"ClashBet" | Projekt w ramach przedmiotu BD2 | Zespół nr. 5

Backend oraz API

Część backendowa projektu wraz z API zostały stworzone w Typescript, przy użyciu NestJS. W tej części zrealizowane zostały wszelkie interakcje bazy danych z serwerem oraz frontendem w celu wyświetlania odpowiednich danych na stronach frontendowych, oraz zapisywania do bazy odpowiednich informacji zależnych od wykonywanych przez użytkownika czynności. Dokładniej wymienić można:

- system rejestrowania użytkowników i zapisywania danych z formularza rejestracji w bazie danych do tabeli Users
- system logowania użytkowników, porównujący dane z formularza do danych w bazie
- Session-Based Authentication, służący do zachowywania aktywnej sesji w ciasteczkach użytkownika
- możliwość wylogowania się z konta, deaktywując i usuwając wcześniej aktywne ciasteczko użytkownika
- aktualizowanie na bieżąco stanu konta zalogowanego użytkownika, korzystający z danych w tabeli Wallets
- wyświetlanie podstawowych informacji o profilu, korzystające z zapisanych danych w tabeli Users
- możliwość zmiany podstawowych informacji o profilu, aktualizująca dane w tabeli Users
- możliwość wpłaty lub wypłaty środków, zapisujący dane o transkacji w tabeli Transactions oraz BalanceState, aktualizująca Wallets należący do deponenta / wypłacającego
- zapisywanie każdej zmiany stanu środków użytkownika dzięki Triggerowi dla tabeli BalanceState
- zapisywanie meczów w tabeli Matches z wykorzystaniem Triggera odpowiadającym za nadawanie kolejnych w ciągu 1, 2, 3... identyfikatorów MatchID
- aktualizowanie PlayerStatistics w zależności od rezultatu zakończonego meczu 1 na 1, obliczanie WinRate na bazie ilości wygranych, przegranych, oraz remisów.
- aktualizowanie stanu środków użytkownika (Wallets) w zależności od rezultatu meczu
- pobieranie danych z tabeli Matches, PlayerStatistics (również Users), BalanceState w celu wyświetlenia odpowiednio historii gier dla użytkownika, wykresu WinRate od czasu oraz tablicy wyników, i wykresu stanu środków na koncie od czasu.
- organizację danych do wykresu WinRate od czasu, obliczającą stan WinRate w każdym punkcie wykresu według danych z tabeli Matches
- organizację danych do wykresu Balance od czasu, biorąca pod uwagę wszystkie zmiany wynikające z danych w tabeli BalanceState
- system wyszukiwania gry 1 na 1 zależny od wybranej gry oraz opłaty EntryFee, bazujący na połączeniu WebSocket w celu aktualizowania stanu wyszukiwania meczu na bieżąco
- system parowania graczy
- backendowa logika gry w Papier, Kamień, Nożyce, opierająca się na połączeniu WebSocket pomiędzy sparowanymi graczami
- backendowa logika gry w Sapera na czas, opierająca się na połączeniu WebSocket pomiędzy sparowanymi graczami
- własny adapter WebSocket, stworzony do współpracy z implementacją Session-Based Authentication
- strukturę projektu, pozwalającą na modularność oraz na mało skomplikowane dodawanie nowych gier do strony

Krótki opis struktury projektu (backend)

NestJS to framework stworzony do budowania skalowalnych aplikacji serwerowych, wykorzystujący TypeScript, bądź JavaScript. Aplikacja w NestJS podzielona jest na moduły, z których każdy enkapsuluje powiązaną z nimi logikę za pomocą kontrolerów oraz serwisów. Kontrolery odpowiadają za obsługę przychodzących żądań HTTP i definiowanie endpointów API, podczas gdy serwisy opisują logikę biznesową i są wstrzykiwane do kontrolerów, lub też innych serwisów. W kontrolerach wykorzystywane są dekoratory @Get(), @Post() oraz inne. Dekoratory te umożliwiają deklaratywne definiowanie metadanych dla klas oraz należącej do danej klasy metod (np. @Get(), @Post(), @Body()). Wykorzystanie NestJS pozwala więc na jasny podział odpowiedzialności, oraz na stosowanie modularnego podejścia do tworzenia aplikacji serwerowej.

W folderze ./apps/api widnieją pliki konfiguracyjne, wygenerowane przez Nest oraz npm. Pliki w folderze ./apps/api/src to główne pliki wygenerowane przez NestJS. Obie te grupy plików służą do włączania serwera lokalnego na porcie 3001 (localhost:3001). Z kolei każdy podfolder folderu ./apps/api/src zawiera już faktyczne moduły programu, podzielone wedle spełnianych przez nie zadań.

Podfolder **misc** zawiera definicję przychodzącej wiadomości HTTP (używane w grach) oraz własny adapter WebSocket, pozwalający na zachowywanie połączenia pomiędzy dwoma, zalogowanymi graczami. Podfolder **services** przetrzymuje wszystkie moduły, które nie organizują tabel bezpośrednio związanych z profilem użytkownika (Users, Wallets, PlayerStatistics). Zawiera on:

- moduł do autoryzacji (**auth**)
- moduł do odczytywania z tabeli BalanceState (**balance-state**)
- podfolder **games**, zawierający moduły związane z tabelami GameMoves i GameTypes
- specjalny interfejs do parsowania danych związanych z grą w Sapera
- bramki WebSocket do gier, organizujące komunikację gracz-serwer-gracz

Wewnątrz podfolderu **users** znajdują się moduły związane z tabelami Users, PlayerStatistics oraz Wallets. Celem istnienia takiego podfolderu jest wyodrębnienie organizacji profilu oraz danych o profilu użytkownika od danych analitycznych, oraz metod serwerowych.

Do realizacji interakcji z bazą danych wykorzystano TypeORM. Jest to narzędzie do mapowania encji na obiekty innych języków programowania. Wszelkie insercje, aktualizacje i usunięcia organizowane są na podstawie **repozytoriów** TypeORM, które posiadają możliwość realizowania takich operacji poprzez transakcje bazodanowe.

Fragmenty części serwerowej

Session-Based Authentication

Do zachowywania danych o logowaniu, i uniknięcia potrzeby logowania przy każdym wejściu na stronę, zaimplementowano funkcję Session-Based Authentication. Logowanie bądź rejestracja na stronie aktywuje ciasteczko, które później weryfikowane jest poprzez serwer przy pomocy modułu **auth**. Realizacja tego typu identyfikacji polegała na wykorzystaniu "strażników" identyfikacji (AuthenticatedGuard, LocalAuthGuard) za

pomocą specjalnego dekoratora @UseGuards(), który przed wykonaniem metody kontrolera, woła metody odpowiedniego strażnika.

Matchmaking, oraz połączenie Websocket

Wyszukiwanie meczu "1 na 1" działa na bazie "lobby" przechowywanych w pamięci serwera. Gracz wpisuje w formularz stawkę za którą chce zagrać, oraz zakres stawki, który pozwala na rozszerzenie kontekstu wyszukiwania. Wewnątrz podmodułu **matchmaking** modułu **matches** zdefiniowana jest logika wyszukiwania i filtrowania pokoi w zależności od wybranej gry, oraz dobranej stawki. Jeżeli po filtracji lista dostępnych pokoi jest pusta, automatycznie tworzony jest nowy pokój, a osoba szukająca meczu staje się **hostem** tego nowego lobby. **Zakres stawki nie ma wpływu na hostowanie lobby** - oznacza to, że pokój jest tworzony z wcześniej ustaloną przez hosta stawką, a zakres ma wpływ tylko i wyłącznie na wyszukiwanie meczu.

Do matchmakingu wykorzystałem połączenie za pomocą WebSocket. Było to wymagane, ponieważ połączenie HTTP nie pozwalało mi na komunikowanie obu graczy o znalezieniu pokoju. HTTP działa na żądaniach, nie an aktywnym połączeniu. Skutkowało to tym, że host wysyłał żądanie o znalezienie meczu, druga osoba ten mecz znajdowała, ale po znalezieniu meczu jedynym dostępnym odbiorcą był drugi nadawca żądania, przez co host nie był informowany o sparowaniu. Połączenie działa więc w trakcie wyszukiwania meczu, i zrywane jest od razu po znalezieniu. Dalsze połączenie realizowane jest już nowej sesji WebSocket - w grach 1 na 1.

Logika gier 1 na 1 (backend)

Logika gier 1 na 1 po stronie backendu opiera się na definicji odbieranych przez WebSocket żądań od frontendu. Te żądania róznią się w zależności od gry, więc też każda gra ma stworzony własny **gateway**. Każdy gateway posiada swój własny adres, co pozwala na oddzielenie połączeń dla np. Kamień, Papier, Nożyce od innych gier.

Punkty końcowe API

Punkty końcowe API zdefiniowane są w kontrolerach posiadających je modułów. Wewnątrz frontendu opisano adres serwera jako /api do prostszego wykorzystywania endpointów:

```
module.exports = {
    async rewrites() {
        return [
            {
                source: "/api/:path*",
                destination: "http://localhost:3001/:path*",
            },
        ];
    },
};
```

Do każdego kontrolera można przypisać inną ścieżkę, na przykład:

```
@Controller('auth')
export class AuthController {...}
```

Wówczas chcąc wykorzystać metodę kontrolera, po stronie frontendu wykonujemy żądanie do "/api/auth/<ścieżka_metody>", gdzie ścieżka_metody działa podobnie jak powyżej:

```
@Post('register')
@HttpCode(HttpStatus.CREATED)
async register(
  @Body() userData: { username: string; email: string; password: string },
  @Request() req,
)
```

Wykorzystanie po stronie frontendowej *register* wymaga wtedy żądania do "/api/auth/register".

Fragmenty części bazodanowej

Profil użytkownika - rejestracja, logowanie, i modyfikacja

Poprawna rejestracja użytkownika kończy się wstawieniem nowego rzędu do tabeli Users:

```
CREATE TABLE Users (
    UserID INTEGER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    Username VARCHAR(255) NOT NULL,
    Email VARCHAR(255) UNIQUE NOT NULL,
    UserFullName VARCHAR(50) DEFAULT '',
    UserBio VARCHAR(255) DEFAULT '',
    PasswordHash VARCHAR(255) NOT NULL,
    CreatedAt TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    UpdatedAt TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    UserRole VARCHAR(100) DEFAULT 'user'
);
```

Proces logowania wyszukuje wewnątrz tej tabeli użytkownika o danym mailu. Jeżeli w bazie danych nie ma tego adresu, logowanie się nie powodzi. Jeżeli jest, porównywane jest zaszyfrowane hasło wpisane przez użytkownika z przetrzymywanym w bazie danych zaszyfrowanym faktycznym hasłem użytkownika.

Użytkownik nie posiada od momentu rejestracji ani opisu, ani pełnego imienia i nazwiska. Te kolumny można wypełnić na stronie, wpisując do formularza dane, i naciskając przycisk "Save". Po przyciśnięciu, na początek Session-Based Authentication identyfikuje użytkownika, zbiera UserID, w bazie danych wyszukiwany jest według UserID odpowiedni rząd, i aktualizowane są kolumny UserBio oraz UserFullName.

Stan konta użytkownika, oraz jego aktualizacja.

Stan konta realizowany jest w tabeli Wallets:

```
CREATE TABLE Wallets (
    WalletID INTEGER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    UserID INTEGER NOT NULL,
    Balance DECIMAL(6, 2) DEFAULT 0,
    PendingBalance DECIMAL(6, 2) DEFAULT 0,
    CreatedAt TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    UpdatedAt TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (UserID) REFERENCES Users(UserID) ON DELETE CASCADE
);
```

Stan konta pobierany jest za pomocą UserID z kolumny Balance. Aktualizacje stanu konta przebiegają w trzech scenariuszach: przy depozytach i wypłatach, przy rozpoczęciu, oraz przy zakończeniu meczu.

Organizacja meczów oraz statystyk z punktu widzenia bazy Oracle

Tabela Matches przechowuje mecze w trakcie, jak i mecze zakończone.

```
CREATE TABLE Matches (
    MatchID INTEGER PRIMARY KEY,
    GameTypeID INTEGER NOT NULL,
    PlayerOneID INTEGER NOT NULL,
    PlayerTwoID INTEGER NOT NULL,
    EntryFee NUMBER(10,2),
    MatchState VARCHAR(40),
    WinnerID INTEGER,
    StartTime TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    EndTime TIMESTAMP,
    LobbyUUID VARCHAR2(36) UNIQUE NOT NULL,
    Score VARCHAR(500),
    FOREIGN KEY (GameTypeID) REFERENCES GameTypes(GameTypeID),
    FOREIGN KEY (PlayerOneID) REFERENCES Users(UserID),
    FOREIGN KEY (PlayerTwoID) REFERENCES Users(UserID),
    FOREIGN KEY (WinnerID) REFERENCES Users(UserID)
);
```

Przy rozpoczęciu wykonywana jest insercja rzędu z pustymi kolumnami WinnerID, EndTime oraz Score, a po zakończeniu, mecz z tym samym uuid jest aktualizowany, a puste miejsca są wypełniane danymi.

Frontend

Aplikacja internetowa, znajdująca w (`./apps/web`) zainicjowana została przy użyciu narzędzia do tworzenia szkieletu aplikacji `create-next-app`. Stylizacja jest zarządzana przez Tailwind CSS.

Podstawowa struktura i routing (`src/app`):

Serce aplikacji znajduje się w katalogu `src/app`. Konkretne ścieżki aplikacji, takie jak `/dashboard`, `/game`, `/join` i `/login`, są wyraźnie wyznaczone przez odpowiadające im foldery w `src/app`. Każdy z tych katalogów zawiera plik `page.tsx` odpowiedzialny za renderowanie interfejsu użytkownika dla danego segmentu.

Główny katalog `src/app` zawiera podstawowe pliki:

- `layout.tsx`: Definiuje główny layout (układ), otaczający wszystkie strony.
- `page.tsx`: Służy jako punkt wejściowy dla strony głównej aplikacji (ścieżka `/`).
- `globals.css`: Plik przeznaczony na style globalne.
- `favicon.ico`: Ikona aplikacji wyświetlana w karcie przeglądarki.
- `page.module.css`: Moduły CSS do stylizacji komponentów w `page.tsx`.

Konfiguracja i narzędzia:

Projekt jest skonfigurowany ze standardowymi dla framework'a plikami:

- **Specyficzne dla Next.js**:
  - `.next/`: Katalog automatycznie generowany przez Next.js, zawierający wyniki kompilacji i pamięć podręczną.
  - `next.config.ts`: Główny plik konfiguracyjny do dostosowywania zachowania Next.js.
  - `next-env.d.ts`: Plik deklaracji TypeScript zapewniający rozpoznawanie typów Next.js.
- **Stylizacja**:
  - `postcss.config.mjs`: Konfiguruje PostCSS - przetwarza dyrektywy Tailwind i transformacje CSS.
- **TypeScript i Linting**:
  - `tsconfig.json`: Konfiguruje opcje kompilatora TypeScript.
  - `eslint.config.mjs`: Definiuje reguły i ustawienia dla ESLint.
- **Zarządzanie projektem i zależnościami**:
  - `package.json`: Zawiera metadane projektu, zależności i skrypty.
  - `node_modules/`: Zawiera wszystkie zainstalowane pakiety npm (pnpm).

Przepływ i struktura rozgrywki: `/game/[gameId]`

Rozgrywki zorganizowane są poprzez dynamiczny system routingu skoncentrowany wokół ścieżki `/game/[gameId]/`.

1. **Dołączanie do pokoju i przekierowanie:**
   - Użytkownicy najpierw dołączają do pokoju gry poprzez interfejs znajdujący się w `/dashboard/waitroom`.
   - Po pomyślnym dołączeniu lub utworzeniu pokoju, użytkownikowi przypisywany jest unikalny **`gameId`**.
   - Aplikacja następnie przekierowuje użytkownika pod adres `/game/[gameId]`.
2. **Strona hostująca sesję gry (`game/[gameId]/page.tsx`):** Plik `page.tsx` znajdujący się bezpośrednio w katalogu `[gameId]` jest głównym komponentem renderowanym dla tej konkretnej sesji gry.
   - Ta strona działa jako współdzielona powłoka sesji gry.

- Po załadowaniu komponent `page.tsx` użyje `gameId` do pobrania danych o konkretnej sesji gry z backendu.
- Te pobrane dane będą zawierać:
  - **Typ gry** rozgrywanej w tym lobby ("battleships", "minesweeper", "rock-paper-scissors").
  - Aktualny **stan gry** dla tego lobby.
  - Informacje o innych graczach w lobby.
- Na podstawie pobranego `gameType`, ta strona następnie dynamicznie załaduje i wyrenderuje odpowiednie konkretne komponenty gry.
- Stylizacja współdzielonej powłoki jest obsługiwana przez `GamePage.module.css`.

3. **Ładowanie i renderowanie konkretnej gry (`game/games/...`):**

- Gdy komponent `game/[gameId]/page.tsx` pobierze dane lobby i zidentyfikuje `gameType`, dynamicznie importuje i renderuje główny komponent dla konkretnego typu gry z podkatalogu `game/games/`.
- Ten "konkretny komponent gry" (np. `BattleshipsGame`) jest następnie wstrzykiwany do wyznaczonego kontenera w układzie `game/[gameId]/page.tsx`.