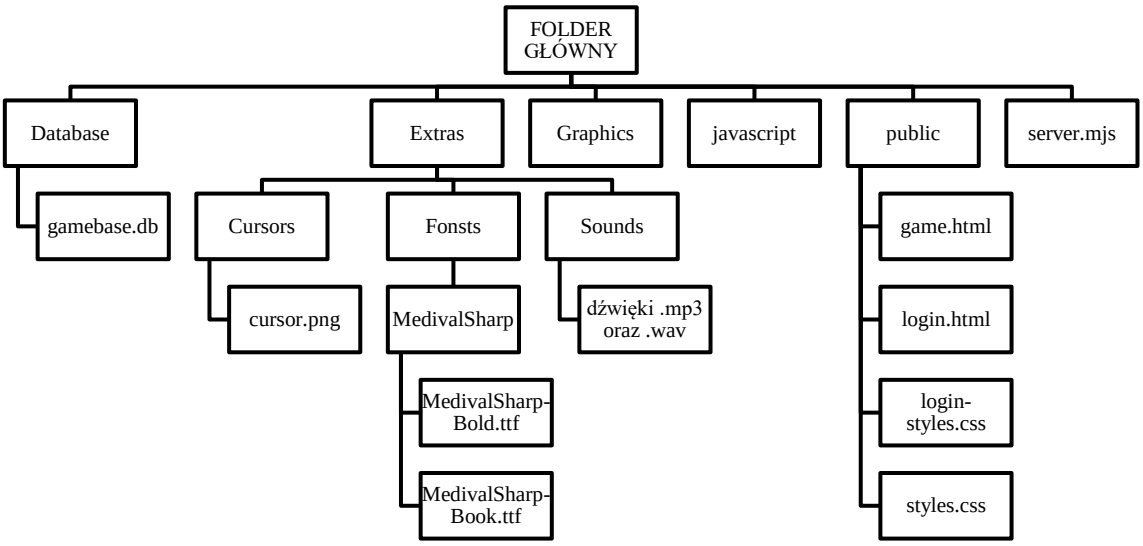
```
app.get('/game/database/connect', (req, res) => {
  db = new sqlite3.Database(databasePath, (err) => {
    if (err) {
      res.status(500).json({ message: err.message });
    } else {
      res.json({ message: 'Connected to the database successfully!'
      });
    }
  });
});
```

Blok kodu 8. *Żądanie połączenia z bazą danych*
Źródło: Opracowanie własne

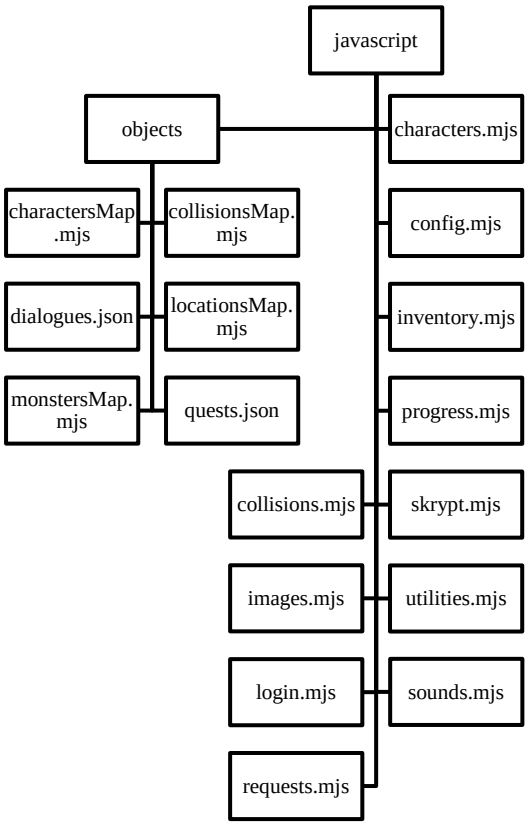
5.4. Implementacja aplikacji webowej

5.4.1. Budowa aplikacji

Aplikacja składa się z wielu plików zależnych od siebie, dlatego idealnym sposobem na przedstawienie całej struktury są diagramy poniżej.



Rysunek 20. *Struktura folderu głównego 1*
Źródło: Opracowanie własne



Rysunek 21. *Struktura folderu głównego 2*
Źródło: Opracowanie własne

Wszystkie pliki z językiem skryptowym JavaScript[3] zapisane są z rozszerzeniem **.mjs**. Rozszerzenie to jest niezbędne, aby zapewnić integralność plików ze środowiskiem uruchomieniowym Node.js oraz umożliwić operacje *import/export*. Pochodzi ono z modułu ECMAScript[23]. Pierwotnie Node.js obsługiwał moduł CommonJS[24], jednakże podczas ciągłej ewaluacji środowiska, dodano również wsparcie dla plików bazujących na module ECMAScript. Opis plików znajdujących się w projekcie:

- **server.mjs** – Kod źródłowy serwera.
- **game.html** – Kod strony internetowej w której uruchamiana jest gra. Zapisany w języku HTML[4]**.**
- **login.html** – Kod strony internetowej w której uruchamiany jest formularz logowania oraz rejestracji. Zapisany w języku HTML.
- **login-styles.css** – Kod służący do stylizacji elementów strony internetowej **login.html** zapisany w języku CSS.
- **styles.css** - Kod służący do stylizacji elementów strony internetowej **game.html** zapisany w języku CSS.
- **characters.mjs** – Plik z kodem JavaScript[5]. Znajdują się tu klasy opisujące bohatera, postacie niezależne oraz potwory wraz z metodami oraz funkcje niezależne.
- **collisions.mjs** – Zawarta jest tutaj klasa opisująca element kolizyjny wraz z metodą i funkcjami niezależnymi służącymi m.in. do tworzenia nowych obiektów (elementów kolizyjnych) i obliczania odległości bohatera lub potwora od najbliższej przeszkody.
- **config.mjs** – Plik konfiguracyjny w którym znajdują się stałe opisujące różne elementy gry. Wartości są eksportowane do konkretnych plików, gdzie są wykorzystywane.
- **images.mjs** – Znajdują się w nim stałe do których przypisane są obiekty *Image* oraz funkcje przypisujące do nich konkretne pliki graficzne. Elementy te są również eksportowane do innych plików w projekcie.

[23] https://ecma-international.org/technical-committees/tc39/
[24] https://en.wikipedia.org/wiki/CommonJS

- **inventory.mjs** – Zdefiniowane są tutaj klasy opisujące miejsce w ekwipunku oraz przedmioty wraz z ich metodami, funkcje niezależne powiązane z ekwipunkiem bohatera i tworzeniem obiektów.
- **login.mjs** – Kod JavaScript[6] obsługujący stronę logowania i wszystkie elementy, które się na niej znajdują.
- **progress.mjs** – Kod, który odpowiada za obsługę mechanizmu odkrywania nowych lokacji i wykonywania zadań.
- **requests.mjs** – Zdefiniowane i eksportowane są tutaj funkcje asynchroniczne wysyłające żądania do serwera.
- **skrypt.mjs** – Główny skrypt aplikacji odpowiadający za uruchomienie i sprawdzenie poprawności poszczególnych funkcji. Jest wykonywany w pierwszej kolejności, podczas ładowania strony z grą.
- **sounds.mjs** – W kodzie zdefiniowane i eksportowane są obiekty audio oraz ich konfiguracja.
- **utilities.mjs** – W pliku znajdują się funkcje użytkowe, wykorzystywane podczas działania aplikacji.
- **charactersMap.mjs** – Plik z którego eksportowana jest tablica z danymi o lokalizacji umieszczonych na mapie postaci niezależnych.
- **collisionsMap.mjs** - Plik z którego eksportowana jest tablica z danymi o lokalizacji umieszczonych na mapie elementów kolizyjnych.
- **dialogues.json** – Znajdują się tutaj treści dialogów między bohaterem a postaciami niezależnymi wraz informacjami o zadaniach w których występują.
- **locationsMap.mjs** - Plik z którego eksportowana jest tablica z danymi o lokalizacji umieszczonych na mapie lokacji do odkrycia.
- **monstersMap.mjs** - Plik z którego eksportowana jest tablica z danymi o lokalizacji miejsc występowania potworów.

- **quests.json** – Plik z treścią dostępnych zadań w grze. Każde zadanie zapisane jest w postaci obiektu i zawiera informacje o jego zrealizowaniu.

ROŹDZIAŁ 6
FUNKCJONALNOŚCI

6.1. Panel logowania/rejestracji

Pierwszą funkcjonalnością aplikacji, na którą natrafia użytkownik jest panel logowania[10][11].

Rysunek 22. *Panel logowania*
Opracowanie własne na podstawie: Wygenerowanej grafiki przez Microsoft Copilot, data dostępu: 04.01.2024 r.

6.1.1. Logowanie

Proces logowania obsługuje metoda **addEventListener()** interfejsu EventTarget. Celem zdarzenia typu **click** jest przycisk **loginButton**. Po interakcji użytkownika z tym elementem wykonywana jest funkcja asynchroniczna w której przeprowadzana jest walidacja wprowadzonych danych oraz wysyłanie żądania do serwera.

```
loginButton.addEventListener('click', async function() {
    sound.CLICK_SOUND.play();
    const username = document.getElementById('username').value;
    const password = document.getElementById('password').value;

    if (username.length < 3 || password.length < 3) {
        const message = `Nazwa użytkownika i/lub hasło jest za
         krótkie`;
        return displayErrorMessage(message);
    }

    if (!await request.loginRequest(username, password)) {
        const message = `Nieprawidłowa nazwa użytkownika lub hasło!`;
        return displayErrorMessage(message);
    }
    loadGame();
})
```

Blok kodu 9. *Proces logowania*
Źródło: Opracowanie własne

Po pomyślnej weryfikacji danych logowania, użytkownik przekierowywany jest do strony z grą. Przekierowanie obsługuje funkcja **loadGame()**.

```
function loadGame() {
    return window.location.assign('http://192.168.1.138:3000/game');
}
```

Blok kodu 10. *Przekierowanie do strony z grą*
Źródło: Opracowanie własne

6.1.2. Rejestracja

Rejestracja nowego użytkownika przebiega w bardzo podobny sposób co logowanie. Po wciśnięciu przycisku „Nowa gra”, wysyłane jest żądanie do serwera w celu sprawdzenia czy użytkownik już istnieje w bazie danych. Jeżeli odpowiedź nie jest negatywna, po stronie serwera wykonywana jest operacja dodawania nowego użytkownika wraz z tworzeniem nowego bohatera i jego ekwipunku startowego.

```
registerButton.addEventListener('click', async function() {
    sound.CLICK_SOUND.play();
    const username = document.getElementById('username').value;
    const password = document.getElementById('password').value;

    if (username.length < 3 || password.length < 3) {
```

```
            const message = `Nazwa użytkownika i/lub hasło jest za
             krótkie`;
            return displayErrorMessage(message);
        }

        if (!await request.registerRequest(username, password)) {
            const message = `Bohater o podanym imieniu już istnieje!`;
            return displayErrorMessage(message);
        }
        if (!await request.createHeroRequest()) {
            const message = `Błąd podczas tworzenia bohatera!`;
            return displayErrorMessage(message);
        }
        for (let i = 0; i < 14; i++) {
            const frame = 'frame' + i;
            if (!await request.createInventoryRequest(frame)) {
                const message = `Błąd podczas tworzenia ekwipunku!`;
                return displayErrorMessage(message);
            }
        }
        loadGame();
})
```

Blok kodu 11. *Proces rejestracji*
Źródło: Opracowanie własne

Pomyślne wykonanie operacji kończy się przekierowaniem użytkownika do strony z grą.

6.2. Gra

Funkcjonalności stricte związane z samą rozgrywką dotyczą głównie interakcji z ekwipunkiem, menu, działania minimapy, dziennika zadań, mechaniki związanej z ruchem postaci, interakcji z obiektami, systemem walki, wykonywania zadań i odkrywania nowych lokacji.

6.2.1. Ekwipunek

Otwarcie ekwipunku bohatera odbywa się poprzez wciśnięcie klawisza „I”. Kod klawisza przechwytywany jest dzięki zdarzeniu **onkeydown** połączonym ze zdarzeniem **KeyboardEvent.KeyCode**.

```
this.document.onkeydown = async function(e) {
        switch (e.keyCode) {
            case 73:
                if (e.repeat) {
                    break;
                }
                inventoryButton.style.filter = "contrast(50%)";
                drawInventory();
                break;
}}
```

Blok kodu 12. *Otwieranie ekwipunku 1*
Źródło: Opracowanie własne

Następnie wykonywana jest funkcja **drawInventory()** wyświetlająca ekwipunek. W sytuacji, gdy ekwipunek jest już otwarty, zostaje on zamknięty.

```
function drawInventory() {
        const inventoryWindow = document.querySelector('.inventory');
        if (isInvenoryOpen) {
            isInvenoryOpen = false;
            inventory.drawItems();
            inventoryWindow.style.display = 'none';
        } else if (!isInvenoryOpen) {
            isInvenoryOpen = true;
            inventory.writeHeroStats();
            inventory.writeHeroMoney();
            inventory.drawItems();
            inventoryWindow.style.display = 'block';
        }
}
```

Blok kodu 13. *Otwieranie ekwipunku 2*
Źródło: Opracowanie własne

6.2.1.1. Wyświetlanie szczegółów przedmiotu

Jeżeli bohater posiada jakikolwiek przedmiot na wyposażeniu, gracz może podejrzeć parametry tego przedmiotu poprzez najechanie kursorem na ramkę w której się znajduje.

Rysunek 23. *Wyświetlanie szczegółów przedmiotu*
Źródło: Opracowanie własne

Każda ramka w ekwipunku jest obiektem klasy **InvenotryFrame** i posiada zdarzenia **mouseover** i **mouseout** obsługujące ten proces.

```
export class InventoryFrame {
    constructor(number, x, y, item=null) {
        this.frame = document.createElement('div');
        this.frame.setAttribute('class', 'itemFrame');
        this.frame.id = 'frame' + number;
        this.frame.style.top = y + 'px';
        this.frame.style.left = x + 'px';
        this.frame.addEventListener('mouseover',
         this.mouseOverEffect.bind(this));
        this.frame.addEventListener('mouseout',
         this.mouseOutEffect.bind(this));
        this.frame.addEventListener('mousedown',
         this.chooseItem.bind(this));
```

```
        gameArea.appendChild(this.frame);
        this.isFrameEmpty = true;
        this.item = item;
        this.setParameters();
    }
```

Blok kodu 14. *Klasa InventoryFrame*
Źródło: Opracowanie własne

```
mouseOverEffect() {
        if (isInvenoryOpen) {
            this.frame.style.opacity = 0.2;
            onHoverFrame = this;
        }

        if (isInvenoryOpen && !this.isFrameEmpty) {
            const informationFrame =
             document.querySelector('.itemDetailsFrame');
            const firstTextLine = '<span style="color:
             #936537;">Nazwa: <span style="color: whitesmoke;">'
             + this.item.name + '</span> </span>';
            let secondTextLine = '';
            let thirdTextLine = '';
            let fourthTextLine = '';
            let describtionTextLine = '';
            switch (this.item.type) {
                case 'weapon':
                    secondTextLine = '<span style="color:
                     #936537;">Obrażenia: <span style="color:
                     whitesmoke;">' + this.item.damage +
                     '</span> </span>';
                    describtionTextLine = '<span style="font-
                     size: 0.7rem; color: #936537; font-style:
                     italic;"> "Muszę uważać żeby się nie
                     skaleczyć!" </span>';
                    break;

                default:
                    secondTextLine = '<span style="color:
                     #936537;">Punkty życia: <span style="color:
                     whitesmoke;">' + this.item.hitpoints +
                     '</span> </span>';
                    thirdTextLine = '<span style="color:
                     #936537;">Obrona: <span style="color:
                     whitesmoke;">' + this.item.defense +
                     '</span> </span>'
                    describtionTextLine = '<span style="font-
                     size: 0.7rem; color: #936537; font-style:
```

```
                    italic;"> "Dzięki temu czuję się
                    bezpiecznie!" </span>';
                   break;
            }

            informationFrame.innerHTML = firstTextLine +
             secondTextLine + thirdTextLine +
             describtionTextLine;
            informationFrame.style.display = 'grid';
            gameArea.addEventListener('mousemove',
             this.moveItem.bind(this));
        }
}
```

Blok kodu 15. *Metoda mouseOverEffect*
Źródło: Opracowanie własne

```
mouseOutEffect() {
        const informationFrame =
         document.querySelector('.itemDetailsFrame');
        informationFrame.style.display = 'none';
        this.frame.style.opacity = 0;
        onHoverFrame = null;
}
```

Blok kodu 16. *Metoda mouseOutEffect*
Źródło: Opracowanie własne

6.2.1.2. Podnoszenie przedmiotu

Przedmiot znajdujący się w ekwipunku można przenieść w celu ubrania go na swojego bohatera lub zmiany jego miejsca. W tym celu wystarczy na niego najechać kursorem[12] myszy i kliknąć lewy przycisk myszy. Przedmiot zostanie podniesiony.

Rysunek 24. *Podnoszenie przedmiotu*
Źródło: Opracowanie własne

Przedmiot zostanie „przyczepiony” do kursora myszy i będzie przesuwał się razem z nim do momentu umieszczania go w nowym miejscu. Zdarzenie obsługuje metoda **chooseItem()**.

```
chooseItem() {
        if (isInvenoryOpen) {
            if (!this.isFrameEmpty && pickedItem == null) {
                pickedItem = this.item;
                this.pickUpItem();
                if (wearableFrames.includes(this.frame.id)) {
                    calculateStats(pickedItem, 'takeoff');
                }
```

```
    this.isFrameEmpty = true;
    this.item = null;
    gameArea.addEventListener('mousemove',
      this.moveItem.bind(this));
}
```

Blok kodu 17. *Podnoszenie przedmiotu*
Źródło: Opracowanie własne

6.2.1.3. Upuszczanie przedmiotu

Upuszczenie aktualnie podniesionego przedmiotu pozwala na umiejscowienie go w wybranym przez gracza okienku ekwipunku. Proces obsługuje dalszy fragment kodu z metody powyżej (blok kodu 16).

```
else if (this.isFrameEmpty && pickedItem != null &&
  this.checkItemType(pickedItem)) {
    this.placeItem();
    onHoverFrame.setParameters();
    saveInventory(this.item.name, this.frame.id);
    pickedItem = null;
}

    else if (!this.isFrameEmpty && pickedItem !=
      null) {
    const inFrameItem = onHoverFrame.item;
    const itemToPlace = pickedItem;

    const itemToPlaceID =
      document.getElementById(pickedItem.getItemID()
      );
    this.placeItem(itemToPlaceID);
    onHoverFrame.item = itemToPlace;
    onHoverFrame.isFrameEmpty = false;
    pickedItem = inFrameItem;
    this.pickUpItem();
    saveInventory(itemToPlace.name, this.frame.id);
}}}
```

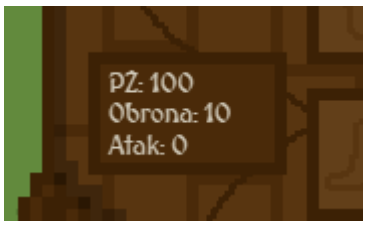
Blok kodu 18. *Upuszczanie przedmiotu*
Źródło: Opracowanie własne

6.2.1.4. Zakładanie przedmiotu

Niektóre z przedmiotów mogą zostać założone na bohatera. Podczas umieszczania przedmiotu w danym okienku w ekwipunku, zostaje wykonana funkcja warunkowa, sprawdzająca czy docelowe okienko jest miejscem na przedmiot ubioru lub broni. Po założeniu lub ściągnięciu przedmiotu aktualizowane są statystyki bohatera.

6.2.1.5. Statystyki bohatera

Statystyki bohatera obejmują kolejno punkty życia (PŻ), obrony oraz ataku. Pokazane są po lewej stronie w ekwipunku.



Rysunek 25. *Statystyki*
Źródło: Opracowanie własne

6.2.2. Menu

Otwarcie menu odbywa się poprzez naciśnięcie klawisza „L”. Proces obsługuje zdarzenie **KeyboardEvent.KeyCode**.

```
case 76:
     if (e.repeat) {
          break;
     }
     settingsButton.style.filter = "contrast(50%)";
     drawSettings();
     break;
```

Blok kodu 19. *Otwarcie menu*
Źródło: Opracowanie własne

Rysunek 26. *Menu*
Źródło: Opracowanie własne

6.2.2.1. Wczytywanie ostatniego zapisu gry

Kliknięcie lewym przyciskiem myszy pierwszego przycisku w menu, skutkuje załadowaniem pliku HTML z grą. Wynikiem tej operacji jest wysłanie na nowo żądań do serwera w celu pobrania informacji z bazy danych o statystykach i przedmiotach bohatera.

```
loadButton.addEventListener('click', async function() {
        sound.CLICK_SOUND.play();
        window.location.assign('http://localhost:3000/game');
})
```

Blok kodu 20. *Interakcja z przyciskiem loadButton*
Źródło: Opracowanie własne

6.2.2.2. Wylogowanie się

Proces ten zaczyna się w momencie kliknięcia przez użytkownika drugiego przycisku w menu.

```
logoutButton.addEventListener('click', async function() {
        sound.CLICK_SOUND.play();
        const logout = await request.logoutRequest();
})
```

Blok kodu 21. *Interakcja z przyciskiem logoutButton*
Źródło: Opracowanie własne

Zostaje wykonane żądanie do serwera z informacją o chęci wylogowania się przez użytkownika. Pomyśle wykonanie operacji kończy się przeniesieniem na stronę logowania.

```
export async function logoutRequest() {
    try {
        const response = await fetch('/game/logout');
        if (!response.ok) {
            throw new Error('Failed to logout!');
        }
        const data = await response.json();

window.location.assign('http://localhost:3000/login');
        return true;
    } catch (error) {
        console.error('Request failed:', error);
        return false;
    }
}
```

Blok kodu 22. *Wysłanie żądania wylogowania*
Źródło: Opracowanie własne

Serwer odbiera żądanie kończy sesję dla aktualnie zalogowanego użytkownika.

```
app.get('/game/logout', (req, res) => {
  req.session.destroy(err => {
    if (err) {
      return res.json({ success: false, message: 'Error
       destroying session!' });
    } else {
      return res.json({ success: true });
    }
  });
});
```

Blok kodu 23. *Obsługa żądania wylogowania*
Źródło: Opracowanie własne

6.2.3. Minimapa

Minimapa jest elementem informującym gracza o aktualnej pozycji bohatera na mapie. Bohater oznaczony jest małym, czerwonym kwadratem.

Rysunek 27. *Minimapa*
Źródło: Opracowanie własne

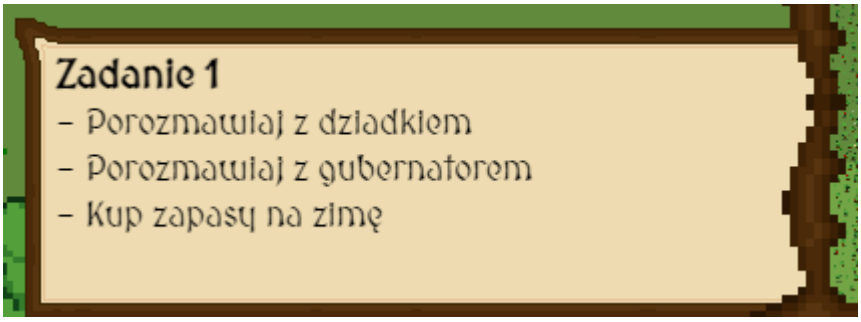
Funkcjonalność obsługuje funkcja **drawMinimap()**.

```
function drawMinimap() {
        const scale = minimapCanvas.width / image.backgroundForest.width;
        let heroMinimapPositionXScaled = -mapX * scale + hero.positionX *
         scale;
        let heroMiniapPositionYScaled = -mapY * scale + hero.positionY *
         scale;
        ctx.rect(canvas.width - minimapCanvas.width - 8, canvas.height -
         minimapCanvas.height - 8, minimapCanvas.width,
         minimapCanvas.height);
        ctx.fillStyle = "black";
        ctx.fill();
        ctx.drawImage(minimapCanvas, canvas.width - minimapCanvas.width - 8,
         canvas.height - minimapCanvas.height - 8);
        minimapCtx.drawImage(image.backgroundForest, 0, 0,
         image.backgroundForest.width, image.backgroundForest.height, 0, 0,
         image.backgroundForest.width * scale, image.backgroundForest.height
         * scale);
        minimapCtx.fillStyle = "red";
        minimapCtx.fillRect(heroMinimapPositionXScaled,
         heroMiniapPositionYScaled, hero.width * scale + 2, hero.height *
         scale + 2);
        ctx.drawImage(image.minimapFrame, canvas.width -
         image.minimapFrame.width, canvas.height -
         image.minimapFrame.height);
}
```

Blok kodu 24. *Rysowanie minimapy*
Źródło: Opracowanie własne

6.2.4. Dziennik zadań

Dziennik zadań znajduje się w prawym, dolnym rogu ekranu. Wyświetlane są w nim aktualnie wykonywane zadania.



Rysunek 28. *Dziennik zadań*
Źródło: Opracowanie własne

Wyświetlanie ramki wraz z zadaniami obsługuje funkcja **displayQuestFrame()**.

```
export function displayQuestFrame(ctx, textCtx, minimap) {
    const minimapDimension = minimap.width;
    const posX = Math.round(config.CANVAS_WIDTH - minimapDimension
     - image.questFrame.width - 15);
    const posY = Math.round(config.CANVAS_HEIGHT -
     image.questFrame.height);
    ctx.drawImage(image.questFrame, posX, posY);
    const textPosX = posX + 20;
    let textPosY = 0;
    const currentStage = characters.hero.statistics.stage;
    const questsCount = Object.keys(quest[currentStage]).length;
    for (let i = 0; i < questsCount; i++) {
        const text = quest[currentStage][i + 1]['quest'];
        textPosY = Math.round(posY + 23 * (i + 2.5));
        if (quest[currentStage][i + 1]['done']) {
            textCtx.font = `20px ${config.FONT_FAMILY}`;
            const textWidth = textCtx.measureText(text).width;
            crossOutText(textPosX, textPosY, textWidth)
        }
        writeQuestText('- ' + text, 17, textPosX, textPosY);
    }

    writeQuestText('Zadanie ' + characters.hero.statistics.stage,
     20, textPosX, posY + 35, 'bold');
```

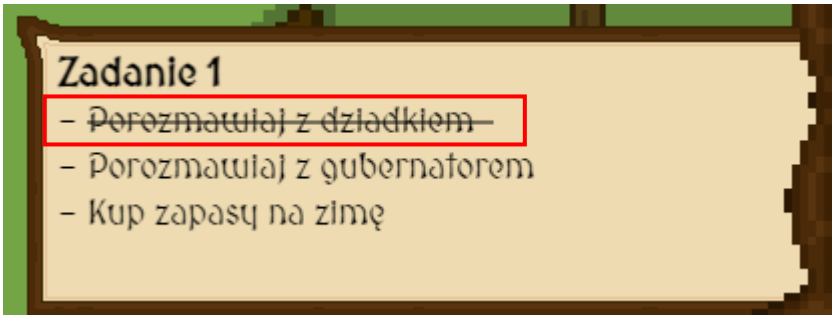
```
    function writeQuestText(content, size, x, y, bold = '') {
            const text = content;
            const fontSize = size;
            textCtx.font = `${bold} ${fontSize}px
             ${config.FONT_FAMILY}`;
            textCtx.fillStyle = 'black';
            textCtx.fillText(text, x, y);
    }

    function crossOutText(ax, ay, width) {
            textCtx.strokeStyle = "black";
            textCtx.beginPath();
            textCtx.moveTo(ax + 15, ay - 5);
            textCtx.lineTo(ax + width - 10, ay - 5);
            textCtx.stroke();
    }
}
```

Blok kodu 25. *Wyświetlanie dziennika zadań*
Źródło: Opracowanie własne

6.2.5. System wykonywania zadań

W grze do wykonania są trzy zadania. Każde z nich zawiera po trzy etapy i są one opisane w dzienniku. Gdy gracz zrealizuje któryś z etapów zadania zostanie on przekreślony, a po wykonaniu wszystkich trzech na ekranie zostaje wyświetlony stosowny napis. Operacje obsługuje poprzedni fragment kodu (blok kodu 24).

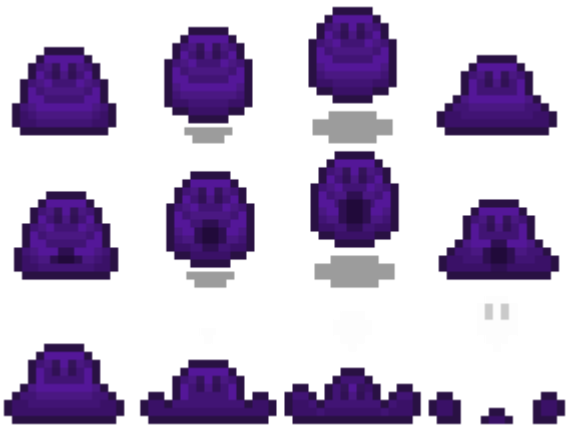


Rysunek 29. *Zakończony etap zadania*
Źródło: Opracowanie własne

6.2.6. Animacja ruchu

Mechanizm animacji ruchu postaci w grze można porównać do zmiany slajdów. Każda z postaci przedstawiona jest jako plik graficzny w którym narysowana jest każda jej animacja. Program pobiera dany ruch postaci w odpowiednim interwale czasowym tworząc na ekranie animacje ruchu.

Rysunek 30. *Animacja ruchu bohatera*
Źródło: Opracowanie własne



Rysunek 31. *Animacja ruchu potwora*
Źródło: Opracowanie własne

Kod obsługujący zmianę ruchu każdej z postaci jest bardzo podobny. Poniżej pokazana jest przykładowa metoda klasy **Hero** obsługująca tą funkcjonalność dla bohatera głównego.

```
async move() {
        this.stepInterval++;
        while (this.stepInterval == 2) {
            if (this.frameX < 3) {
                this.frameX++;
            }
            else {
                this.frameX = 0;
            }
            this.stepInterval = 0;
        }
}
```

Blok kodu 26. *Metoda obsługująca animacje ruchu*
Źródło: Opracowanie własne

6.2.7. Interakcje z obiektami

6.2.7.1. Interakcja z przeszkodami

Bohater podczas przemierzania mapy napotka różne elementy otoczenia takie jak drzewa, mury, czy rzeki. Elementy te są naturalnymi przeszkodami, przez które nie da się przejść. Funkcja sprawdzająca otoczenie wykonuje się w zależności od obiektu którego dotyczy. Może to być bohater główny lub potwór ścigający gracza.

```
export function checkObstacleCollision(object, direction) {
    const objectCenterX = object.positionX + object.width/2;
    const objectCenterY = object.positionY + object.height/1.5;
    const obstacle = findNearestObstacle(object);
    const obstacleCenterX = obstacle.positionX +
     obstacle.width/2;
    const obstacleCenterY = obstacle.positionY +
     obstacle.height/2;
    const heroRadius = Math.min(object.width,
     object.height)/2.7;
    const distanceX = Math.abs(objectCenterX - obstacleCenterX);
    const distanceY = Math.abs(objectCenterY - obstacleCenterY);

    if (direction === 'up' && objectCenterY > obstacleCenterY &&
     distanceX <= heroRadius && distanceY <= heroRadius +
     object.stepLength) {
        return true;
    }
    else if (direction === 'down' && objectCenterY <
     obstacleCenterY && distanceX <= heroRadius && distanceY <=
     heroRadius + object.stepLength) {
        return true;
    }
    else if (direction === 'left' && objectCenterX >
     obstacleCenterX && distanceX <= heroRadius +
     object.stepLength && distanceY <= heroRadius) {
        return true;
    }
    else if (direction === 'right' && objectCenterX <
     obstacleCenterX && distanceX <= heroRadius +
     object.stepLength && distanceY <= heroRadius) {
        return true;
    }
    return false;
}
```

Blok kodu 27. *Funkcja sprawdzająca kolizję*
Źródło: Opracowanie własne

6.2.7.2. Interakcja z bohaterem niezależnym

Gracz przemierzając mapę natknie się na wiele postaci niezależnych (NPC) z którymi może wejść w interakcje poprzez rozpoczęcie dialogu. Dialog rozpoczyna się w momencie znalezienia się bohatera w pobliżu któregoś z NPC i naciśnięcia przez gracza klawisza „T”.

Rysunek 32. *Interakcja z NPC*
Źródło: Opracowanie własne

Rysunek 33. *Dialog*
Źródło: Opracowanie własne

Za wyświetlanie dialogu odpowiedzialne jest zdarzenie **KeyboardEvent.KeyCode**.

```
case 84:
        keyState["KeyT"] = true;
        if (characters.displayInteractionText(textCtx) &&
         !hero.conversation.isTalking) {
              hero.conversation.isTalking = true;
        }
        else if (!characters.displayInteractionText(textCtx) ||
         hero.conversation.isTalking) {
             hero.conversation.isTalking = false;
             hero.conversation.role = 'answer';
             hero.conversation.dialogue = 0;
        }
        break;
```

Blok kodu 28. *Wyświetlanie dialogu*
Źródło: Opracowanie własne

6.2.8. System walki

Jedną z ważniejszych funkcjonalności dających rozrywkę w grze jest system walki. Gracz może zmierzyć się z potworami rozsianymi po całej mapie. System walki obejmuje zadawanie obrażeń przez bohatera oraz otrzymywanie ich. Istnieje również możliwość otrzymania obrażeń śmiertelnych które dyskwalifikują gracza z dalszej rozgrywki. To samo dotyczy się przeciwników.

6.2.8.1. Zadawanie obrażeń

Zadawanie obrażeń potworom odbywa się poprzez naciśnięcie klawisza spacji i wykonaniu zamaszystego cięcia mieczem. Jeżeli bohater znajduje się odpowiednio blisko przeciwnika, zada mu obrażenia. Zadanie obrażeń sygnalizowane jest efektem dźwiękowym.

Rysunek 34. *Atakowanie*
Źródło: Opracowanie własne

Zadawanie obrażeń obsługuje asynchroniczna metoda klasy **Hero**, która przyjmuje jako argument kierunek w którym bohater jest aktualnie obrócony.

```
async attack(direction) {
        if (this.isAttacking || this.isMoving ||
         this.conversation.isTalking) {
            return;
        }
        sound.ATTACK_SOUND.play();
        if (this.canAttack == true) {
            this.isAttacking = true;
            switch (direction) {
                case 'up':
                    this.frameY = 11;
                    break;

                case 'down':
                    this.frameY = 8;
                    break;

                case 'left':
                    this.frameY = 10;
                    break;

                case 'right':
                    this.frameY = 9;
                    break;
```

```
            }

            for (let i = 0; i < config.HERO_FRAMES_COUNT; i++) {
                const monster = nearestMonster();
                if (i == 2 && canDealDamage(this, monster)) {
                    monster.calculateHitpoints(10);
                }
                this.frameX = i;
                updateCanvasRequest();
                await sleep(125);
            }
            sound.ATTACK_SOUND.pause();
            sound.ATTACK_SOUND.currentTime = 0;
            this.isAttacking = false;
            this.breathing();
        }
    }
```

Blok kodu 29. *Zadawanie obrażeń przez bohatera*
Źródło: Opracowanie własne

6.2.8.2. Otrzymywanie obrażeń

Aby system walki miał sens przeciwnicy również mają możliwość zaatakowania bohatera. Proces ten opiera się o dwie asynchroniczne metody. Pierwszą jest metoda **attack()** klasy **Monster**, a drugą **receiveDamage()** klasy **Hero**.

```
async attack() {
        while (true) {
            if (canDealDamage(this, hero) && !isInvenoryOpen &&
             !this.flag && !initialScreenDisplayed) {
                this.frameY = 1;
                hero.receiveDamage(config.GOLEM_ATTACK);
            } else if (!canDealDamage(this, hero)){
                this.frameY = 0;
            }
            await sleep(800);
        }
}
```

Blok kodu 30. *Wykonywanie ataku przez potwora*
Źródło: Opracowanie własne

```
async receiveDamage(damage) {
        if (this.isDying) {
            return;
        }
        const armorFactor = this.defense / 100;
        const trueDamage = damage * (1 - armorFactor);
        if (this.health - trueDamage <= 0) {
            await this.die();
            return;
        }
        this.health -= trueDamage;
}
```

Blok kodu 31. *Otrzymywanie obrażeń przez bohatera*
Źródło: Opracowanie własne

6.2.8.3. Otrzymywanie śmiertelnych obrażeń

Zarówno bohater jak i stwory z którymi walczy, mogą otrzymać obrażenia które powodują koniec rozgrywki. A co za tym idzie, zmuszają gracza do rozpoczęcia gry od ostatniego puntu kontrolnego (zapisu).

```
async die() {
        this.isDying = true;
        this.frameY = 12;
        for (let i = 0; i < config.HERO_FRAMES_COUNT; i++) {
            this.frameX = i;
            updateCanvasRequest();
            await sleep(150);
        }
        this.isDead = true;
}
```

Blok kodu 32. *Otrzymanie śmiertelnych obrażeń przez bohatera*
Źródło: Opracowanie własne

```
async die() {
        hero.statistics.killedMonsters++;
        this.frameY = 2;
        for (let i = 0; i < config.GOLEM_FRAMES_COUNT - 1; i++){
            this.frameX = i;
            updateCanvasRequest();
            await sleep(180);
        }
}
```

Blok kodu 33. *Otrzymanie śmiertelnych obrażeń przez potwora*
Źródło: Opracowanie własne

Rysunek 35. *Koniec gry*
Źródło: Opracowanie własne

6.2.9. System odkrywania nowych lokacji

W grze występuje sześć lokacji, które gracz może odkryć. Każda lokacja jest obiektem klasy **Locations** i narysowana jest na mapie przezroczystym obszarem. W momencie, gdy bohater wejdzie w ten obszar zostaje wyświetlona informacja o odkryciu nowego miejsca wraz z efektem dźwiękowym.

```
class Locations {
    static width = config.SQUARE_WIDTH;
    static height = config.SQUARE_HEIGHT;
    constructor(name, x, y) {
        this.name = name;
        this.width = Locations.width;
        this.height = Locations.height;
        this.positionX = x;
        this.positionY = y;
        this.discovered = false;
    }

    draw(ctx) {
        ctx.fillStyle = 'rgba(255, 200, 255, 0.1)';
        ctx.fillRect(this.positionX, this.positionY, this.width,
         this.height);
    }
}
```

Blok kodu 34. *Klasa Locations*
Źródło: Opracowanie własne

```
export function discoverLocation() {
    const location = findNearestLocation();
    const flag = 'locationText';
    if (flags['locationText']) {
        setFlag(flag, false);
    }

    if
     (characters.hero.statistics.discoveredPlaces.includes(location.na
     me) || flags['locationText']) {
        return;
    }
    const heroCenterPointX = characters.hero.positionX +
     characters.hero.frameWidth / 2;
    const heroCenterPointY = characters.hero.positionY +
     characters.hero.frameHeight / 1.5;

    const locationCenterPointX = location.positionX +
     location.width/2;
    const locationCenterPointY = location.positionY +
     location.height/2;
    if (heroCenterPointX >= locationCenterPointX -
         config.SQUARE_WIDTH/2 &&
        heroCenterPointX <= locationCenterPointX +
         config.SQUARE_WIDTH/2 &&
        heroCenterPointY >= locationCenterPointY -
         config.SQUARE_HEIGHT / 2 &&
        heroCenterPointY <= locationCenterPointY +
         config.SQUARE_HEIGHT / 2) {

            characters.hero.statistics.discoveredPlaces.push(location.n
            ame);
           setFlag(flag, true);
    }
}
```

Blok kodu 35. *Odkrywanie nowej lokacji*
Źródło: Opracowanie własne

ROŹDZIAŁ 7
TESTOWANIE

7.1. Testy manualne

Aplikacja webowa została poddana testom manualnym polegającym na realizacji przez testerów określonych wcześniej scenariuszy w celu sprawdzenia jej użyteczności i poprawności funkcjonowania.

Testerami były trzy osoby chętne do wypróbowania produktu finalnego. Każda z nich miała przejść grę od początku do końca, a następnie uruchomić od nowa aplikacje i przetestować poniższe elementy:

- Działanie aplikacji w różnych przeglądarkach
- Tworzenie wielu nowych kont użytkownika
- Próby logowania się na istniejące konta używając tych samych nazw użytkownika i/lub haseł
- Wyświetlanie elementów graficznych na różnych rozdzielczościach ekranu
- Otrzymanie wynagrodzenia za wykonane zadanie
- Wyświetlanie napisów informujących
- Omijanie przeszkód przez potwory

Elementy te nie zostały wybrane przypadkowo. Są to funkcjonalności najbardziej narażone na nieprawidłowe działanie i występowanie błędów.

7.2. Wnioski

Podczas testowania manualnego aplikacji webowej, na różnych etapach, była przeprowadzana w szerokim zakresie analiza, skupiając się na ocenie danych funkcjonalności i użyteczności.

Działanie aplikacji zostało przeprowadzone w najpopularniejszych przeglądarkach jakimi są Microsoft Edge, Mozilla Firefox[25], Google Chrome, Brave[26] i Opera. Testowanie wykazało problem w przeglądarce Mozilla Firefox. Próba

[25] https://www.mozilla.org/pl/firefox/
[26] https://brave.com/pl/

uruchomienia gry kończyła się wyświetleniem czarnego ekranu w oknie przeglądarki. Konsola wykazała błąd podczas deklaracji importu w niektórych plikach JavaScript. Podczas analizy przypadku okazało się, że Mozilla Firefox nie obsługuje deklaracji importu (import assertions).

Uruchamianie gry na ekranach o różnej rozdzielczości i proporcjach wykazało, że elementy graficzne poprawnie wyświetlają się na ekranach panoramicznych (16:9 lub 16:10) o rozdzielczości minimalnej 1280x720. Zaburzenie tej proporcji skutkowało utratą ostrości paru wyświetlanych elementów i w niektórych przypadkach „wyjściem" tekstu poza ustalony obszar.

Pozostałe scenariusze nie wykazały żadnych nieprawidłowości. Sama realizacja zaś przebiegła płynnie i skrupulatnie.

PODSUMOWANIE

Praca inżynierska skupiła się na kompleksowym projekcie i implementacji gry przeglądarkowej, wykorzystującej języki webowe. Celem projektu było stworzenie interaktywnej i angażującej aplikacji, dostępnej dla użytkowników bez konieczności instalacji dodatkowego oprogramowania.

W pierwszej części pracy określono cele projektu oraz dokonano analizy wymagań funkcjonalnych i niefunkcjonalnych gry, uwzględniając zarówno aspekty rozgrywki, jak i interfejsu użytkownika. Zidentyfikowane zostały kluczowe elementy, m.in. mechanika gry, system rejestracji użytkownika, czy dostępność aplikacji.

Następnie zaprezentowano projekt gry. Nakreślono koncepcje gry i pokazano szkic wstępny elementów interfejsu użytkownika. Zostały również przedstawione narzędzia pracy, które są wykorzystane podczas realizacji projektu.

W trakcie implementacji skupiono się na efektywnym wykorzystaniu serwera Expres i systemu do zarządzania bazą danych. Naturalnie, uwzględniono również trzon aplikacji – implementacje samej gry. Przedstawiono kluczowe funkcje, takie jak system logowania, interakcja gracza z otoczeniem i interfejsem, czy animacje ruchu obiektów.

Po zakończeniu procesu implementacyjnego i pokazaniu funkcjonalności przeprowadzono testy manualne, celem weryfikacji poprawności działania aplikacji.

W rezultacie uzyskano funkcjonalną i szeroko dostępną grę przeglądarkową, spełniającą założenia projektowe. Praca stanowi cenne źródło wiedzy dla osób zainteresowanych projektowaniem i implementacją aplikacji webowych, zwłaszcza w kontekście gier przeglądarkowych.

Wnioski z przeprowadzonych testów i analiz mogą być wykorzystane jako punkt wyjścia do dalszych prac na rozwijaniem oraz doskonaleniem aplikacji w przyszłości. Oprócz tego projekt może zostać jeszcze bardziej zoptymalizowany pod kątem wydajności. Jako że rozgrywka polega na przejściu prologu, projekt staje się idealnym wstępem do rozwinięcia fabuły, dodania nowych lokacji, przeciwników, animacji, czy też systemu handlu. Projektowanie gier ma to do siebie, że twórcę ogranicza tylko i wyłącznie jego kreatywność.

SPIS LITERATURY

1. **Duckett Jon.** *JavaScript and JQuery: Interactive Fron-End Web Development.* Indianapolis : John Wiley & Sons, Inc., 2014. 978-83-8322-755-9.

2. **Forta Ben.** *SQL in 10 Minutes a Day, Sams Teach Yourself (5th Edition).* Oak Park : Pearson Education, Inc., 2020. 978-83-283-9490-2.

3. **Cecco Raffaele.** *Supercharged JavaScript Graphics. with HTML5 canvas, jQuery, and More.* Sebastopol : O'Reilly Media, 2011. 978-1449393632.

4. **Rowell Eric.** *HTML5 Canvas Cookbook.* Birmingham : Packt Publishing, 2011. 978-83-246-5075-0.

5. **Arsever Selim.** *jQuery Game Development Essentials.* Birmingham : Packt Publishing, 2013. 978-83-246-8608-7.

6. **Svekis Laurence Lars, van Putten Maaike i Percival Rob.** *JavaScript from Beginner to Professional: Learn JavaScript quickly by building fun, interactive, and dynamic web apps, games, and pages.* Birmingham : Packt Publishing, 2021. 978-83-8322-197-7.

ŹRÓDŁA INTERNETOWE

7. ***Mixkit.*** [Online] https://mixkit.co/free-sound-effects/click/.

8. **alkakrab.** *itch.io.* [Online] https://alkakrab.itch.io/fantasy-rpg-soundtrack-music.

9. **Leohpaz.** *itch.io.* [Online] https://leohpaz.itch.io/minifantasy-dungeon-sfx-pack.

10. **Ryan.** *Codepen.* [Online] https://codepen.io/ryandsouza13/pen/yEBJQV.

11. **Kalinowski Wojciech.** *DaFont.* [Online] https://www.dafont.com/medieval-sharp.font.

12. **The Game Smith.** *itch.io.* [Online] https://smallmak.itch.io/animated-axe-cursor.

SPIS TABEL

SPIS RYSUNKÓW

SPIS BLOKÓW KODU

Katowice, dnia ……

Imię, nazwisko: ……

Kierunek: Informatyka

Nr albumu: ……...

OŚWIADCZENIE

Oświadczam, że przedłożoną inżynierską pracę dyplomową wykonaną w Wyższej Szkole Technologii Informatycznych w Katowicach na kierunku Informatyka opracowała(e)m samodzielnie i nie naruszyła(e)m praw autorskich innych osób, które chronione są przepisami ustawy z dnia 4 lutego 1994 roku o prawie autorskim i prawach pokrewnych.
Jednocześnie oświadczam, że przedłożona praca nie była nigdzie wcześniej publikowana i oceniana.

Podpis studenta……

Katowice, dnia ……

Imię, nazwisko: ……

Kierunek: Informatyka

Nr albumu: ……...

OŚWIADCZENIE

Oświadczam, że praca inżynierska p.t.:

”.....
..... ."

zapisana na załączonym nośniku elektronicznym, jest zgodna z treścią zawartą w wydrukowanej wersji pracy przedstawionej do oceny.

Oświadczam, że przedłożona praca dyplomowa jest zatwierdzona przez promotora.

Podpis studenta……