POLITECHNIKA POZNAŃSKA

Wydział Informatyki i Telekomunikacji
Instytut Informatyki

Praca inżynierska

Stworzenie gry bazującej na zasadach szachów wykorzystującej sztuczną inteligencję - Aplikacja sieciowa

Autorzy
Cieślak Patryk, 140693
Mazurowski Wojciech, 140746
Stelmasiak Mateusz, 140783
Zarębski Karol, 140809

Promotor
dr inż. Anna Grocholewska-Czuryło

Poznań, 2022

„Poufna rozmowa bez słów; przejmująca dreszczykiem
aktywność przy stanie pełnej bezwładności, triumf i klęska,
nadzieja i zwątpienie, życie i śmierć – i to wszystko na 64 polach;
połączenie poezji z nauką i rywalizacją – oto czym są szachy”

~ Ksawery Tartakower

Abstract

Creating a game based on rules of chess that uses Artificial Intelligence - Web Application

This paper concerns the creation of an internet application allowing users to partake in games based on the rules of chess. The aforementioned application was created as a group engineering thesis project. The software allows gameplay either between two players or between a player and artificial intelligence. During the game, players can communicate with each other via text chat. A logged in user can view their statistics, history of played games, edit their account, etc. In order to use all the functionalities of the application, a user must create and confirm an account.

The work discusses the goals and motivation of the team, and describes the theoretical foundations necessary to understand the rules of the games. The theoretical plan of the application, technologies and tools used were presented. The implementation of individual functionalities is described in detail and the tests that have been performed are presented. The last chapter of the work is devoted to a summary of encountered problems and prospects for further development of the application.

Streszczenie

Stworzenie gry bazującej na zasadach szachów wykorzystującej sztuczną inteligencję - Aplikacja sieciowa

W ramach dyplomowej pracy zespołowej stworzono aplikację internetową, umożliwiającą rozgrywki szachowe. Udostępnia ona rozgrywki zarówno między graczami jak i między graczem, a sztuczną inteligencją. Podczas rozgrywki gracze mogą komunikować się ze sobą za pośrednictwem czatu tekstowego. Zalogowany użytkownik ma możliwość przeglądania swoich statystyk, historii rozegranych partii szachowych, edytować konto, itp. Aby korzystać ze wszystkich funkcjonalności aplikacji, należy utworzyć oraz potwierdzić konto gracza.

W pracy omówiono cele oraz motywację do jej stworzenia, opisano podstawy teoretyczne konieczne do zrozumienia zasad obowiązujących w rozgrywce. Przedstawiono koncepcję aplikacji, użyte technologie oraz narzędzia. Obszernie opisano implementację poszczególnych funkcjonalności oraz przedstawiono testy jakie zostały wykonane. Ostatni z rozdziałów pracy poświęcony został na podsumowanie, w którym umieszczono napotkane problemy oraz perspektywy dalszego rozwoju aplikacji.

Spis treści

Wykaz Stosowanych Akronimów

2FA	ang. *two-factor authentication*; weryfikacja dwuetapowa
AN	ang. *algebric notation*; szachowa notacja algebraiczna
API	ang. *application programming interface*; interfejs programowania aplikacji
CAPTCHA	ang. *completely automated public turing test to tell computers and humans apart;* całkowicie zautomatyzowany publiczny test turinga pozwalający odróżnić komputery od ludzi
CORS	ang. *cross-origin resource sharing*
FEN	ang. *Forsyth–Edwards notation;* notacja Forsytha-Edwardsa
FIDE	fr. *Fédération Internationale des Échecs;* Międzynarodowa Federacja Szachowa
HTML	ang. *HyperText Markup Language*
HTTP	ang. *hypertext transfer protocol*
ID	ang. *identifier*
IPv4	ang. *internet protocol version 4*
JSON	ang. *javascript object notation*
MD	ang. *message digest*
OTP	ang. *one-time password*; hasło jednorazowe
PERFT	ang. *performance test, move path enumeration*
PGN	ang. *portable game notation*
SHA	ang. *secure hash*
SID	ang. *socket identifier*
SMTP	ang. *simple mail transfer protocol*
SQL	ang. *structured query language*
SSL	ang. *secure socket layer*

TCP	ang. *transmission control protocol*
UML	ang. *unified modeling language*
URL	ang. *uniform resource locator*
QR	ang. *quick response*
VPS	ang. *virtual private server*
WWW	ang. *world wide web*
XSS	ang. *cross-site scripting*

1. WSTĘP

Celem pracy jest opracowanie aplikacji internetowej umożliwiającej użytkownikom rozgrywkę w szachy oraz gry oparte na ich zasadach, przy pomocy przeglądarki internetowej. Głównym motywem podjęcia pracy była chęć stworzenia autorskiego trybu gry, opisanego w dalszej części pracy (patrz sekcja 2.2). Dodatkową motywacją był fakt, że jedyna podobna istniejąca gra została wycofana ze strony chess.com bez informacji czy kiedykolwiek zostanie przywrócona.

Struktura pracy jest następująca. Pierwszy rozdział to wstęp, opisujący założenia i cele pracy, istniejące na rynku rozwiązania oraz podział pracy między członków zespołu. W rozdziale 2 przedstawiono podstawy teoretyczne rozwiązania projektowego, w tym zasady rozgrywki oraz wykorzystywane sposoby reprezentacji szachownicy i ruchów na niej wykonywanych. Rozdział 3 jest poświęcony przybliżeniu ogólnej koncepcji rozwiązania projektowego oraz technologii jakie zostały w nim użyte. Rozdział 4 zawiera szczegółowe omówienie struktur danych oraz sposobów implementacji konkretnych funkcjonalności stworzonego systemu. Rozdział 5 zawiera opis przeprowadzonych testów. Rozdział 6 natomiast stanowi podsumowanie pracy.

1.1 Cel i zakres pracy

Głównym celem aplikacji jest umożliwienie użytkownikom rozgrywki szachowej w trzech trybach. Pierwszym z trybów jest rozgrywka według standardowych zasad szachowych (patrz sekcja 2.1). W pozostałych trybach rozgrywka odbywa się według zasad bazujących na zasadach szachowych opisanych dokładniej w dalszej części pracy (patrz sekcja 2.2 i sekcja 2.3).

Istnieje więc konieczność implementacji możliwości rejestracji konta oraz logowania. Po zalogowaniu użytkownik będzie mógł nie tylko rozpocząć rozgrywkę, ale również przeglądać swoje statystyki, na które będą składały się punkty ELO (patrz sekcja 2.6), liczbę rozegranych, wygranych oraz zremisowanych gier. Dodatkowo w statystykach pojawi się historia rozegranych gier, zawierająca stan punktów ELO przed rozpoczęciem gry, datę gry oraz jej wynik.

Użytkownik będzie mógł zarządzać swoimi danymi. Aplikacja będzie pozwalała na zmianę hasła, adresu email, ustawienie weryfikacji dwuetapowej oraz usunięcie wszystkich danych o użytkowniku.

W trakcie rozgrywki aplikacja będzie wyznaczała wszystkie możliwe ruchy danego gracza, które będą wyświetlane na planszy w ramach podpowiedzi. Aplikacja umożliwia przeprowadzenie rozgrywki przeciwko sztucznej inteligencji. Próba wykonania niedozwolonego ruchu zakończy się niepowodzeniem, a w ostateczności próba takiej czynności zostanie cofnięta przez serwer. Dodatkowo aplikacja będzie obsługiwać zegar szachowy, synchronizowany po stronie serwera, w celu zapewnienia dynamiki rozgrywki, w trakcie której będzie też możliwe poddanie się lub zaproponowanie remisu w dowolnym momencie.

1.2 Przegląd podobnych rozwiązań

Obecnie istnieje wiele aplikacji internetowych umożliwiających grę w szachy oferujących funkcjonalności podobne do aplikacji będącej tematem tej pracy.

Jeżeli chodzi jednak o tryb *ChessDefender*, jedyna podobna aplikacja została bezterminowo wyłączona z użytku. W ostatnim czasie szachy stają się coraz bardziej popularną grą, przez co można zaobserwować rozwój stron internetowych pozwalających rozegranie partii szachowych.

1.2.1 Portal lichess.org

Lichess.org jest jednym z najbardziej popularnych na świecie portali umożliwiającym grę w szachy. Według informacji udostępnionych przez sam portal, miesięcznie jest rozgrywanych blisko 90 milionów[1] takich rozgrywek.

Użytkownik na portalu, ma możliwość rozegrania partii szachowych zarówno z przeciwnikami na podobnym poziomie z całego świata jak i ze znajomymi. Strona ta wprowadza też system zwany arenami, w których użytkownik rozgrywa kilka partii w danym okresie czasu, aby pod koniec zsumować wyniki partii i ustalić zwycięzcę areny.

Aplikacja pozwala na przeprowadzanie analiz gier szachowych za pomocą silnika szachowego *StockFish*. Analizowane partie można wczytywać za pomocą notacji FEN (patrz sekcja 2.5), PGN (ang. *portable game notation*) lub wprowadzać ręcznie.

Użytkownik może także wykonywać zadania szachowe, które są generowane na podstawie wcześniejszych gier wszystkich użytkowników. W tym trybie użytkownik posiada punkty rankingowe, na podstawie których dostosowywane są trudności zadań.

Na portalu gracz może przeglądać swoje statystyki jak i statystyki innych użytkowników. Składają się one z historii rozgrywek z możliwością ich odtworzenia, rankingów szachowych różnych trybów rozgrywki, wykresów przedstawiających punkty rankingowe w zależności od czasu, łącznego czas gry, liczby wygranych oraz przegranych rozgrywek, itp.

Aplikacja wprowadza możliwość oglądania partii innych graczy, a także udostępnia fora społecznościowe do dyskusji na tematy szachowe.

Niewątpliwym atutem portalu, jest to, że jest on w pełni darmowy. Użytkownik ma dostęp do wszystkich funkcjonalności aplikacji całkowicie za darmo. Jest to jedna z nielicznych stron, która zdecydowała się na takie podejście.

1.2.2 Portal chess.com

Portal chess.com jest koleną, jedną z najczęściej używanych aplikacji internetowych umożliwiających rozgrywanie partii szachowych. Według informacji udostępnionych przez portal, już w 2014 roku zostało rozegranych ponad miliard gier.

Portal umożliwia użytkownikowi rozegranie partii szachowych, nie tylko w trybie klasycznym ale też w innych, różnorodnych wariantach. Najbardziej popularnymi trybami, poza szachami klasycznymi są:

[1] https://database.lichess.org/

- “4 Player Chess” - umożliwia grę w szachy na cztery osoby,
- “Crazyhouse” - umożliwia wstawianie zbitych bierek z powrotem na planszę,
- “Fog of war” - tryb, w którym widoczne są tylko te pola, które “widzą” bierki danego gracza,
- “Atomic” - tryb, w którym po zbiciu bierki, wszystkie bierki wokół bitej również znikają z planszy.

W aplikacji istniał tryb zwany “Automate”, który funkcjonalnością przypominał *ChessDefender*. Dla wielu graczy (patrz sekcja 2.2). W tym wariancie, użytkownicy naprzemiennie wystawiali figury za punkty przypisane na początku gry. Po wydaniu wszystkich punktów, można było wystawić króla, po czym rozgrywkę na tak ustawionych pozycjach przeprowadzał komputer. Użytkownik, po ustawieniu pozycji, nie miał możliwości żadnej dalszej ingerencji w rozgrywkę. Według wielu użytkowników takie rozwiązanie sprawiało, że rozgrywka stawała się nudna, co doprowadziło do coraz mniejszej popularności wariantu. Ostatecznie tryb ten przestał być dostępny na stronie.

Portal udostępnia także możliwość rozwiązywania zadań szachowych oraz analizowania rozgrywek, które można wczytywać za pomocą FEN, PGN czy wprowadzać ręcznie. Użytkownicy portalu mają dostęp do forów społecznościowych.

Dużą wadą aplikacji jest to, że większość funkcjonalności jest dostępna tylko dla użytkowników premium. Co prawda, nie ma limitu przeprowadzanych rozgrywek szachowych, jednak dziennie można zrobić tylko trzy analizy, wykonać tylko trzy zadania szachowe, itp.

1.2.3 Podsumowanie

Podmioty opisane w rozdziale 1.2 posiadają wiele podobieństw do aplikacji implementowanej w ramach rozwiązania projektowego. Podstawową i główną różnicą, jest tryb rozgrywki - *ChessDefender*, zarówno w trybie wieloosobowym jak i jednoosobowym. Nie jest on bowiem dostępny na żadnej ze znalezionych stron, nawet w przybliżonej formie. Jedyny tryb, najbliższy formie *ChessDefender* - “Automate” nie był rozwijany i został porzucony przez brak popularności.

1.3 Podział zadań

W tabeli 1 przedstawiono podział prac między członków zespołu projektowego.

Tabela 1. Podział pracy między członków zespołu

Autor	**Przydział zadań**
Patryk Cieślak	Glicko2, Wysyłanie wiadomości za pośrednictwem poczty elektronicznej, Mechanizm generowania kodów QR podczas weryfikacji dwuetapowej, Wdrożenie aplikacji do środowiska produkcyjnego

Wojciech Mazurowski	Silnik szachowy po stronie klienta, graficzny interfejs szachownicy, zaprojektowanie zasad trybu ChessDefender w trybie wieloosobowym i jednoosobowym, implementacja trybu ChessDefender, synchronizacja silnika szachowego z trybem ChessDefender, testy trybów gry.
Mateusz Stelmasiak	Implementacja aplikacji klienckiej, projekt graficzny aplikacji klienckiej, implementacja systemu doboru przeciwnika, implementacja przeprowadzania rozgrywki po stronie serwera, implementację komunikacji w trybie asynchronicznym, implementację komunikacji w trybie synchronicznym, mechanizm uwierzytelniania użytkownika, implementacja podglądu historii rozgrywek użytkownika, implementacja podglądu statystyk użytkownika.
Karol Zarębski	Mechanizm weryfikacji dwuetapowej, mechanizm zarządzania kontem użytkownika, usuwanie danych użytkownika z serwera, możliwość resetowania aktualnego hasła logowania, aktywowanie nowo utworzonego konta, implementacja bazy danych, konteneryzacja bazy danych, testy mechanizmów logowania, potwierdzania konta, itp.

1.3 Wymagania funkcjonalne

W projekcie wyróżniono niżej wymienione wymagania funkcjonalne. Wśród głównych aktorów można wymienić: niezalogowanego użytkownika, zalogowanego użytkownika, użytkownika w grze, system oraz administratora. Poniżej przedstawione są wymagania funkcjonalne.

- Administrator uruchamia aplikację serwera.
- Administrator wyłącza aplikację serwera.
- Niezalogowany użytkownik rejestruje się na stronie internetowej aplikacji oraz aktywuje swoje konto.
- Niezalogowany użytkownik loguje się na stronie internetowej aplikacji.
- Zalogowany użytkownik wylogowuje się ze strony internetowej aplikacji.
- Zalogowany użytkownik przegląda statystyki swoich gier.
- Zalogowany użytkownik przegląda historię swoich gier.
- Zalogowany użytkownik zarządza swoim kontem

- Zalogowany użytkownik dołącza do kolejki oczekiwania na grę w szachy.
- Zalogowany użytkownik wychodzi z kolejki oczekiwania na grę w szachy.
- System dobiera pary użytkowników z kolejki oczekiwania do wspólnej gry.
- Użytkownik w grze wykonuje ruchy zgodne z zasadami wybranej gry.
- System kończy grę po wykonaniu ruchu matującego przez jednego z uczestników gry.
- System kończy grę po upływie czasu jednego z użytkowników biorącego udział w rozgrywce.
- System kończy grę po spełnieniu innego warunku zależnego od trybu gry.
- System wyświetla użytkownikom wiadomość o wyniku gry.

1.4 Wymagania pozafunkcjonalne

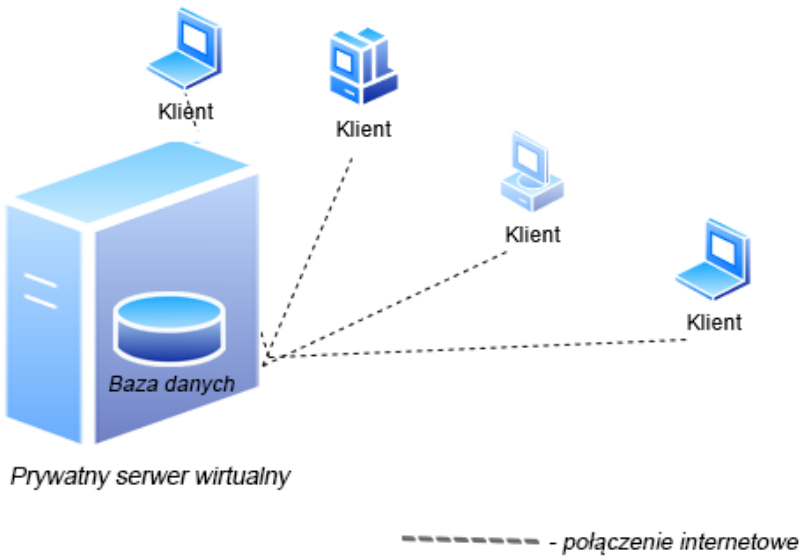
W projekcie wyspecyfikowano niżej wymienione wymagania pozafunkcjonalne.

- Aplikacja kliencka ma być dostępna w formie aplikacji sieciowej.
- Aplikacja kliencka ma działać na następujących przeglądarkach internetowych: Google Chrome, Mozilla Firefox, Opera i Microsoft Edge.
- Dane użytkowników mają być przechowywane w bazie danych.
- Hasła użytkowników mają być przechowywane w formie skrótu hasła z dodaną solą.
- Hasła użytkowników mają zawierać minimalnie 6 znaków, co najmniej jedną wielką literę i co najmniej jedną cyfrę.
- Dane o przeprowadzonych grach szachowych mają być przechowywane w bazie danych.
- Baza danych ma być przechowywana w formie skonteneryzowanej.
- Aplikacja serwera ma być przeznaczona na system Linux.
- Aplikacja serwera ma wykorzystywać szachową notację algebraiczną do zapisu ruchów (patrz sekcja 2.4).
- Aplikacja serwera ma wykorzystywać FEN (patrz sekcja 2.5) do zapisu pozycji bierek w grze.
- Aplikacja serwera oraz kliencka ma być dostępna w języku angielskim.
- Dostęp do aplikacji klienckiej ma być ciągły, z wyjątkiem przerw technicznych.

- System ma informować użytkowników o powodzeniu lub niepowodzeniu przeprowadzonych operacji.
- Aplikacja kliencka ma posiadać przejrzysty interfejs graficzny.

1.5 Architektura systemu

W projekcie zastosowana zostanie struktura klient-serwer, umożliwiająca wielu graczom rozgrywkę. Poglądowo została przedstawiona na rysunku Rys. 1.



Rys. 1. Architektura systemu

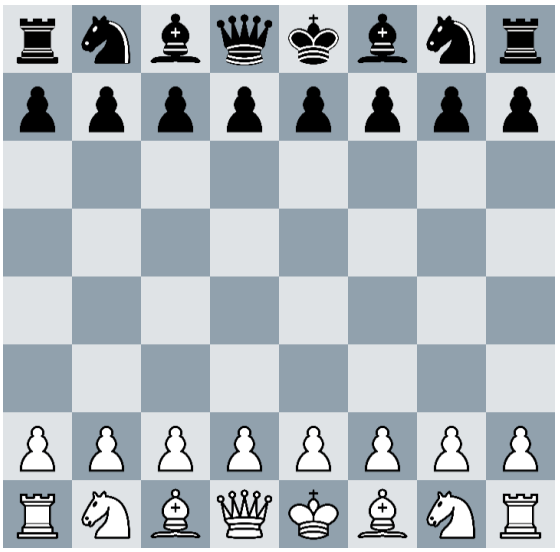
Głównym zadaniem samego serwera jest przesyłanie danych między aplikacjami klienckimi. Niemniej jednak serwer będzie również odpowiadał za komunikację z bazą danych przy rejestracji, logowaniu oraz edycji konta użytkowników, pobieraniu ich statystyk i zapisywaniu wyników gier. Na serwerze przechowywane są również informacje o bieżących grach, które wykorzystywane są, między innymi, do weryfikacji ruchów graczy. Baza danych przechowywana jest w postaci skonteneryzowanej.

2. PODSTAWY TEORETYCZNE

Poniżej zostały przedstawione opisy reguł gry we wszystkich dostępnych trybach rozgrywki w omawianej aplikacji.

2.1 Opis reguł gry w szachy klasyczne

Gra w szachy odbywa się na planszy - szachownicy w kształcie kwadratu o 64 polach w dwóch kolorach. Kolory pól są ułożone naprzemiennie w ośmiu rzędach, które opisane są cyframi od 1 do 8, i kolumnach opisanych literami od “a” do “h”. Gra odbywa się przy użyciu 32 bierek, po 16 dla każdego z zawodników. Ustawienie początkowe planszy zawsze jest takie samo (patrz rysunek Rys. 2.).



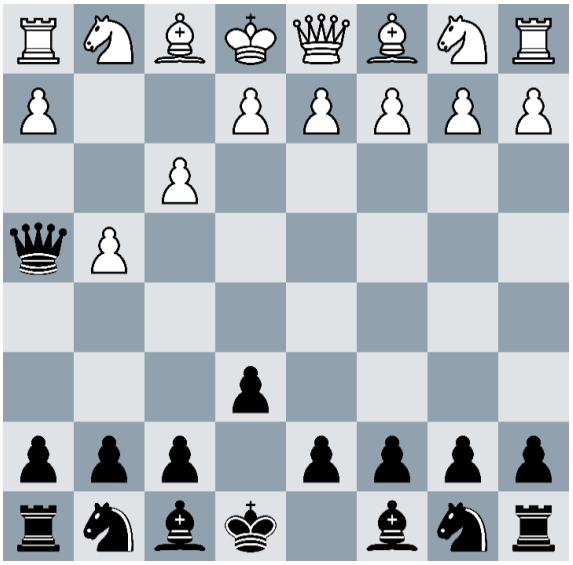
Rys. 2. Początkowa pozycja bierek na szachownicy

Każdy z graczy na początku gry ma do dyspozycji osiem pionów, dwie wieże, dwa gońce, dwa skoczki, hetmana oraz króla. Grę zawsze rozpoczyna gracz z białymi figurami, dlatego o tym, kto zagra białymi decyduje się przez losowanie.

W grze każda z sześciu różnych rodzajów bierek porusza się inaczej. Żadna z figur nie może przechodzić przez inne bierki oraz nie mogą iść na pole, na którym stoi bierka tego samego koloru, z wyjątkiem skoczka, który może przeskakiwać nad bierkami. Jeżeli bierka może wykonać ruch, na pole, na którym znajduje się bierka przeciwnego koloru, bierka ta zostaje zbita, czyli zostaje zdjęta z planszy. Szczegółowy opis możliwych ruchów każdej z bierek można znaleźć w literaturze [16].

W szachach jest kilka warunków końcowych, zakańczających grę: mat, remis, poddanie się czy upływ czasu.

Głównym celem gry jest zmatowanie króla przeciwnika. Mat następuje wtedy, gdy król jest atakowany przez inną bierkę i nie ma żadnego pola do ucieczki (patrz rysunek Rys. 3). Wygrywa wtedy osoba z kolorem, która wykonała ruch matujący. Kolejnym sposobem na zwycięstwo jest przekroczenie przez przeciwnika dozwolonego limitu czasowego. Dodatkowo obaj z zawodników mogą się w każdej chwili poddać.

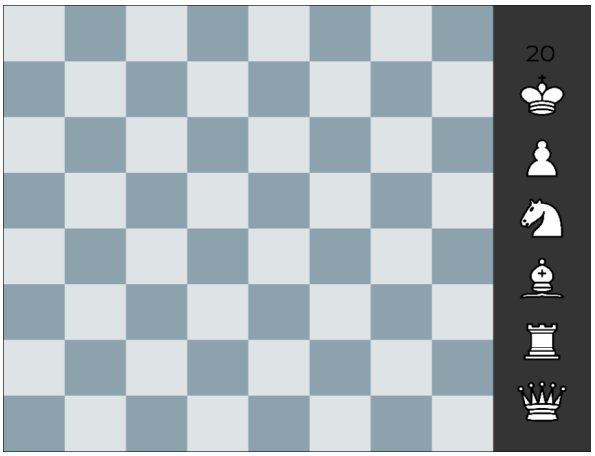


Rys. 3. Przykład mata wykonanego przez gracza z czarnymi bierkami

Przed rozpoczęciem gry w szachy, obu graczom ustawia się czas gry (10 minut). W momencie rozpoczęcia gry, uruchamia się zegar gracza białych bierek. Po wykonaniu ruchu, uruchamia się zegar przeciwnika, a zegar gracza, który wykonał ruch, zostaje zatrzymany. Czas jest odmierzany w ten sposób na przemian przez całą grę. Upływ czasu jednego z zawodników jest jednym z warunków końcowych gry i przyznaje wygraną przeciwnikowi.

2.2 Opis reguł trybu *ChessDefender* dla dwóch graczy

W trybie *ChessDefender* dla dwóch graczy, plansza nie jest inicjalizowana początkowym ustawieniem przedstawionym w punkcie 2.1, a jest pozostawiona pusta (patrz rysunek Rys. 4).



Rys. 4. Początkowy widok trybu chessDefender z perspektywy białych bierek

Na początku gry, graczom zostają przydzielone punkty (dalej punkty *defender*), które mogą wydawać w zamian za ustawienie bierek na planszy. Ten etap rozgrywki będzie dalej nazywany pierwszą fazą rozgrywki. Koszty figur w pierwszej fazie rozgrywki są następujące:

- pion - 1 punkt *defender*,
- skoczek - 3 punkty *defender*,
- goniec - 3 punkty *defender*,
- wieża - 5 punktów *defender*,
- hetman - 9 punktów *defender*.

Bierki można stawiać tylko na trzech pierwszych rzędach planszy patrząc z perspektywy danego gracza. Aby ustawić bierkę, trzeba przeciągnąć ją na plansze podczas swojego ruchu. Jeżeli jednemu z graczy zostanie zero punktów, czeka on aż jego przeciwnik wyda resztę punktów. Gdy obu graczom zostanie zero punktów, następuje możliwość położenia króla. Gracz grający bierkami białymi ustawia króla jako pierwszy. Króla nie można postawić na polu atakowanym przez bierkę przeciwnika. Od tego momentu rozpoczyna się faza druga rozgrywki, w której rozgrywka toczy się według zasad szachowych przytoczonych w sekcji 2.1.

2.3 Opis reguł trybu *ChessDefender* dla jednego gracza

W trybie *ChessDefender* dla jednego gracza, użytkownik rozgrywa partię przeciwko graczowi komputerowemu korzystającego ze sztucznej inteligencji. Plansza jest inicjalizowana przez jedną z wcześniej przygotowanych pozycji. Gracz od początku widzi końcowe ułożenie bierek przeciwnika.

Zadaniem gracza jest ułożenie pozycji, którą wygra z komputerem, używając do tego jak najmniejszej liczby punktów. Gracz ustawia bierki według zasad opisanych w rozdziale 2.2, przyjmując, że przeciwnik od początku nie posiada żadnych punktów. Po wydaniu wszystkich punktów i ustawieniu króla, gra toczy się według zasad szachowych przytoczonych w sekcji 2.1. W tym trybie nie wykorzystano punktów ELO, a jedynie zapisuje się najlepsze dotychczas uzyskane wyniki, które są odzwierciedleniem odwrotności wydanych punktów.

2.4 Szachowa notacja algebraiczna

Do zapisu wykonywanych przez graczy ruchów szachowych w bazie danych wykorzystano szachową notację algebraiczną (ang. *algebric notation*, skrót *AN*). Jest to notacja [13] obowiązująca w turniejach i grach Międzynarodowej Federacji Szachowej (fr. *Fédération Internationale des Échecs,* w skrócie *FIDE*) [1]. FIDE zaleca stosowanie tej ujednoliconej notacji w literaturze. Jest ona też wykorzystywana w innych programach szachowych (np. *StockFish* [8]).

Poniżej przedstawione są zasady szachowej notacji algebraicznej.

1. Każda figura jest oznaczona dużą literą od angielskiego skrótu (król - K, pion - P, skoczek- N, goniec - B, wieża - R, hetman- Q). Jedynie piony są rozpoznawane przez brak tej litery (np. e5, oznacza ruch piona na pozycję e5).

2. Osiem linii pól szachownicy oznaczonych jest małymi literami a,b,c,d,e,f,g,h (od prawej do lewej, patrząc z perspektywy białych).

3. Osiem rzędów pól szachownicy numeruje się kolejno 1, 2, 3, 4, 5, 6, 7, 8 (od najbliższego pola do najdalszego, patrząc z perspektywy białych).

4. Posunięcia figurami zapisuje się za pomocą symbolu figury oraz pola docelowego (np. Ne5, Rd3). Jeżeli dwie identyczne figury mogą wykonać ruch na to samo pole, rozróżnia się je podając literę kolumny lub numer rzędu, z którego następuje ruch.

5. Jeżeli figura bije bierkę przeciwnika zapisuje się to wstawiając znak “x” pomiędzy symbol literowy figury i pole docelowe (np. Bxf3).

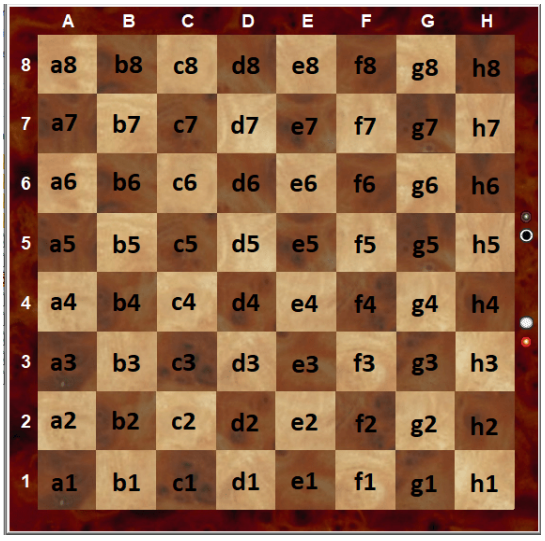
6. Jeżeli to pionek wykonuje bicie to zapisuje się symbol literowy linii na której stał pion wykonujący bicie, oraz oznaczenie pola docelowego poprzedzone znakiem “x” (np. dxe3, hxg7).

7. W przypadku promocji piona, po jego posunięciu należy dodać symbol literowy figury, do której dany pion zostanie wypromowany (np. d8Q, exf8N).

8. Pozostałe ruchy zapisuje się w postaci skrótów:

    - O-O, oznacza roszadę krótką,
    - O-O-O, oznacza roszadę długą,
    - +, oznacza szach (dopisuje się go na końcu posunięcia),
    - #, oznacza mat.

Z powyższych punktów wynika, że każde z 64 pól szachownicy ma jednoznaczne oznaczenie literowo-cyfrowe (patrz rysunek Rys. 5). Przykład partii opisanej szachową notacją algebraiczną jest widoczny w załączniku nr 2 pracy.



Rys. 5. Notacja pól na szachownicy

2.5 Notacja Forsytha-Edwardsa

Notacja Forsytha-Edwardsa (ang. *Forsyth–Edwards Notation*, w skrócie *FEN*) [10] to sposób zapisu pozycji gry szachowej, którego celem jest przechowywanie wszystkich niezbędnych informacji do ponownego rozpoczęcia gry od podanej pozycji.

Zapis *FEN* definiuje dowolną pozycję szachową, używając jedynie znaków ASCII. Jest podzielony na 6 pól, oddzielonych spacją.

1. Pozycja bierek - w zapisie FEN pozycje bierek są zapisane zawsze z perspektywy białego, zaczynając od najwyższej linii i kończąc na najniższej. Tak samo jak w szachowej notacji algebraicznej (patrz sekcja 2.4). Białe bierki zapisuje się wielkimi literami, a czarne małymi (np. biały pion to P, a czarny to p). Puste pola są notowane przy użyciu cyfr od 1 do 8, które oznaczają liczbę wolnych pól z rzędu (np. zapis /4P3/ oznaczałby, że w linii znajdują się cztery puste pola, biały pion, a następnie trzy puste pola). Kolejne linie w zapisie są oddzielone znakiem "/".

2. Kolor gracza, który aktualnie powinien wykonać ruch, zapisany w postaci skrótu koloru z języka angielskiego. "w" (ang. *white*), oznacza, że ruch mają białe, "b" (ang. *black*), że czarne.

3. Możliwość roszady [16]. Jeżeli żadna ze stron nie może wykonać roszady, notuje się "-". Możliwość roszady od strony króla, notuje się literą "k", natomiast roszadę od strony hetmana za pomocą litery "q". Możliwość roszady po stronie białych zapisuje się wielkimi literami, a po stronie czarnych małymi. (np. "Qk", oznaczałoby, że biały może wykonać roszadę po stronie hetmana, a czarny po stronie króla).

4. Możliwość bicia w przelocie [16] na polu docelowym. Brak możliwości bicia w przelocie oznacza się "-". Jeżeli taka możliwość istnieje, notuje się pozycję za poruszonym pionem (czyli pozycję docelową bicia w przelocie).

5. Liczba połówek ruchów - zwiększana przy każdym posunięciu. Resetuje się ją po poruszeniu dowolnego piona lub po jego zbiciu.

6. Liczba pełnych ruchów - zwiększana o jeden co pełny cykl, czyli poruszenie figury białych oraz figury czarnych.

Przykład użycia FEN widoczny w załączniku nr 3.

2.6 Ranking szachowy ELO

Punktacja ELO jest liczbą naturalną wyznaczającą relatywną siłę gry szachistów. Nazwa pochodzi od nazwiska pomysłodawcy systemu - Arpada Elo.

Każdemu z nowych graczy przyznawana jest na początek określona liczba punktów ELO. Następnie po każdej grze punktacja ELO jest aktualizowana. Proces aktualizacji polega na obliczeniu spodziewanego wyniku na podstawie aktualnej liczby punktów ELO graczy oraz porównaniu tych obliczeń z realnym wynikiem gry. Jeżeli zawodnik uzyskał lepszy

wynik niż oczekiwany jego ranking powinien wzrosnąć, jeżeli gorszy to jego ranking powinien zmaleć.

W projekcie używana jest zmodernizowana wersja rankingu ELO - GLICKO-2 (nazwa pochodząca od nazwiska twórcy - Marka Glickmana), która poza obliczaniem relatywnej siły gry szachistów używa również odchylenia standardowego rankingu i jego zmienności. Wartość zmienności to oczekiwana fluktuacja w rankingu gracza. Jest ona wysoka gdy wyniki gracza przestają być stabilne (np. gdy gracz osiąga wyjątkowo dobre wyniki po okresie, w którym wcześniej przegrywał). Gdy gracz osiąga wyniki na stałym poziomie wartość zmienności jest niska. Dokładne wzory pozwalające obliczyć ranking szachowy ELO dostępne są w literaturze [2].

2.7 Silnik szachowy *StockFish*

StockFish to otwartoźródłowy silnik szachowy, stworzony pierwotnie przez Tord'a Romstad'a, Marco Costalb'a oraz Joon'a Kiiski'ego. Aktualnie liczba kontrybutorów wynosi dwieście osób. W 2021 *StockFish* w wersji 14, zdobył ranking ELO o wartości około 3544, dla porównania Magnus Carlsen - wielokrotny i aktualny mistrz świata w szachy, posiada 2865 punktów ELO[2].

Silnik *StockFish* opiera się na zaawansowanym algorytmie przeszukującym - Alpha-Beta [4]. Zdobył popularność przez osiąganie wysokich prędkości poszukiwań, nawet na dużych głębokościach programu. Stało się to dzięki dużo bardziej agresywnemu algorytmowi cięć Alfa-Beta oraz tzw. *late move reductions*.

Late move reductions jest ulepszeniem algorytmu Alpha-Beta, polegającym na zakładaniu, które ruchy będą najbardziej opłacalne, aby następnie przeszukać je w pierwszej kolejności. W szachach, jednymi z takich ruchów, są na przykład ruchy zbijające bierki przeciwnika. Dodatkowo *StockFish* zakłada, że jeżeli dany ruch, bądź kilka ruchów w pozycji, okaże się bardzo opłacalna, to w podobnych pozycjach, również jest na to duża szansa. W takim wypadku te ruchy też są traktowane priorytetowo przez algorytm.

Przedstawione wyżej zasady są tylko niektórymi, z których przy ewaluacji korzysta *StockFish*. Takie podejście pozwala zredukować czas wykonywania algorytmu i pozwala programowi na głębsze przeszukiwanie krytycznych gałęzi drzewa.

StockFish korzysta również z bazy danych zawierających najpopularniejsze otwarcia, z których na początku każdej z gier wybiera losowo jedno do zagrania. Dodatkowo w 2014 roku, *StockFish* zaczął korzystać z bazy danych rozwiązanych końcówek "Syzygy"[3]. Jest to baza danych, zawierająca już przeanalizowane końcówki szachowe. Na ten moment, rozwiązane są wszystkie końcówki, które mogą wystąpić przy dowolnej kombinacji siedmiu bierek. A więc w momencie gdy na planszy pozostaje mniej niż osiem bierek, *StockFish* zna już wynik rozgrywki, dla każdej z możliwych pozycji.

[2] https://ratings.fide.com/profile/1503014
[3] https://syzygy-tables.info/

3. KONCEPCJA

3.1 Języki programowania

Poniżej zostały przedstawione języki programowania, które zostaną wykorzystane przy implementacji zadania projektowego.

- Javascript - język umożliwiający proste tworzenie funkcjonalności stron internetowych.
- Python - język użyty do implementacji serwera używanego w omawianej aplikacji. Dodatkowo biblioteka Python-chess umożliwia integrację silnika Stockfish z Pythonem.
- SQL - język niezbędny do tworzenia zapytań do bazy danych.

3.2 Biblioteki

Poniżej zostały przedstawione biblioteki, które zostaną wykorzystane przy implementacji zadania projektowego.

- P5.js - biblioteka służąca głównie do tworzenia aplikacji graficznych dla stron internetowych. W implementowanej aplikacji klienckiej zostanie użyta do renderowania szachownicy, bierek, obsługi przycisków myszy oraz pozycji kursora.
- Hashlib - biblioteka dająca dostęp do implementacji wielu różnych algorytmów typu *secure hash* (SHA) oraz *message digest* (MD). Między innymi uwzględnione są algorytmy FIPS SHA1, SHA224, SHA256, SHA384 oraz SHA512. Zostanie użyta do tworzenia skrótów haseł użytkowników w aplikacji serwera.
- Flask - biblioteka służąca głównie do implementacji REST API. Została wybrana głównie ze względu na doświadczenie zespołu projektowego i zostanie użyta do implementacji komunikacji asynchronicznej między klientem a serwerem.
- Flask socketio - biblioteka użyta do ułatwienia implementacji *websocket*'ów, które zostaną użyte do dupleksowej komunikacji między klientem a serwerem. Wybrana ze względu na łatwość łączenia z serwerem Flask.
- MySQL Connector - biblioteka pozwalająca na nawiązanie połączenia z relacyjną bazą danych MySQL, wysyłanie do niej zapytań i odbierania danych. Zostanie wykorzystana do umożliwienia obsługi zapytań serwera Flask do bazy danych.
- Python-chess - biblioteka użyta do weryfikacji poprawności otrzymanych ruchów po stronie serwera, jej użycie pozwala na odrzucenie nielegalnych ruchów przez serwer.

- Qrcode - biblioteka umożliwiająca wygenerowanie kodu QR, za pomocą którego użytkownik będzie mógł skorzystać z weryfikacji dwuetapowej zwiększającej poziom bezpieczeństwa konta.
- Itsdangerous - biblioteka pozwalająca na podpisanie wrażliwych danych takich jak tokeny w celu upewnienia się, że odebrane dane nie zostały zmodyfikowane przez osoby trzecie.
- Email - biblioteka pozwalająca na przygotowywanie wiadomości e-mail m.in. przez dodawanie adresata, tematu wiadomości, jej zawartości, itp. Wykorzystywana m.in. do wysyłania maili potwierdzających założenie konta.
- Smtplib - biblioteka pozwalająca na wysyłanie wcześniej przygotowanej wiadomości e-mail za pomocą poczty elektronicznej do wskazanego adresata. Zostanie wykorzystana m.in. przy weryfikacji adresu email podanego przez użytkownika.
- PyOTP - biblioteka pozwalająca na generowanie oraz weryfikowanie tzw. *one-time password* przy weryfikacji dwuetapowej.
- React - biblioteka javascript wykorzystywana do tworzenia interfejsów graficznych aplikacji internetowych.
- Redux - biblioteka pozwalająca na tworzenie ‘kontenera stanu’ (ang. *state container*) po stronie klienta, który umożliwia łatwy dostęp do konkretnych danych stanowych (jak np. nazwa aktualnie zalogowanego użytkownika) z wielu miejsc w kodzie.
- React router - biblioteka pozwalająca na *routing* (przenoszenie między podstronami) po stronie klienta w aplikacji klienckiej. Zostanie użyta do łatwej implementacji podstron aplikacji.
- React hot toast - biblioteka umożliwiająca wyświetlanie w przejrzysty sposób powiadomień użytkownikom aplikacji sieciowej (np. o poprawnym dołączeniu do gry).
- Js sha256 - biblioteka dająca dostęp do implementacji algorytmu SHA256. Zostanie użyta do tworzenia skrótu hasła użytkownika w aplikacji klienckiej.
- React google recaptcha - biblioteka ułatwiająca implementację captchy google w aplikacji klienta.
- Font awesome - biblioteka ułatwiająca dostęp do bezpłatnej galerii ikon *font awsome*. Zostanie użyta do zwiększenia estetyki i przejrzystości aplikacji klienckiej.

3.3 Protokoły komunikacyjne

Poniżej zostały przedstawione protokoły komunikacyjne, które zostaną wykorzystane przy implementacji zadania projektowego.

- HTTP (*Hypertext Transfer Protocol*) - używany do przesyłania dokumentów WWW, jak i komunikacji w REST API. Zapytania do REST API używane są w systemie logowania, rejestracji oraz przy pobieraniu przez użytkownika statystyk gier, ponieważ nie jest wymagana w tym celu stała, dupleksowa komunikacja. Dodatkowo, każde zapytanie REST API wykonane po zalogowaniu gracza jest opatrzone nagłówkiem "*Authorize*" zawierającym klucz sesyjny użytkownika, w ten sposób upoważniając go do dostępu do prywatnych zasobów, takich jak historia gier. Protokół HTTP udostępnia wiele metod przystosowanych do poszczególnych celów. Najpopularniejsze z nich to:
    - GET - Zapytanie służące do pobrania zasobów z serwera znajdujących się pod konkretnym adresem. Zgodnie ze standardem HTTP [4] zapytanie to powinno służyć tylko do tego celu.
    - POST - Metoda wykorzystywana do wysyłania żądania na serwer w celu utworzenia zasobu. Format danych zawartych w nagłówku określany jest w polu Content-Type. Najpopularniejszym formatem jest format application/json
    - PUT - Żądanie oznaczające, że dany zasób istnieje na serwerze, a dane dostarczone razem z żądaniem oznaczają ich nowszą wersję, a zasób istniejący powinien zostać nadpisany.
    - DELETE - żądanie mające na celu usuwanie danego zasobu.

  Po otrzymaniu żądania jest ono przetwarzane przez serwer. Następnie, do klienta odsyłana jest odpowiedź zawierająca kod oznaczający status odpowiedzi, odpowiedni nagłówek oraz ew. ciało odpowiedzi. Wyróżnia się pięć grup kodów odpowiedzi:
    - 1xx - Kody informacyjne. Serwer otrzymał żądanie i jest w trakcie jego przetwarzania.
    - 2xx - Kody powodzenia. Serwer pomyślnie odebrał żądanie od klienta, zrozumiał je i zaakceptował.
    - 3xx - Kody przekierowania. Konieczne jest wykonanie dodatkowych akcji w celu zakończenia żądania.
    - 4xx - Kody błędu aplikacji klienta. Zapytanie zawiera niepoprawną składnię i nie może zostać wykonane.
    - 5xx - Kody błędu serwera. Serwer nie był w stanie przetworzyć żądania mimo poprawnego zapytania.

- TCP (*Transmission Control Protocol)* - połączeniowy, niezawodny, strumieniowy protokół komunikacyjny stosowany do przesyłania danych między procesami uruchomionymi na różnych maszynach.

- IPv4 (*Internet Protocol*) - pozwala między innymi na identyfikację hostów, co jest niezbędne do transmisji danych.
- SMTP (Simple Mail Transfer Protocol) - protokół komunikacyjny specyfikujący sposób dostarczania poczty elektronicznej w Internecie.

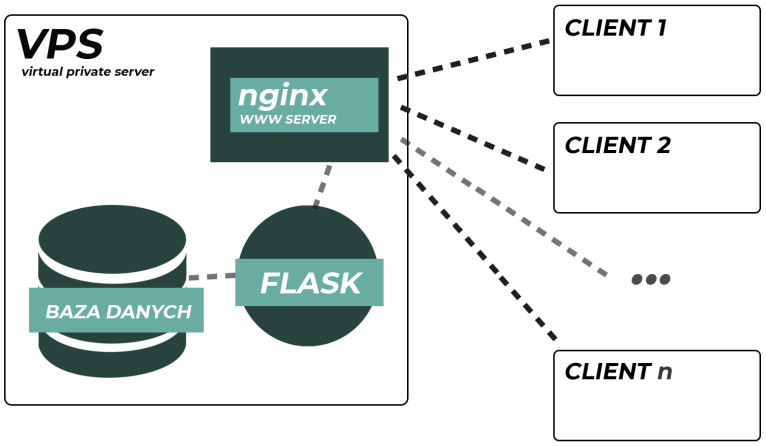
3.4 Narzędzia programistyczne

Poniżej zostały przedstawione narzędzia programistyczne, które zostaną wykorzystane przy implementacji zadania projektowego.

- Webstorm - środowisko programistyczne do tworzenia aplikacji internetowych. Wybrane głównie ze względu na prostotę interfejsu graficznego. Środowisko to pozwala na między innymi edycję oraz analizę kodu źródłowego, użycie graficznego debuggera zintegrowanego z przeglądarką czy integrację z systemem kontroli wersji.
- Pycharm - środowisko programistyczne dla języka programowania Python. Został wybrany z powodów analogicznych do środowiska Webstorm.
- Visual Studio Code - darmowy edytor kodu źródłowego. Dzięki wielu wtyczkom pozwala na bardzo swobodną analizę oraz edycję tekstu. Szczególnie użyteczny do pisania i testowania kodu aplikacji klienckiej.
- Docker - narzędzie pozwalające na wirtualizację oraz konteneryzację zasobów. Umożliwia łączenie aplikacji z zależnościami i bibliotekami systemu operacyjnego wymaganymi do uruchomienia w dowolnym środowisku. W projekcie został on wykorzystany do wirtualizacji bazy danych. Docker został wybrany przede wszystkim na potrzeby rozwijania aplikacji w środowisku lokalnym, jednakże narzędzie można wykorzystać w środowisku produkcyjnym systemu.
- Github - serwis internetowy wykorzystujący system kontroli wersji pozwalający na wersjonowanie kodu aplikacji.
- Postman - narzędzie umożliwiające testowanie odpowiedzi API na żądania. Aplikacja pozwala m.in. na wysyłanie zapytań GET, POST, PUT, DELETE [7].

3.5 Elementy systemu

Wyodrębnione w opracowanym systemie informatycznym moduły przedstawiono na rysunku Rys. 6.



Rys. 6. Elementy systemu

Można wśród nich wyróżnić:

- aplikacja kliencka - odpowiedzialna za komunikację z serwerem nginx,
- serwer nginx - udostępnia stronę internetową oraz jest pośrednikiem między serwerem Flask a klientami,
- serwer Flask - używany przy implementacji *REST API* obsługującego m.in. serwis logowania,
- baza danych - przechowuje dane użytkowników oraz zapis przebiegu gier.

4. IMPLEMENTACJA

4.1 Struktury danych

Ponieważ w językach programowania użytych w projekcie nie określa się jawnie typów zmiennych, w przedstawianych strukturach danych zostały one podane jedynie orientacyjnie. Ich przeznaczeniem jest ułatwienie zrozumienia użycia zmiennych jednak nie mają one bezpośredniego przełożenia na kod programu. Poniżej opisano najważniejsze zdefiniowane struktury danych, w formacie:

- nazwa struktury - krótki opis.
    - nazwa zmiennej (typ zmiennej) - opis pola.

4.1.1 Struktury danych w aplikacji serwera

Poniżej opisano struktury danych zdefiniowane po stronie aplikacji serwera:

- *Player* - struktura danych przechowująca informacje o graczu ułatwiające obsługę klientów.
    - *UserID* (int) - unikalny identyfikator użytkownika pobrane z bazy danych po zalogowaniu.
    - *Username* (string) - nazwa użytkownika.
    - *Password* (string) - skrót hasła użytkownika.
    - *Salt* (string) - sól dodana do hasła.
    - *Email* (string) - adres e-mail użytkownika, niezbędny do aktywowania konta / resetowania hasła.
    - *2FA* (bool) - użycie weryfikacji dwuetapowej przez użytkownika.
    - *OTPSecret* (string) - pozwala na weryfikację wprowadzonego przez użytkownika hasła jednorazowego.
    - *AccountConfirmed* (bool) - pole zawierające stan aktywacji konta użytkownika.
    - CreatedAt (string) - Zawiera datę oraz czas utworzenie konta.
    - UpdatedAt (string) - Data oraz czas ostatniej aktualizacji konta.
    - LoggedInAt (string) - Data aktualnego logowania na konto.
    - LastLoggedInAt (string) - Data ostatniego logowania na konto.
    - *ELO* (int) - aktualne ELO (patrz sekcja 2.6) użytkownika, niezbędne przy doborze przeciwnika.
    - *Playing_as* (char) - kolor, którym aktualnie gra użytkownik zapisany w postaci jednej litery (biały (ang. *white*) - w, czarny (ang. *black)* - b, żaden (ang. *none*) - n).
    - *Time_spent_in_queue* (int) - aktualny czas spędzony w kolejce podany w milisekundach.
    - *Current_scope* (int) - maksymalna różnica w punktach ELO między dwoma użytkownikami przy doborze przeciwnika.

- *Timer* - struktura danych przechowująca czas, jaki pozostał na grę każdemu z graczy.
    - *White_time* (float) - czas, jaki pozostał na grę graczowi grającemu białymi bierkami. Podany jest w sekundach.
    - *Black_time* (float) - czas, jaki pozostał na grę graczowi grającemu czarnymi bierkami, podany w sekundach.
    - *Last_timestamp* (float) - czas serwera, kiedy ostatnio były aktualizowane zmienne *White_time* oraz *Black_time*. Różnica między nowym czasem serwera a *Last_timestamp* jest używana do aktualizacji czasu, jaki pozostał graczom.

- *Game* - struktura danych przechowująca informacje o stanie aktualnie prowadzonej gry.
    - *Game_id* (string) - unikalny identyfikator gry.
    - *Game_mode_id* (int) - liczba identyfikująca tryb gry.
    - *White_player (Player)* - informacje o graczu grającym białymi bierkami.
    - *Black_player (Player)* - informacje o graczu grającym czarnymi bierkami.
    - *Curr_FEN* (string) - aktualny FEN (patrz sekcja 2.5) gry.
    - *Timer* (*Timer*) - aktualne stany zegarów szachowych graczy.
    - *Defender_state* (DefenderState) - aktualny stan gry w trybie defender.
    - *Draw_proposed* (string) - kolor bierek gracza, który zaproponował remis, gdy żaden z graczy tego nie zrobił ustawiona na 'null’.

- GameMode - struktura danych przechowująca informacje o trybie rozgrywki.
    - *Game_mode_id* (int) - unikalny identyfikator trybu gry.
    - Game_mode_name (string) - nazwa trybu gry.
    - Game_mode_desc (string) - opis trybu gry.
    - Game_mode_time (int) - maksymalny czas rozgrywki w trybie gry, podany w sekundach.
    - Game_mode_starting_FEN (string) - FEN od jakiego rozpoczyna się rozgrywka w trybie.
    - Game_mode_multiplayer (bool) - czy tryb rozgrywki jest rozgrywany z przeciwnikami online (wartość *true*) czy jedynie z przeciwnikiem komputerowym (wartość *false*).

- DefenderState - struktura danych przechowująca informacje o aktualnym stanie rozgrywki w trybie defender.
    - *Black_score* (int) - liczba punktów *defender* gracza grającego bierkami czarnymi.
    - *White_score* (int) - liczba punktów *defender* gracza grającego bierkami białymi.
    - *Phase* (int) - faza rozgrywki *defender* (patrz sekcja 2.2).

- *Server State* - struktura danych przechowująca informacje o aktualnym stanie serwera oraz stałe niezbędne do jego działania.

- *Sessions* (dictionary) - słownik przechowujący pary, w których kluczem jest *id* użytkownika a wartością tablica zawierająca tokeny używane przy uwierzytelnieniu (patrz sekcja 4.11).
- *AuthorizedSockets* (dictionary) - słownik przechowujący pary, w których kluczem jest *id* użytkownika a wartością *sid* (*socket id*), unikalny identyfikator gniazda *websocket* klienta. Dodawane są do niego jedynie identyfikatory zautoryzowanych gniazd.
- *GameModes* (array) - tablica zawierająca wszystkie zdefiniowane tryby gry.
- *Queue* (dict) - słownik przechowujący pary, w których kluczem jest *id* trybu gry a wartością tablica zawierająca obiekty *Player* reprezentujące graczy aktualnie oczekujących w kolejce danego trybu gry.
- *Games* (dictionary) - słownik przechowujący pary, w których kluczem jest *id gry* a wartością obiekt *Game*.
- *InitialScope* (int) - domyślna maksymalna różnica w punktach ELO między dwoma użytkownikami przy doborze przeciwnika po dołączeniu do kolejki. W aktualnej wersji programu równa się 50.
- *ScopeUpdateInterval* (int) - czas oczekiwania gracza w kolejce, po jakim zwiększa się maksymalna różnica w punktach ELO między dwoma użytkownikami przy doborze przeciwnika, podany w milisekundach.
- *ScopeUpdateAmmount* (int) - wartość, o jaką zwiększa się *scope* po każdorazowym upłynięciu *scopeUpdateInterval*. W aktualnej wersji programu równa się 50.
- *GameModeTimes* (array) - tablica zawierająca maksymalny czas gry każdego użytkownika w trybie gry o *id* równym indeksowi.
- *GameModeStartingFEN* (array) - tablica zawierająca początkowe stany planszy zapisane w notacji FEN.

4.1.2 Struktury danych w aplikacji klienckiej

Poniżej opisano struktury danych zdefiniowane po stronie aplikacji klienckiej.

- *Board* - struktura przechowująca dane o aktualnym stanie planszy szachowej.
  - *Grid* (array) - tablica o stałej długości 64, reprezentująca aktualny stan gry. Przechowuje obiekty struktury *Piece*.
  - GameMode2Grid - tablica wykorzystywana do przechowywania obiektów Piece w fazie inicjalizacji pozycji w trybie *ChessDefender*.
  - *FEN* (string) - skrótowy zapis aktualnej pozycji.
  - *ColorToMove* (char) - kolor, który aktualnie ma prawo wykonywać ruch. Zapisany jest w postaci pierwszej litery koloru w języku angielskim (biały (ang. *white*) - w, czarny (ang. *black)* - b).
  - *LastPawnMoveOrCatpure* (int) - liczba ruchów od ostatniego ruchu pionem lub zbicia piona (potrzebne do sprawdzania zasad gry (patrz sekcja 2.1)).
  - *NumOfMoves* (int) - ilość ruchów wykonanych w grze. Zgodnie z zasadami szachowymi, liczba ruchów zwiększa się co ruch czarnego.

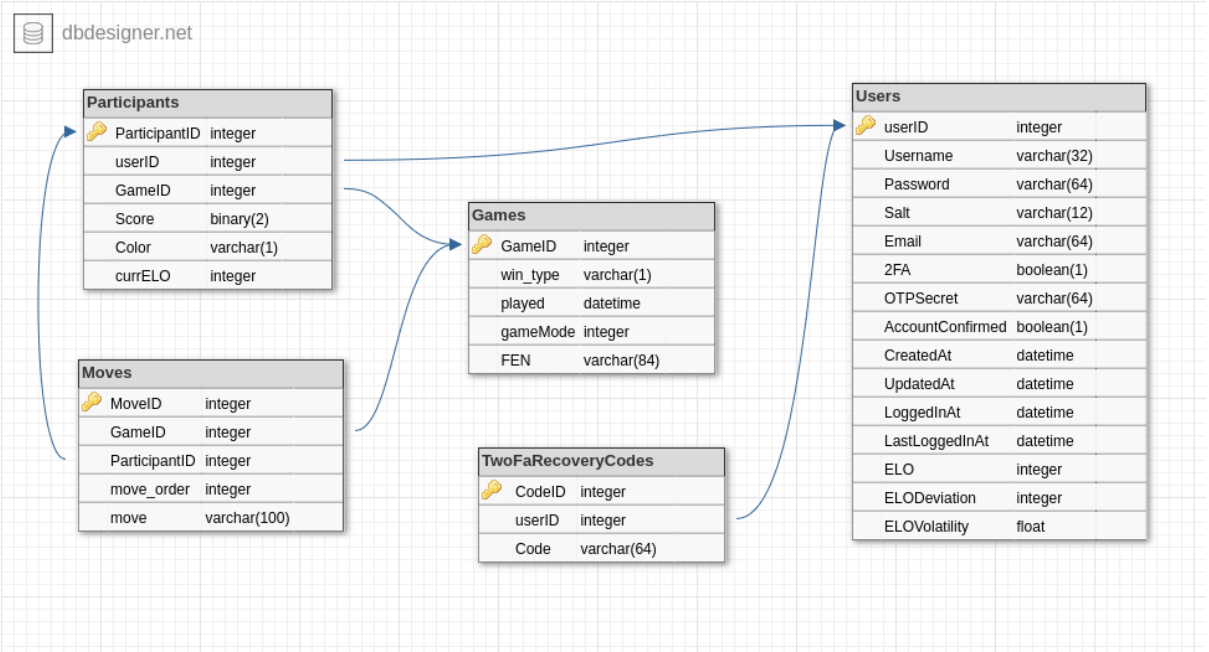
    - *Check* (bool) - ustawiona na True, gdy król aktualnego gracza znajduje się w szachu.
    - *EnPassant* (string) - zmienna przechowująca pozycję za pionem, na którym można wykonać ruch zbijania *en passant* [16] (potrzebna do sprawdzania zasad gry).
    - SetupState (int) - zmienna przechowująca pozostałą ilość punktów *defender*.

- *Move* - struktura przechowująca jeden możliwy ruch bierki.
    - *StartSquare* (int) - indeks w tablicy z aktualnym stanem gry (*grid*), pod którym znajduje się bierka wykonująca ruch.
    - *EndSquare* (int) - indeks pola, na które wykonuje ruch bierka.
    - *Type* (char) - typ wykonywanego ruchu (R - roszada długa, r - roszada krótka, CP - bicie w przelocie, P - wystawienie piona o dwa pola, C - bicie innej bierki).

- *Piece* - struktura przechowująca informacje o bierce.
    - *Color* (char) - kolor bierki. Zapisany w postaci pierwszej litery koloru w języku angielskim (biały (ang. *white*) - w, czarny (ang. *black)* - b, żaden (ang. *none*) - n).
    - *Type* (char) - typ bierki zapisany w postaci jednej litery. (król (ang. *king*) - k, pion (ang. *pawn*) - p, skoczek (ang. *knight*) - n, goniec (ang. *bishop*) - b, wieża (ang. *rook*) - r, królowa (ang. *queen*) - q, żaden - e).
    - *X,Y* (int) - pozycje bierki zapisana jako ilość pikseli od kolejno górnej i lewej krawędzi planszy.
    - oldX, oldY - pozycje bierki przed podniesieniem jej przez użytkownika.
    - *GridPos* (int) - indeks bierki w tablicy *grid*.
    - *PossibleMoves (*array) - tablica przechowująca dozwolone ruchy dla danej bierki w postaci struktury *Move*.
    - *ScaledSize* (int) - rozmiar bierki po skalowaniu, służący do zmieniania jej rozmiaru w razie zmiany rozmiaru okna przeglądarki.
    - didMove (bool) - zmienna przechowująca informacje o tym, czy dana bierka wykonała już jakiś ruch, 0 - nie wykonała, 1 - wykonała. Wykorzystywane między innymi przy sprawdzaniu możliwości wykonania roszady

4.2 Model bazy danych

W projekcie jest zastosowana relacyjna baza danych obsługiwana przy użyciu MySQL. Baza danych jest przechowywana w postaci skonteneryzowanej za pomocą narzędzia docker. Ma ona na celu przechowywanie informacji na temat użytkowników jak i rozegranych przez nich gier. Baza danych zawiera 5 tabel powiązanych ze sobą odpowiednimi relacjami. Rysunek Rys. 7 zawiera wszystkie niezbędne tabele wraz z relacjami oraz kolumnami.



Rys 7. Model bazy danych

4.2.1 Tabela Users

Tabela, która przetrzymuje dane użytkownika niezbędne do zalogowania oraz do określenia i aktualizacji pozycji rankingowej. Poniżej wymieniono kolumny tej tabeli.

- *userID* - identyfikator użytkownika.
- *Username* - unikalna nazwa wybierana przy rejestracji.
- *Password* - hasło wybierane przy rejestracji, przechowywane w postaci skrótu SHA-256 z dodaną solą.
- *Salt* - 10 znakowa sól dodawana do hasła podczas rejestracji nowego konta.
- *Email* - unikalny adres e-mail konta użytkownika podawany przy rejestracji.
- *2FA* - czy użytkownik używa weryfikacji dwuetapowej.
- *OTPSecret* - unikalny Kod pozwalający na weryfikację wprowadzonego hasła jednorazowego.
- AccountConfirmed - Stan aktywacji konta.
- *CreatedAt* - data rejestracji użytkownika.
- *UpdatedAt* - data ostatniej modyfikacji konta.
- LoggedInAt - data obecnego logowania na konto.
- LastLoggedInAt - data ostatniego logowania na konto.
- *ELO* - liczba punktów ELO gracza.
- *ELODeviation* - odchylenie rankingowe używane w algorytmie GLICO-2.
- *ELOVolatility* - zmienność rankingowa używana w algorytmie GLICO-2.

Zważając na fakt, iż tabela *Users* składa się z 15 kolumn, przykładowa zawartość została podzielona między dwie tabele (patrz tabela 2 oraz 3).

Tabela 2. Przykładowa wypełniona tabela Users

userID	Username	Password	Salt	Email	2FA	OTPSecret
1	Test	d202d66cd88c6ff5e067408930bef	g8SYpe1t8CI?	testowy123@gmail.com	0	ORSXG5DPO54TCMRTIBTW2YLJNQ
2	test2	92e9b55e6b2b45f05d2040886574	o7k*j2IjB$OT	test@gmail.com	0	ORSXG5CAM5WWC2LMFZRW63I=
3	user123	7276c71ce838344e91b46f800056	H^akGE0HTfzi	user@gmail.com	1	OVZWK4SAM5WWC2LMFZRW63I=
4	karol	9c42c3b2f34843196115b9acaf053	uZj@mV1mmJPU	karol.zarebski1@wp.pl	0	NNQXE33MFZ5GC4TFMJZWW2JRIB
5	marekmarczewski	9cf5c14b375029789cabc978bdef1	cM7TTtL6Lee1	marekmarczewski1234@gmail.com	0	NVQXEZLLNVQXEY32MV3XG23JGE
6	ankaankowska	ad8e3baf29d19d0e5260b0759b33	S7sMHHifQ$Dm	ankaankowska1@gmail.com	0	MFXGWYLBNZVW653TNNQTCQDHI
7	karol123	86d8ad46f5d4f7ba948daf056220(	oVVUS5mdHwkZ	karolkarolski25@outlook.com	1	NNQXE33MNNQXE33MONVWSMRV

Tabela 3. Przykładowa wypełniona tabela Users

AccountConfirmed	CreatedAt	UpdatedAt	LoggedInAt	LastLoggedInAt	ELO	ELODeviation	ELOVolatility
1	2022-01-28 20:57:45	*NULL*	2022-01-28 22:32:21	2022-01-28 22:24:23	849	97	0.0599959
1	2022-01-28 20:58:01	*NULL*	2022-01-28 22:35:24	2022-01-28 22:23:16	1151	97	0.0599896
0	2022-01-28 22:40:06	*NULL*	*NULL*	*NULL*	1000	350	0.06
0	2022-01-28 22:42:26	*NULL*	*NULL*	*NULL*	1000	350	0.06
0	2022-01-28 22:44:43	*NULL*	2022-01-28 22:44:50	2022-01-28 22:44:50	1000	350	0.06
0	2022-01-28 22:45:41	*NULL*	2022-01-28 22:45:48	2022-01-28 22:45:48	1000	350	0.06
0	2022-01-28 22:46:56	*NULL*	2022-01-28 22:47:04	2022-01-28 22:47:04	1000	350	0.06

4.2.2 Tabela TwoFaRecoveryCodes

Tabela zawierająca informacje na temat kodów odzyskiwania do konta, na którym aktywowana została weryfikacja dwuetapowa. Poniżej zostały wymienione poszczególne kolumny wchodzące w skład tabeli.

- *CodeID* - identyfikator kodu odzyskiwania.
- *userID* - identyfikator użytkownika.
- *Code* - kod odzyskiwania w postaci skrótu.

Przykładowe wartości kolumn tabeli *TwoFaRecoveryCodes* przedstawiono w tabeli 4.

Tabela 4. Przykładowa wypełniona tabela TwoFaRecoveryCodes

CodeID	userID	Code
1	3	65d4acbd2a6b5c880d10ae2e618b039ce8ad31d805bab7e88cc71656c759b941
2	3	b9e8202a85751ebd41a62e77387a812a10a25f62b48455deb55e157bb7884248
3	3	61b3ab6bf18c30340e300c6d4802ba0dfaf78ec719c25f28f7aa4e64d051b157
4	3	5a38f62716a33c756c051e835d45fe1d52492b37f4c8b45c5cd8eb8aeb243f8c
5	3	25f7e52bf3a622917b9043f9a2893962e15517bfba1a0e05a9db80926e91648f
6	3	4b47b284a3a764adbca2f3b6ad7278c7725cc36825b760cf353bc18948ae859b
7	3	9289ec5ad49b6b9e36b62668fec7973bfc0c3e88d6940daabb1ef129e9bc4036

4.2.3 Tabela Games

Tabela, która pełni rolę łącznika między innymi tabelami przechowującymi informacje na temat rozegranych gier. Poniżej wymieniono kolumny tej tabeli.

- *GameID* - identyfikator gry.
- *played* - data rozegrania gry.
- GameMode - Tryb gry.
    - 0 - Klasyczna rozgrywka między dwoma graczami,
    - 1 - Rozgrywka w trybie *ChessDefender* dla dwóch graczy.
    - 2 - Rozgrywka w trybie *ChessDefender* dla jednego gracza.
- FEN - Finałowa pozycja, na której zakończyła się dana rozgrywka.

Przykładowe wartości kolumn tabeli *Games* przedstawiono w tabeli 5.

Tabela 5. Przykładowa wypełniona tabela Games

GameID	played	GameMode	FEN
1	2022-01-28 20:34:10	0	rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1
2	2022-01-28 20:34:46	0	rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1
3	2022-01-28 20:37:48	0	rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1
4	2022-01-28 20:40:02	0	rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1
5	2022-01-28 20:44:25	0	rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1
6	2022-01-28 21:29:54	0	rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1
7	2022-01-28 21:33:21	0	rnbqkbnr/pppp1ppp/8/4p3/5P2/8/PPPPP1PP/RNBQKBNR w KQkq - 0 2
8	2022-01-28 22:12:32	0	rnbqkbnr/pppppppp/8/8/4P3/8/PPPP1PPP/RNBQKBNR b KQkq - 0 1
9	2022-01-28 23:05:19	0	rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1
10	2022-01-28 23:09:15	0	rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1

4.2.4 Tabela Participants

Tabela, która przechowuje informacje na temat uczestników gier oraz umożliwia relację wiele do wielu między tabelą użytkowników i tabelą gier. Poniżej wymieniono kolumny tej tabeli.

- *ParticipantID* - identyfikator uczestnika rozgrywki, pełniącego rolę gospodarza.
- *GameID* - identyfikator gry, w której uczestnik brał udział.
- *UserID* - identyfikator użytkownika, pełniącego rolę uczestnika gry.
- *Score* - wynik danej gry, rozpatrywany od strony uczestnika.
- *Color* - kolor bierek, którymi grał uczestnik.
- *currELO* - wartość rankingowa uczestnika w momencie rozgrywania gry.

Przykładowe wartości kolumn tabeli *Participants* przedstawiono w tabeli 6.

Tabela 6. Przykładowa wypełniona tabela Participants

ParticipantID	GameID	userID	Score	Color	currELO
1	9	2	0	White	1000
2	9	1	1	Black	1000
3	10	2	0	White	838
4	10	1	1	Black	1162
5	11	2	0	White	780
6	11	1	1	Black	1220
7	12	1	0	White	1253

4.2.5 Tabela Moves

Tabela, która przechowuje informacje na temat ruchów wykonanych podczas danej rozgrywki. Poniżej zostały wyszczególnione kolumny wchodzące w skład tabeli.

- *MoveID* - identyfikator ruchu.
- *GameID* - identyfikator rozgrywki.
- *ParticipantID* - identyfikator wykonawcy ruchu.
- *move_order* - kolejność ruchu w grze, liczona od 0.
- *Move* - zapis ruchu w szachowej notacji algebraicznej (patrz sekcja 2.4).

Przykładowe wartości kolumn tabeli *Moves* przedstawiono w tabeli 7.

Tabela 7. Przykładowa wypełniona tabela Moves

MoveID	GameID	ParticipantID	move_order	Move
1	17	17	0	d2d4
2	17	18	1	b7b5
3	18	19	0	e2e4
4	18	20	1	d7d5
5	19	21	0	d2d4
6	21	25	0	f2f4
7	21	26	1	d7d6

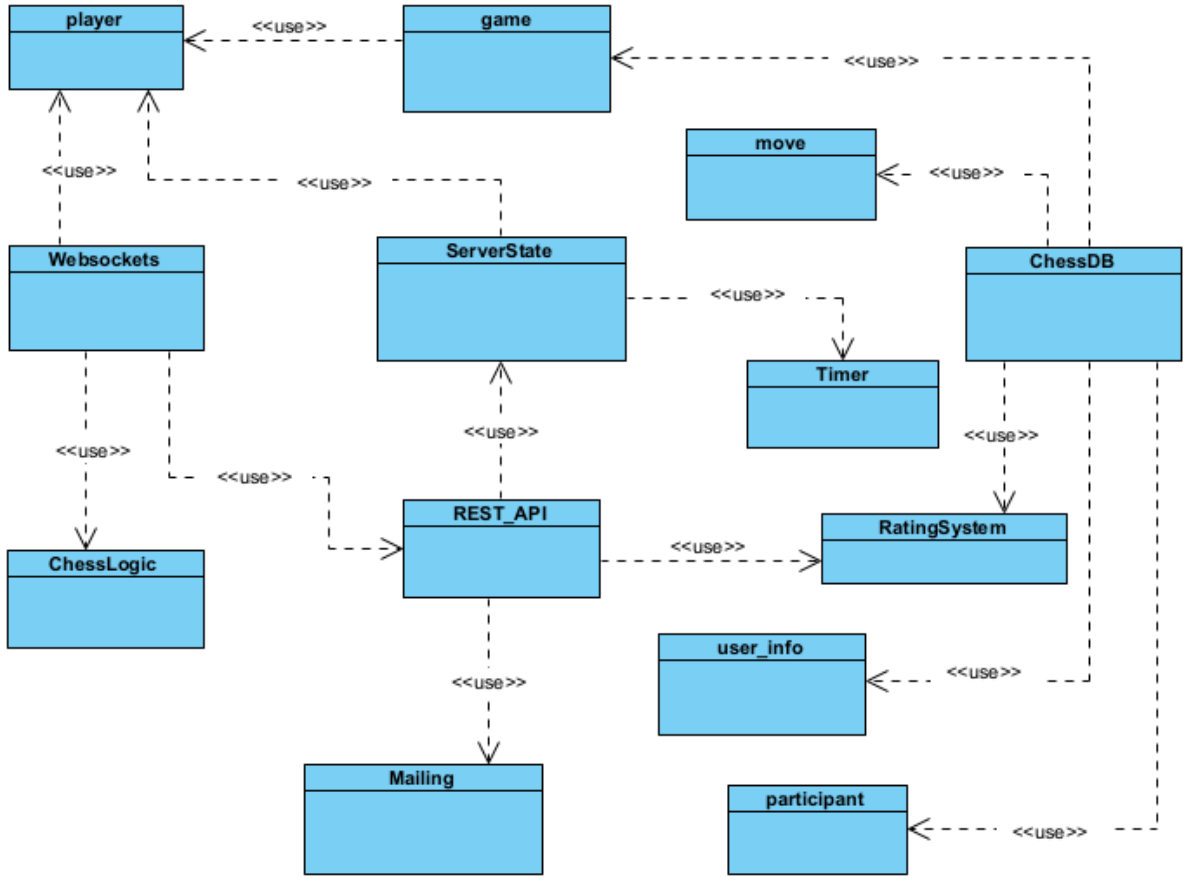
4.3 Diagramy UML

W poniższych sekcjach zamieszczono diagramy UML wykonane w zakresie pracy.

4.3.1 Diagramy klas

Wykonane zostały 2 diagramy klas, gdzie pierwszy pokrywa program serwera (patrz rysunek Rys. 8), a drugi silnika szachowego po stronie klienckiej (patrz rysunek Rys. 9.).

Ze względu na duży rozmiar diagramów klas, informacje o typach danych i metodach zostały umieszczone w postaci tekstowej, poza diagramami. Typy danych poprzedzono punktorem •, natomiast metody punktorem ‣. Reszta zapisu jest zgodna ze specyfikacją *UML*.



Rys. 8. Diagram klas aplikacji serwera Flask

player

- - id : int
- - username : string
- - ELO : int
- - playing_as : char
- - time_spent_in_queue : int
- - current_scope : int

‣ + player(id : int, username : string, ELO : int, playing_as : char, time_spent_in_queue : int, current_scope : int)

game
- + gameID : int
- + game_mode_id : int
- + white_player : player
- + black_player : player
- - curr_FEN : string
- - timer : Timer
- + game(gameID : int, game_mode_id : int, white_player : player, black_player : player, timer : Timer)

ServerState
- - q_max_wait_time : int
- - initial_scope : int
- - scope_update_interval : int
- - games : List<game>
- - default_FEN : string
- - time_dialation : int
- - game_mode_times : pair(int,int)
- - game_mode_starting_FEN : pair(string,string)
- - queues : Dict<List<player>>
- - authorized_sockets : List<Socket>
- + ServerState()
- + get_player_from_queue(player_id : int, game_mode_id : int) : player
- + get_player_from_queue_by_id(player_id : int) : player
- + get_player_from_queue_by_sid(sid : string) : player
- + get_id_by_sid(sid : string) : int
- + get_is_player_in_game(player_id : int) : pair(game,char)

Mailing
- + Mailing()
- + get_qr_code(otp_secret : string) : BytesIO
- + send_welcome_message(receiver_mail : string, msg : string) : void
- + send_reset_password_token(username : string, receiver_mail : string, url : string) : void
- + send_qr_code(username : string, receiver_mail : string, otp_secret : string, code : string) : void
- - send_email(receiver_message : string, msg : string) : void

ChessDB
- + ChessDB()
- + create_db() : void
- + get_curr_date_time() : string
- + get_curr_date() : string
- + user_exists(data: string) : bool

- + add_recovery_codes (user_id: int, recovery_codes: List<string>) : void
- + update_login_times (user_id: int, last_login_time: string) : void
- + add_user(username : string, password : string email : string, is_2_fa_enabled : bool, otp_secret: string, ELO : int, ELO_dv : float, ELO_v : float, recovery_codes : List<string>) : void
- + remove_user(user_id : int) : void
- + activate_user_account(email : string) : void
- + add_game(w_id : int, w_score : float, b_id : int, parameter : float, win_type : char, moves : List<List<move>>) : int
- + add_move(game_id : int, Color : string, move_order : int, Move : move) : void
- + update_elo(user_id : int, new_ELO : int, new_ELO_dv : float, new_ELO_v : float) : void
- + update_user(user : User, new_user_data_json : dict, mail_service : Mailing) : void
- + update_password(new_password : string, Username : string) : void
- + update_scores(Color : string, game_id : int) : void
- + get_user_recovery_codes_by_id(user_id : int) : List<string>
- + get_user(Username : string) : List<user_info>
- + get_user_by_id(userId : int) : List<user_info>
- + get_user_by_email(email : string) : List<user_info>
- + get_participant(Color : string, GameID : int) : List<participant>
- + get_moves(GameID : int) : List<List<move>>
- + get_games(UserID : int, Start : int, End : int) : List<List<game>>
- + get_EloHistory(Username : string) : List<pair(string,int)>
- + count_games(Username : string) : int
- + count_wins(Username : string) : int
- + count_draws(Username : string) : int
- + count_losses(Username : string) : int
- + count_moves(gameID : int) : int

move

- + moveID : int
- + gameID : int
- + participantID : int
- + move_order : int
- + move : string
- + move(moveID : int, gameID : int, participantID : int, move_order : int, move : string)

participant

- + participantID : int
- + gameID : int
- + userID : int
- + score : float
- + color : string

- + currELO : int
- + participant(participantID : int, gameID : int, userID : int, score : float, color : string, currELO : int)

user_info
- + user_id : int
- + username : string
- + password : string
- + salt : string
- + email : string
- + 2FA : bool
- + OTPSecret : string
- + AccountConfirmed : bool
- + CreatedAt : string
- + UpdatedAt : string
- + LoggedInAt : string
- + LastLoggedInAt : string
- + ELO : int
- + ELODeviation : int
- + ELOVolatility : float
- + user_info(user_id : int, username : string, password : string, country : string, joined : string, ELO : int, ELODeviation : int, ELOVolatility : float)

Timer
- - white_time : int
- - black_time : int
- - last_move_timestamp : int
- + Timer()
- + update_timers(curr_turn : int) : char

REST_API
- - frontend_url : string
- - domain : string
- - debug_mode : bool
- - mail : Mailing
- - account_serializer : URLSafeTimedSerializer
- - app : Flask
- + REST_API()
- + generate_response(data : string, HTTP_code : string) : string
- + generate_sesion_token() : string
- + generate_refresh_token() : string
- + authorize_user(user_id : int, session_token : string) : bool

- + verify_ewcaptcha(token : string) : bool
- + login() : string
- + refresh_session() : string
- + logout() : string
- + refresh_session() : string
- + login() : string
- + register() : string
- + check_2_fa() : string
- + delete_user() : string
- + get_user_details() : string
- + update_user() : string
- + resent_activation_email() : string
- + confirm_email(token: string) : string
- + reset() : string
- + forgot_password() : string
- + get_2fa_code() : string
- + is_in_game() : string
- + get_game_info() : string
- + get_player_stats() : string
- + generate_example_match_data() : pair(string,string)
- + get_history() : string

Websockets

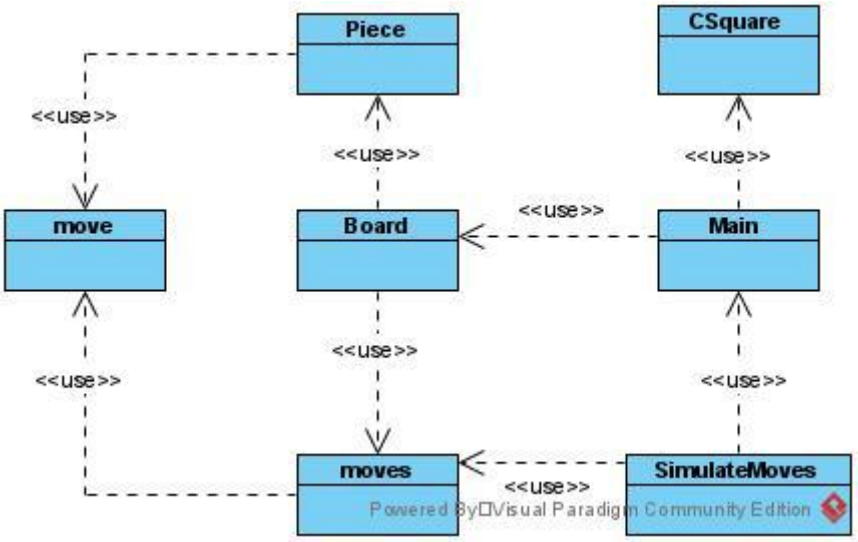
- - app : Flask
- - socketio : SocketIO
- - thread : background_task
- - debug_mode : bool
- + Websockets()
- + check_auth(sid : string, player_id : int) : bool
- + connect() : void
- + disconnect() : void
- + authorize(data : string) : void
- + join_queue(data : string) : void
- + leave_queue(data : string) : void
- + match_maker() : void
- + increment_wait_time(game_mode_id : int, time_taken : int) : void
- + find_match(game_mode_id : int, player : Flask_Serwer.player) : void
- + finish_game(game_info : string, win_color : string) : void
- + surrender(data : string) : void
- + send_chat_to_server(data : string) : void

RatingSystem

- - starting_ELO : int
- - starting_ELO_deviation : int
- - starting_ELO_volatility : float
- + RatingSystem()
- + calculate_elo(winner : float, loser : float, draw : float) : float
- + calculate_glicko(player1_rating : float, player1_RD : float, player1_vol : float, player2_rating : float, player2_RD : float, player2_vol : float, result : float) : pair(int,int)

ChessLogic

- - board : Board
- + ChessLogic()
- + generate_pos_to_stock_not_dict() : string
- + is_valid_move(FEN : string, startSquare : string, targetSquare : string) : bool
- + is_checkmate(FEN : string) : string
- + update_fen_with_turn_info(FEN : string, player_to_move : string) : string
- + get_best_move(FEN : string) : string



Rys. 9. Diagram klas maszyny szachowej

Piece

- - color : char
- - type : char
- - X : int
- - Y : int
- - gridPos : int
- - possibleMoves : List<move>
- - scaledSize : int
- + Piece(type : char, x : int, y : int)

- + get_grid_pos() : int
- + drag() : void
- + movePiece() : bool
- + get_closest_position() : int
- + snap() : void
- + snap_back() : void
- + draw_piece() : void
- + isIntersecting() : bool
- + get_taken() : void

Board

- - FEN : string
- - color_to_move : char
- - lastPawnMoveOrCapture : int
- - lastmove : pair(int,int)
- -numofMoves : int
- - check : int
- - enPassant : string
- - SetupState : int
- + Board()
- + get_pos(i : int, j : int) : pair(int,int)
- + set_FEN_from_grid() : void
- + set_FEN_by_move(StartingSquare : int, TargetSquare : int) : void
- + set_FEN_by_rejected_move(StartingSquare : int, TargetSquare : int) : void
- + load_FEN() : void
- + draw_board() : void
- + change_turn() : void

move

- - startSquare : int
- - endSquare : int
- - type : char
- + move(starting_square : int, ending_square : int, type : char)

moves

- + count_squares_to_edge() : int
- + Generate_moves(grid : List<int>) : List<move>
- + Generate_opponent_moves(grid : List<int>) : List<move>
- + Get_Pawn_moves(startSquare : int, piece : Piece, grid : List<char>) : void
- + check_if_promotion(piece : Piece, targetsquare : int) : bool
- + is_square_save(targetSquare : int) : int
- + Get_king_moves(startSquare : int, piece : Piece, grid : List<char>) : void

‣ + Get_Knight_moves(startSquare : int, piece : Piece, grid : List<char>) : void
‣ + Get_long_moves(startSquare : int, piece : Piece, grid : List<char>) : void
‣ + get_move(StartSquare : int, TargetSquare : int) : move
‣ + check_move(StartSquare : int, TargetSquare : int) : move
‣ + get_pixel_position_from_pixel_position_array(pos : int) : void
‣ + Distance_between_points(x1 : int, y1 : int, x2 : int, y2 : int) : void
‣ + check_if_check() : void
‣ + get_white_king_pos() : int
‣ + get_black_king_pos() : int

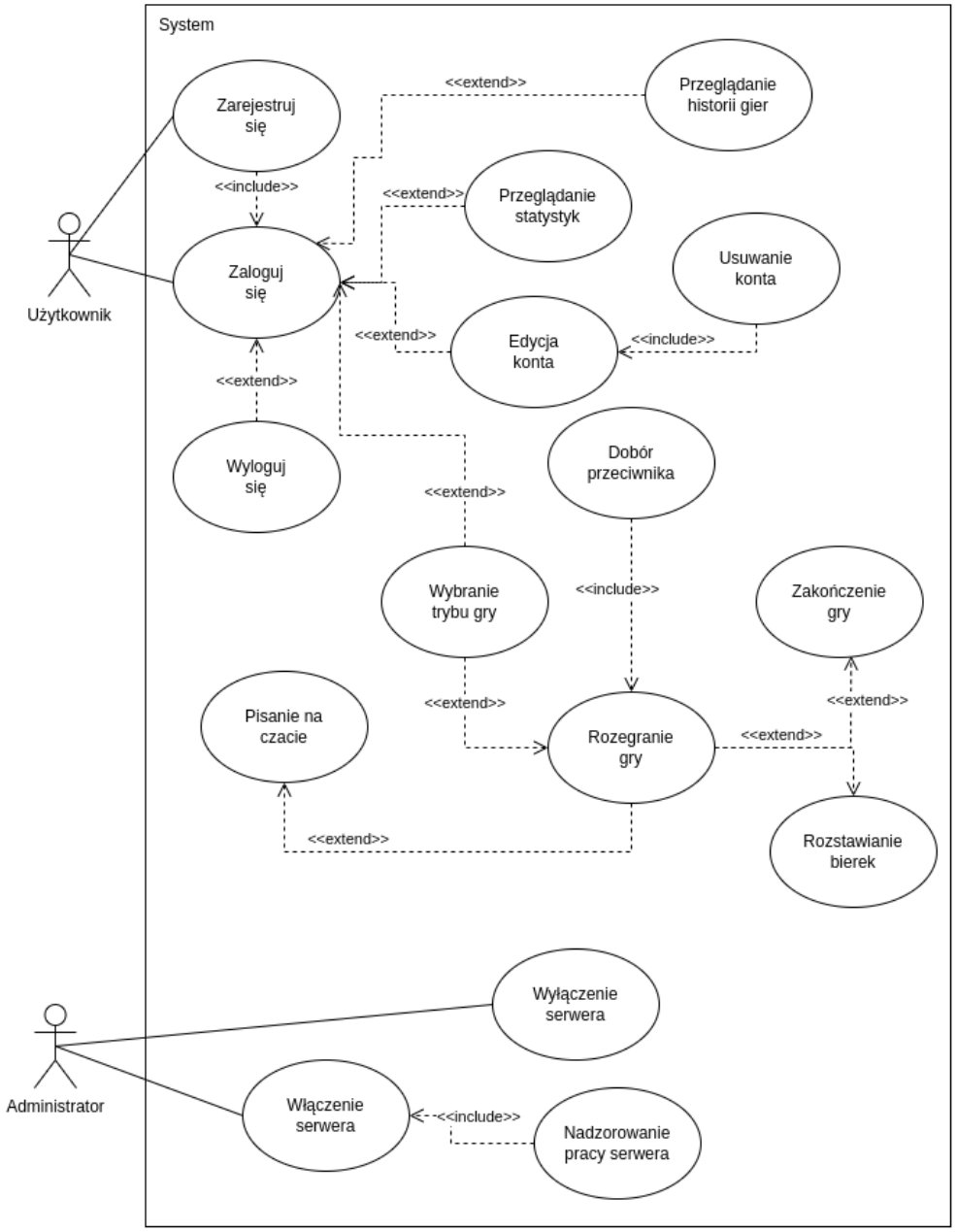
SimulateMoves
‣ + simulate_moves_for_ally(grid : List<char>) : void
‣ + simulate_white_king_pos(grid : List<char>) : int
‣ + simulate_black_king_pos(grid : List<char>) : int
‣ + simulate_get_taken(piece : Piece) : void
‣ + simulate_set_grid_by_move(startingSquare : int, TargetSquare : int, old_grid : List<char>) : void
‣ + fin_move_in_moves_for_simulation(move : Move, ally_moves : List<Maszyna_szachowa.moves>) : int

CSquare
• - size : int
• - i : int
• - j : int
• - x : int
• - y : int
• - x2 : int
• - y2 : int
• - parityFlag : int
‣ + CSquare(x : int.y:int, size : int)
‣ + drawSquares() : void

Main
‣ + importAll() : void
‣ + sketch() : void
‣ + readjustCanvas() : void

4.3.2 Diagram przypadków użycia

Poniżej przedstawiono diagram przypadków użycia opracowanej aplikacji klienckiej (patrz rysunek Rys. 10.).



Rys. 10. Diagram przypadków użycia aplikacji klienckiej

W tabeli 8, 9 oraz 10 zostały szczegółowo omówione wybrane przypadki użycia dla prowadzenia rozgrywki, aktualizacji danych konta użytkownika oraz rozstawianie bierek w trybie gry *ChessDefender*.

Tabela 8. Przypadek użycia - przeprowadzenie rozgrywki

UC1: Przeprowadzenie rozgrywki
Aktor: Gracz
Główny scenariusz 1. Gracz wybiera interesujący go tryb rozgrywki. 2. Gracz zostaje dodany do kolejki oczekiwania na dobór przeciwnika. 3. Gracz zostaje dobrany z przeciwnikiem o podobnych umiejętnościach. 4. Gracz przechodzi do widoku rozgrywki. 5. Gracz przeprowadza rozgrywkę. 6. Gracz kończy rozgrywkę.
Rozszerzenie 2.1 W przypadku braku przeciwnika gracz nie może rozpocząć rozgrywki. 2.2 W przypadku wyboru trybu jednoosobowego rozgrywka rozpoczynana jest natychmiast 4.1 Gracz podczas rozgrywki może korzystać z czatu tekstowego. 4.2 Gracz podczas rozgrywki może zaproponować remis. 4.2.1 Gracz któremu zaproponowano remis może go przyjąć bądź odrzucić. 4.3 Gracz podczas rozgrywki może się poddać.

Tabela 9. Przypadek użycia - aktualizacja danych

UC2: Aktualizacja danych użytkownika
Aktor: Zalogowany użytkownik
Główny scenariusz 1. Aktor przechodzi do widoku edycji konta. 2. Aktor dokonuje aktualizacji interesujących do pozycji. 3. Aktor ponownie wpisuje wymagane dane. 4. Aktor podaje aktualne hasło do konta lub hasło jednorazowe jeżeli jest wymagane. 5. Aktor potwierdza aktualizację danych.

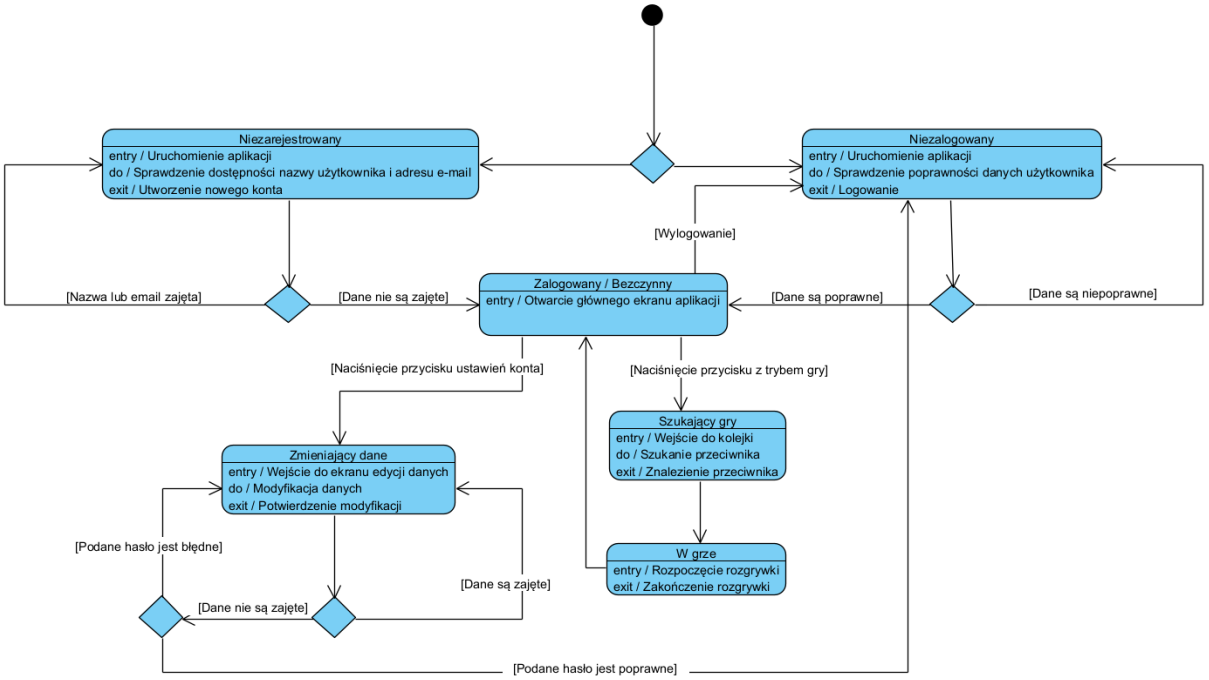
Rozszerzenie
3.1 W przypadku braku ponownego wpisania wymaganych danych, zostanie poproszony o ponowne ich wpisanie - powrót do punktu 2. 4.1 W przypadku podania błędnego hasła lub kodu jednorazowego, aktor zostaje poproszony o ponowne wpisanie wymaganych danych - powrót do punktu 4. 5. W przypadku próby użycia zajętych danych aktor zostaje poproszony o podanie innych danych - powrót do punktu 2.

Tabela 10. Przypadek użycia - pierwsza faza rozgrywki *ChessDefender*

UC3: Pierwsza faza rozgrywki w trybie *ChessDefender*
Aktor: Gracz
Główny scenariusz 1. Gracz wybiera bierkę 2. Gracz ustawia bierkę na wybranym polu. 3. Gracz czeka na swoją kolej. 4. Gracz rozpoczyna fazę drugą rozgrywki.
Rozszerzenie 1.1 W przypadku wyboru bierki, której koszt przewyższa dostępne punkty, bierka powraca na pole wyboru - powrót do punktu 1. 2.1 Jeżeli pole jest już zajęte lub nie znajduje się w dostępnych dla gracza polach, bierka powraca na pole wyboru - powrót do punktu 1. 3.1 W przypadku próby postawienia bierki podczas tury przeciwnika, czynność zostanie cofnięta. 4.1 Aktor podczas rozgrywki może korzystać z czatu tekstowego.

4.3.3 Diagram stanów

Poniżej przedstawiono diagram stanów użytkownika (patrz rysunek Rys. 11).

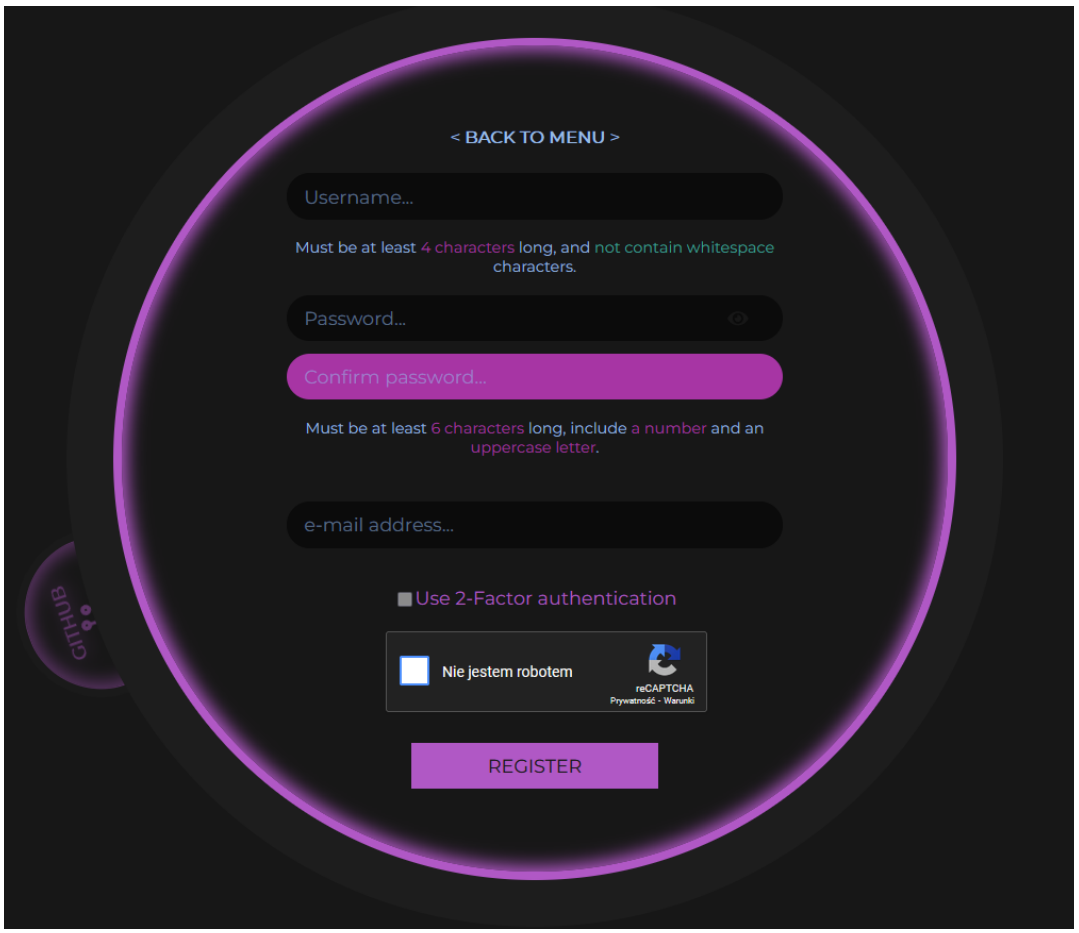


Rys. 11. Diagram stanów użytkownika

4.4 Tworzenie konta

Po przejściu do procesu tworzenia nowego konta, użytkownik podaje nazwę użytkownika, hasło do konta oraz adres e-mail (patrz rysunek Rys. 12). Podczas uzupełniania w.w. informacji, użytkownik jest zobligowany spełnić określoną politykę. W przypadku nazwy użytkownika, nie może ona zawierać białych znaków, a jej długość nie może być mniejsza niż 4 znaki. W aplikacji została użyta polityka hasła, która mówi, iż wprowadzone hasło musi składać się z minimalnie 6 znaków, co najmniej jednej wielkiej litery oraz co najmniej jednej cyfry. Użytkownik może skorzystać z weryfikacji dwuetapowej, która zostanie omówiona w dalszej części pracy (patrz sekcja 4.8).

Mechanizm tworzenia nowego konta zabezpieczony został za pomocą mechanizmu *CAPTCHA*, którego celem jest zapewnienie, że dane wprowadzane są przez człowieka, a nie przez program komputerowy. Mechanizm ten umożliwia zabezpieczenie aplikacji przed tworzeniem fałszywych kont. Pomyślna weryfikacja mechanizmu pozwala utworzyć nowe konto. Warto wspomnieć, iż użytkownik ma określony czas, po którym walidacja mechanizmu *CAPTCHA* wygaśnie i konieczne będzie ponowne jego zaznaczenie.



Rys. 12. Strona tworzenia nowego konta

Jeżeli wprowadzone pola spełniają wymagania, aplikacja dokonuje tworzenia nowego konta, które domyślnie jest nieaktywne. Następnie wprowadzone dane są zapisywane do bazy danych w celu późniejszej weryfikacji podczas logowania. Wprowadzone hasło jest przesyłane do serwera w postaci skrótu uzyskanego za pomocą SHA256, do którego dodawana jest 12 znakowa sól. Tak przygotowane hasło jest ponownie haszowane po stronie serwera i zapisywane do bazy danych.

Dodanie soli do hasła ma za zadanie zmusić ew. atakującego, aby dla każdego z haseł ponownie dokonał obliczeń w postaci ponownego haszowana hasła wraz z dodaną solą, ponieważ gdy dwoje użytkowników ma takie same hasła, w bazie danych są one różne. W tabeli 2 pokazany jest właśnie takie przykład. Dwoje pierwszych użytkowników podało identyczne hasła podczas rejestracji, a ich reprezentacja w bazie danych jest całkowicie inna.

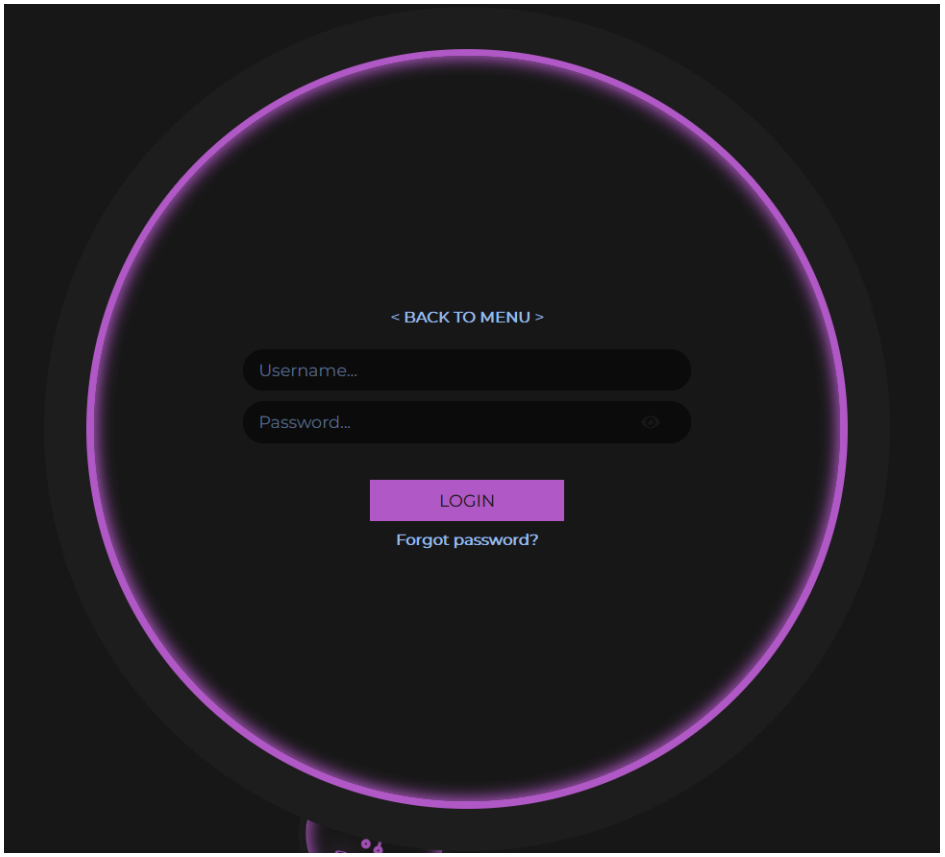
Wiele osób używa tego samego hasła do wielu serwisów. Z tego powodu, w aplikacji hasło haszuje się po stronie klienta oraz po stronie serwera. Haszowanie hasła po stronie klienta, ma na celu uniknięcie przesyłania hasła w formie jawnej. Haszowanie po stronie serwera, uniemożliwia potencjalnie niebezpieczne zarejestrowanie się użytkownika przy pominięciu aplikacji klienckiej.

W przypadku błędu podczas procesu wynikającego np. z zajętości wprowadzonej nazwy użytkownika bądź adresu e-mail, wprowadzenia hasła niespełniającego użytej polityki haseł, itp., system poinformuje użytkownika o błędzie stosownym komunikatem.

Po pomyślnym utworzeniu konta, system wysyła do użytkownika wiadomość e-mail zawierającą link, dzięki któremu użytkownik może aktywować swoje konto.

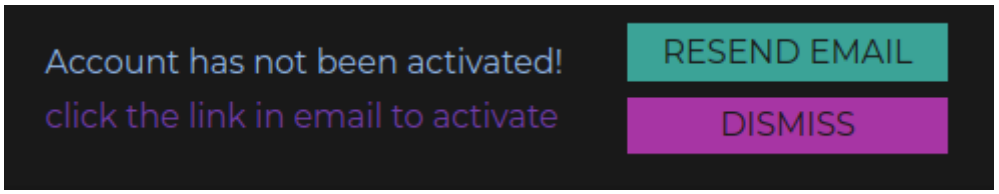
4.5 Logowanie

Proces logowania do konta polega na podaniu przez użytkownika podstawowych, wymaganych danych, tj. nazwy użytkownika oraz hasła (patrz rysunek Rys. 13). Po wprowadzeniu poprawnych danych i próbie zalogowania się na konto, system sprawdza czy konto o podanej nazwie użytkownika zostało aktywowane, tj. po rejestracji konta, użytkownik nacisnął na link wysłany w wiadomości.



Rys. 13. Ekran logowania

Jeżeli konto nie jest aktywne, system wyświetla odpowiedni komunikat (patrz rysunek Rys. 14) i umożliwia ponowne wysłanie linku umożliwiającego aktywację konta. Proces aktywacji został opisany w dalszej części pracy (patrz sekcja 4.6).



Rys. 14. Próba logowania na nieaktywne konto

Jeżeli konto zostało aktywowane, system sprawdza, czy konto używa weryfikacji dwuetapowej. Proces sprawdzania zostanie wykonany tylko po pobraniu odpowiednich danych. W przeciwnym razie, system poinformuje użytkownika o błędnych danych.

Jeżeli użytkownik wyraził chęć korzystania z weryfikacji dwuetapowej podczas tworzenia konta lub podczas jego edycji, system oczekuje od użytkownika podania kodu jednorazowego generowanego przez odpowiednią aplikację bądź kodu odzyskiwania. Jeżeli kod jest poprawny, system pomyślnie loguje użytkownika do systemu.

4.6 Aktywacja konta

Nowo utworzone konto jest nieaktywne. Podczas próby zalogowania się na takie konto, system poinformuje użytkownika o tym fakcie (patrz rysunek Rys. 14) i nie pozwala na logowanie użytkownika do systemu. Przed zakończeniem procesu tworzenia konta, system wysyła, na podany przez użytkownika adres e-mail wiadomość zawierającą odpowiedni link umożliwiający jego aktywowanie (patrz sekcja 4.10). Wysłany link zawiera odpowiednia ciąg znaków (ang. *token*), który jest identyfikowany po stronie stronie serwera. Token wysłany w wiadomości jest aktywny przez czas wynoszący jedną godzinę. Po tym czasie próba aktywowania konta tokenem, zakończy się niepowodzeniem, a zainteresowany użytkownik będzie musiał zażądać nowego tokenu.

Token zawiera zakodowany adres e-mail wraz ze znacznikiem czasu umożliwiającym określenie czy token jest aktywny. Jest on otrzymywany przez hashowanie adresu e-mail wraz z dodaną solą. Podczas aktywowania konta, z dostarczonego tokenu odzyskiwany jest adres e-mail użyty podczas rejestracji. W przypadku wygaśnięcia tokenu, adres e-mail nie zostanie odzyskany, a użytkownik będzie musiał ponownie uzyskać odpowiedni token.

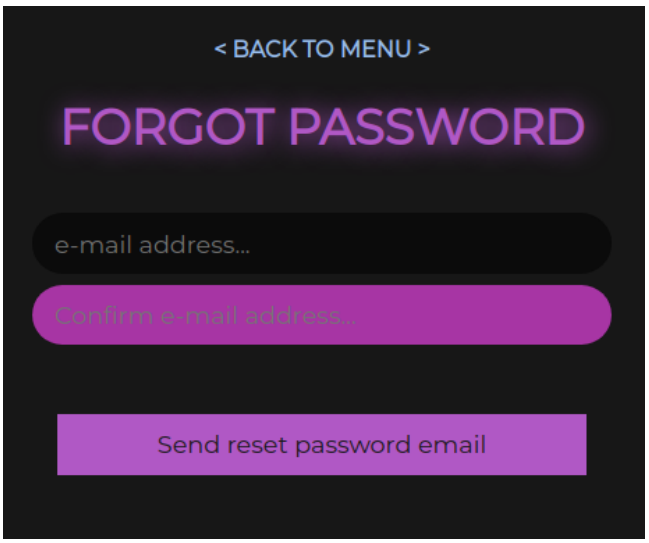
Po odczytaniu adresu e-mail z otrzymanego tokenu, konto o podanym adresie jest aktywowane, tj. flaga aktywacji danego konta w bazie danych ustawiana jest na wartość logiczną 1. Poniżej został zamieszczony przykładowy link pozwalający na aktywację konta.

http://neochess.pl/confirm/Im1hcmVrbWFyY3pld3NraTEyMzRAZ21haWwuY29tIg.YfBNTw.jKHXBbQqB-xJWWuZwQvfy6pyIQM

4.7 Odzyskiwanie hasła

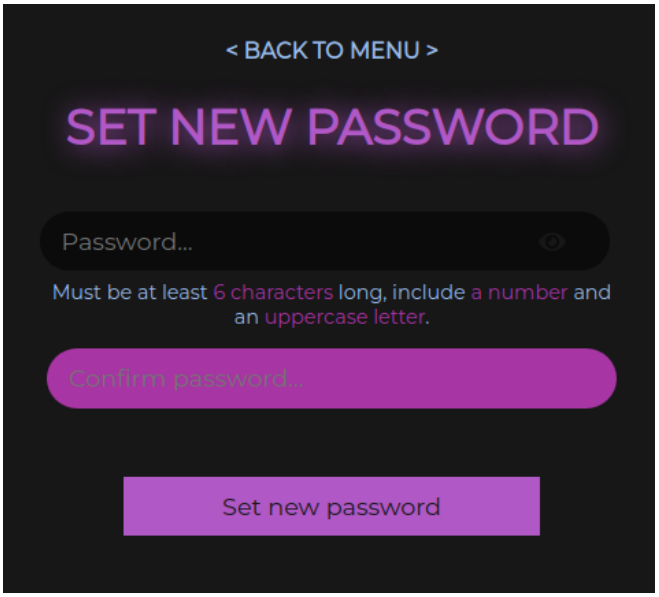
W przypadku zapomnienia hasła do konta wcześniej utworzonego na serwisie, użytkownik ma możliwość jego ponownego ustawienia. W tym celu należy w odpowiednim miejscu (patrz rysunek Rys. 15) podać adres e-mail, aby system mógł wysłać wiadomość zawierającą link prowadzący do strony poświęconej resetowaniu hasła (patrz sekcja 4.10).

Jeżeli użytkownik poda adres e-mail, który nie istnieje w bazie danych, system nie poinformuje użytkownika o tym fakcie. W ten sposób, osoba niepowołana nie zostanie poinformowana o fakcie istnienia danego adresu e-mail w bazie danych, co może wyeliminować próbę włamania się na skrzynkę pocztową danego użytkownika w celu przejęcia dostępu do konta.



Rys. 15. Podstrona wysyłania wiadomości

Wysłana wiadomość zawiera odpowiedni token, pozwalający na identyfikację konta, którego hasło ma zostać zresetowane. Podobnie jak w przypadku potwierdzania konta, token posiada swoją żywotność, określoną na jedną godzinę, po której jest on nieważny. Proces jego tworzenia jest analogiczny jak w przypadku aktywacji konta (patrz sekcja 4.6).



Rys. 16. Podstrona resetowania hasła

Link zawierający token przekierowuje użytkownika na podstronę do ustawiania nowego hasła (patrz rysunek Rys. 16). Aplikacja uniemożliwia użytkownikowi udanie się na tę stronę bez odpowiedniego tokenu. Następnie wraz z nowym hasłem będącym skrótem, jest on przesyłany do serwera w celu weryfikacji. Jeżeli dostarczony token się zgadza, tzn. jest aktywny, hasło w bazie danych jest zastępowane nowym wraz z nową solą. Od tego momentu użytkownik może logować się do konta za pomocą nowego hasła.

4.8 Weryfikacja dwuetapowa

Użytkownik podczas tworzenia nowego konta w serwisie ma możliwość użycia weryfikacji dwuetapowej (ang. t*wo-factor authentication*, *2FA*). Dzięki zastosowaniu tego mechanizmu, dostęp do konta jest bardziej zabezpieczony przed osobami niepowołanymi np. w przypadku wycieku bazy danych zawierających hasło do konta.

W trakcie rejestracji, po wyrażeniu chęci skorzystania z weryfikacji dwuetapowej, na podstawie podanego adresu e-mail, aplikacja generuje kod *QR* oraz specjalny kod, który jest wysyłany na podany adres. Użytkownik może zeskanować kod *QR* lub dostarczony kod, w aplikacji pozwalającej na generowanie kodów jednorazowych (ang. *one-time password, OTP*) np. *Google Authenticator*. Aplikacja wymusza na użytkowniku wprowadzenie poprawnego kodu jednorazowego podczas rejestracji.

Przy logowaniu użytkownik mający aktywną opcję weryfikacji dwuetapowej, zobligowany jest do podania odpowiedniego kodu. W przypadku błędnego kodu, próba logowania zostaje zakończona niepowodzeniem.

Rys. 17 Przykładowe kody odzyskiwania

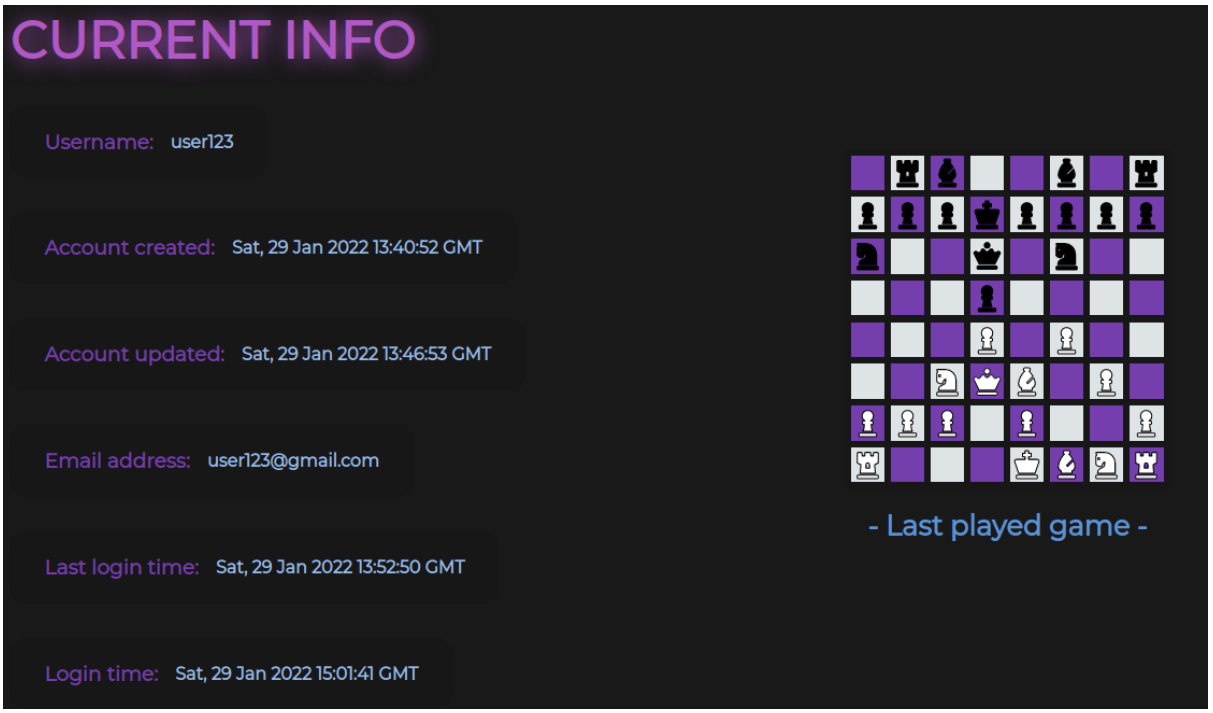
Aplikacja umożliwia zalogowanie się do konta w przypadku utraty możliwości generowania kodów jednorazowych za pomocą tzw. kodów odzyskiwania (and. *recovery codes*). Podczas aktywacji weryfikacji dwuetapowej, aplikacja generuje 16 losowych kodów o długości 16 znaków, składających się z liter, cyfr oraz znaków specjalnych (patrz rysunek Rys. 17), które pozwalają na logowanie się do konta w przypadku utraty możliwości generowania kodów jednorazowych. Po wygenerowaniu kodów odzyskiwania, aplikacja informuje użytkownika o takiej opcji i prosi o ich zapisanie w bezpiecznym miejscu, ponieważ system nie umożliwia ponownego ich ukazania. W przypadku wycieku takich kodów, osoby niepowołane otrzymają dostęp do konta. Tak wygenerowane kody są przesyłane do serwera w postaci skrótu i zapisywane do bazy danych. Podczas próby logowania podany kod jednorazowy lub kod odzyskiwania jest weryfikowany przez serwer a następnie użytkownika jest logowany lub informowany o wystąpieniu błędu. W przypadku utraty możliwości generowania kodów jednorazowych, użytkownik wpisuje jeden z kodów odzyskiwania w pole przeznaczone do wpisania kodu jednorazowego.

4.9 Zarządzanie kontem

System umożliwia użytkownikowi zarządzanie oraz wgląd w swoje dane. Poszczególne operacje zostaną opisane w podrozdziałach 4.9.1, 4.9.2 oraz 4.9.3.

4.9.1 Wyświetlanie danych

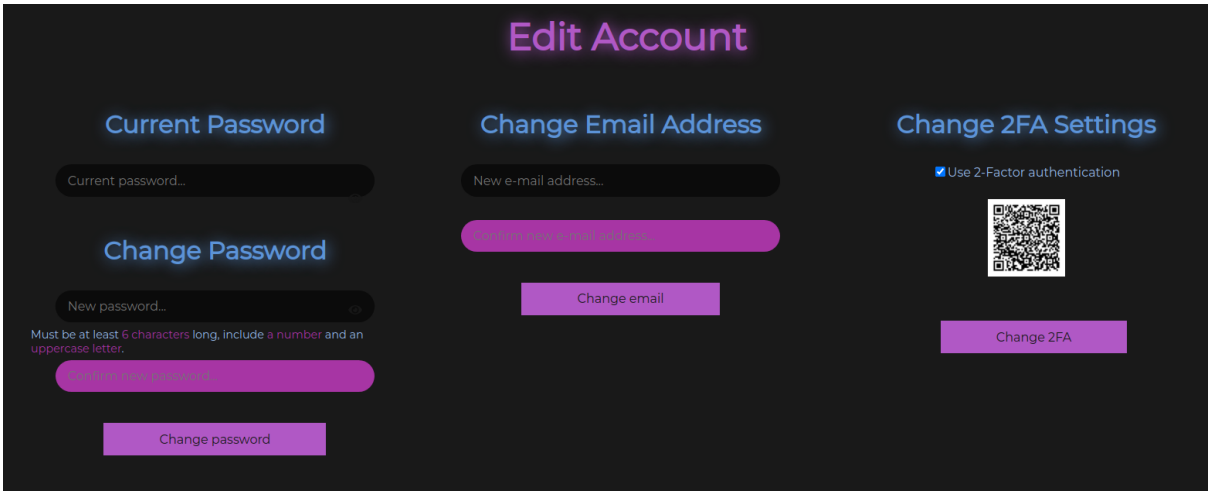
Po przejściu w ustawienia konta, użytkownika ma możliwość wgląd w dane na temat konta. Są to: nazwa użytkownika, czas i data utworzenia, ostatniej modyfikacji danych konta, aktualny adres poczty elektronicznej, czas i datę ostatnie logowania oraz wgląd w ostatnią przeprowadzoną rozgrywkę w postaci rozłożenia bierek na planszy (patrz rysunek Rys. 18).



Rys. 18 Dane użytkownika

4.9.2 Edycja konta

Użytkownik ma możliwość edycji podstawowych danych dotyczących swojego konta takich jak: hasła, adresu e-mail oraz aktywację bądź dezaktywację weryfikacji dwuetapowej (patrz rysunek Rys. 19). Aby możliwe było dokonanie jakiejkolwiek modyfikacji konta, użytkownik zobligowany jest podać aktualne hasło.

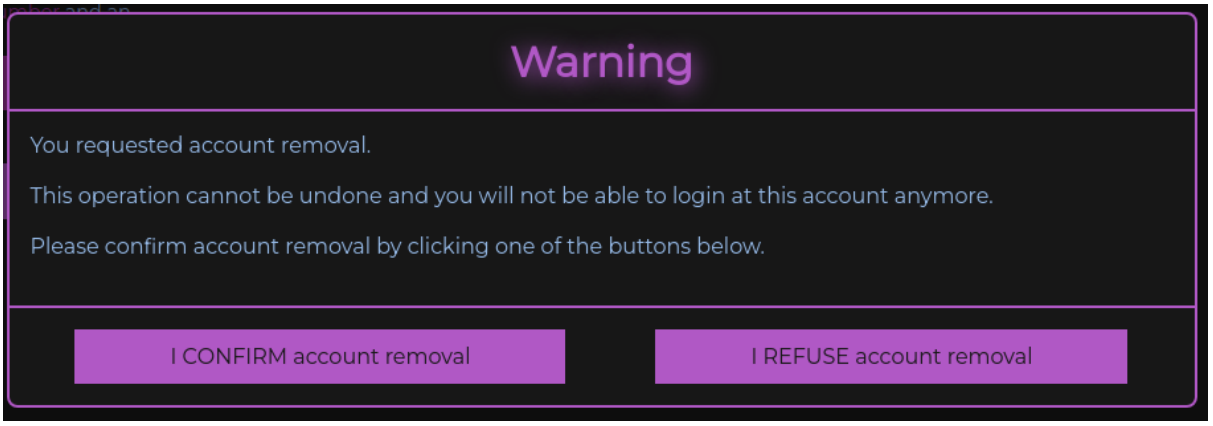


Rys. 19 Edycja danych użytkownika

W przypadku zmiany adresu e-mail, w pierwszym kroku jest on weryfikowany pod kątem poprawności, a następnie, czy nie jest on już używany przez inne konto. Jeżeli weryfikacja przebiegnie pomyślnie, poprzednia wartość jest nadpisywana, a konto jest dezaktywowane. Należy zaznaczyć, że dezaktywacja konta nie oznacza usunięcia danych użytkownika. W ostatnim kroku, system wysyła na nowy adres wiadomość pozwalającą na ponowną aktywację konta.

4.9.3 Usuwanie konta z serwisu

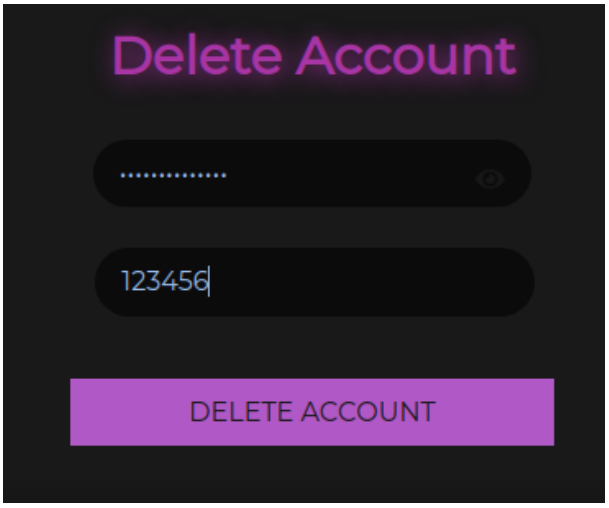
Zgodnie z artykułem 17 RODO [14], użytkownik ma "prawo do bycia zapomnianym", czyli ma mieć możliwość usunięcia swoich danych z bazy danych. Usunięcie konta jest procesem nieodwracalnym i wymaga dodatkowego potwierdzenia (patrz rysunek Rys. 20), aby konto nie zostało usunięte przez przypadek lub przez osoby do tego nieuprawnione.



Rys. 20 Ostrzeżenie podczas próby usunięcia konta użytkownika

Na stronie umożliwiającej zarządzanie kontem użytkownik ma możliwość jego usunięcia. Aplikacja wymusza podanie przez użytkownika obecnego hasła oraz, jeżeli jest aktywna weryfikacja dwuetapowa, kod jednorazowy wygenerowany przez aplikację (patrz rysunek Rys. 21). Jeżeli wprowadzone dane są poprawne, konto użytkownika oraz wszystkie

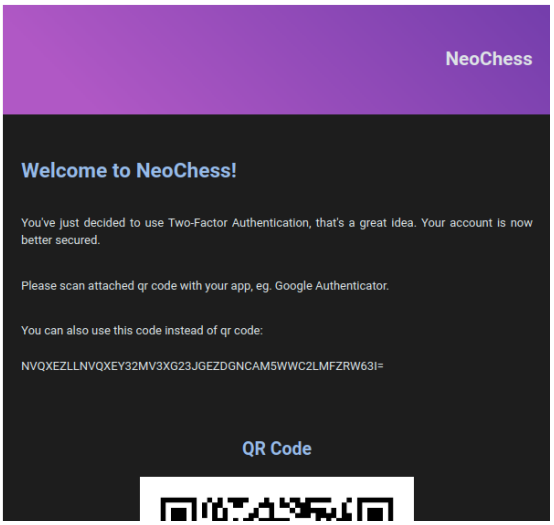
powiązane z nim dane są usuwane z systemu, a ponowne zalogowanie się na to konto nie jest możliwe.



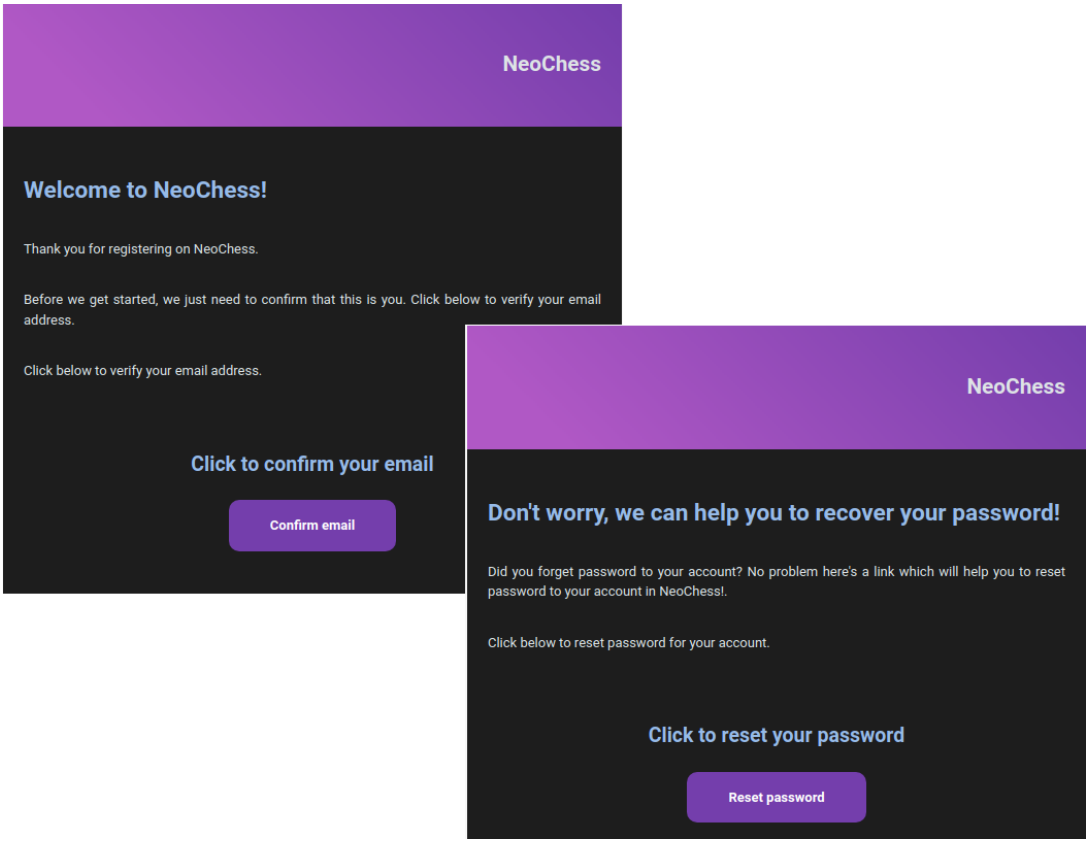
Rys. 21. Pole umożliwiające usunięcie konta z serwisu

4.10 Generowanie i wysyłanie wiadomości e-mail

System wykorzystuje protokół SMTP [5] w celu wysyłania wiadomości za pośrednictwem poczty elektronicznej w celu poinformowania bądź wykonania danej akcji przez użytkowników. Wysyłane wiadomości służą m.in. w celu aktywacji konta (patrz sekcja 4.6), odzyskiwania hasła (patrz sekcja 4.7) oraz informowania użytkownika o aktywacji weryfikacji dwuetapowej (patrz sekcja 4.8). Szablon wiadomości jest generowany na podstawie szablonu *HTML*, który następnie jest dołączany do ciała wiadomości. Link przekierowujący na podstronę pozwalająca wykonać poszczególne akcje, jest dołączony w postaci przycisku, poprawiającego estetykę wysyłanej wiadomości. Następnie dodawany jest adresat, nadawca oraz temat. Tak przygotowana wiadomość jest wysyłana na podany adres e-mial. Na rysunku 22 oraz 23 zostały pokazane przykładowe wiadomości wygenerowane przez system.



Rys. 22 Przykładowa wiadomość e-mail pozwalająca na aktywację *2FA*



Rys. 23 Przykładowe wiadomości e-mail pozwalające na aktywację konta oraz odzyskiwanie hasła

4.11 Uwierzytelnianie

Uwierzytelnianiem w kontekście projektu będzie nazywany proces weryfikacji czy dany użytkownik powinien mieć dostęp do konkretnego zasobu (np. historii gier) lub wykonania danej akcji (np. dokonania ruchu szachowego w danej grze).

W procesie uwierzytelniania wykorzystywane są dwa 256-bitowe ciągi znaków zwane tokenami (ang. *token*):

- token sesyjny (ang. *session token*) - identyfikuje sesję użytkownika, jego okazanie jest wystarczające do potwierdzenia, że użytkownik jest rzeczywiście użytkownikiem o danym identyfikatorze. Token sesyjny otrzymywany jest po poprawnym zalogowaniu lub odświeżeniu przy użyciu tokena odświeżania.
- token odświeżenia (ang. *refresh token)* - pozwala na otrzymanie nowego tokena sesyjnego bez potrzeby ponownego logowania.

Podobny system jest wykorzystywany w popularnym standardzie autoryzacji Auth0 [12]. W ewentualności wygaśnięcia tokena sesyjnego, użycie tokena odświeżania pozwala na podtrzymanie sesji bez konieczności ponownego uzupełniania formy logowania.

Po stronie serwera wartości tokenów przechowywane są dla każdego użytkownika w słowniku *sessions* (patrz sekcja 4.1.1). Wartość tokena odświeżania jest losowo generowana po każdym logowaniu. Tokeny są skrótami liczb losowo generowanych przez

serwer. Poniżej przedstawiono fragment kodu odpowiedzialny za wygenerowanie nowego tokena.

```
def generate_token():
   #generate large random number
   n = random.randint(1000000000000, 9999999999999)
   #hash the number with sha256
   n = hashlib.sha256(str(n).encode())
   #return string form of the hash
   return str(n.hexdigest())
```

Proces autoryzacji po stronie serwera polega więc na porównaniu wpisu w *sessions,* dla klucza będącego identyfikatorem użytkownika, z tokenem sesyjnym okazanym przez użytkownika o danym identyfikatorze.

Po stronie klienta, token odświeżania jest przechowywany w ciasteczku z ustawioną flagą *HTTP-ONLY*. Ustawienie tej flagi oznacza, że ciasteczko nie jest dostępne z poziomu *Javascript*, utrudniając ataki XSS [12]. Ciasteczko ma także ustawioną własność *path*, która ogranicza wysłanie ciasteczka jedynie do *endpointu* 'refresh_session' odpowiedzialnego za tworzenie i dystrybucję tokenów sesyjnych (patrz sekcja 4.15.1). Ustawienie własności *path* zmniejsza szanse na przechwycenie tokena odświeżenia, ponieważ bez ustawienia tej opcji ciasteczko go zawierające byłoby wysyłane z każdym zapytaniem HTTP.

Token sesyjny w aplikacji klienckiej jest przechowywany przy pomocy biblioteki Redux [14]. Aby uwierzytelnić swoje zapytanie, klient załącza wartość tokena sesyjnego w nagłówku *'Authorization'* zapytania wymagającego uwierzytelnienia (zapytania wymagające uwierzytelnienia opisane w sekcji 4.15.1).

Użytkownicy aplikacji mają możliwość uwierzytelnienia się za pomocą weryfikacji dwuetapowej, która została szczegółowo opisana w podrozdziale 4.8.

4.12 Dobór przeciwnika

Parowanie graczy do rozgrywki wieloosobowej (przy rozgrywce z komputerem przeciwnik nie musi być dobierany) odbywa się po stronie serwera. Przy starcie serwera, dla każdego trybu rozgrywki na osobnym wątku, uruchamiana jest funkcja *match_maker*. Funkcja ta iteruje w nieskończonej pętli po tablicy znajdującej się na pozycji o wartości *id* trybu gry w słowniku *queue* (patrz sekcja 4.1.1). Tablice te zawierają obiekty *Player* (patrz sekcja 4.1.1), reprezentujące graczy czekających w kolejce dla danego trybu gry. Dla identyfikatora każdego z graczy wywoływana jest funkcja *find_match(player_id)*, próbująca dopasować przeciwnika do danego gracza. Po każdorazowym przejściu przez pętlę w *match_maker* wywoływana jest także funkcja *increment_wait_time()*, zwiększająca czas oczekiwania każdego gracza w kolejce.

W funkcji *find_match(player_id)* obliczana jest maksymalna różnica w punktacjach ELO graczy, którzy mogą być dobrani do wspólnej gry (ang. *scope*, dalej nazywana zakresem

poszukiwań). Dla danego gracza zakres poszukiwań rośnie wraz z czasem, jaki dany gracz spędził w kolejce. Poniżej przedstawiono wzór użyty przy obliczaniu zakresu poszukiwań.[4]

```
scope = initial_scope + int(player_wait_time / scope_update_interval) *
scope_update_ammount
```

Dalej w funkcji *find_match(player_id)* rozważany gracz porównywany jest z potencjalnymi przeciwnikami z tablicy *queue*. Jeżeli zakresy poszukiwań przeciwnika porównywanych graczy na siebie nachodzą gracze zostają usunięci z kolejki. Następnie serwer losuje, który z nich gra jako biały. Później gra dodawana jest do bazy danych, a gracze są informowani o znalezieniu gry.

Aby rozpocząć proces doboru przeciwnika do rozgrywki użytkownik aplikacji klienckiej musi skorzystać z widżetu o tytule "*FIND A GAME*", klikając na ikonę wybranego trybu gry. Podczas procesu doboru przeciwnika w aplikacji klienckiej wyświetlany jest aktualny czas oczekiwania gracza (ang. *wait time*), zakres poszukiwań (ang. *scope*) oraz liczba graczy także oczekujących w kolejce (ang. *players in queue*) (patrz rysunek Rys. 24).

Rys. 24. Interfejs użytkownika podczas poszukiwania przeciwnika do trybu classic

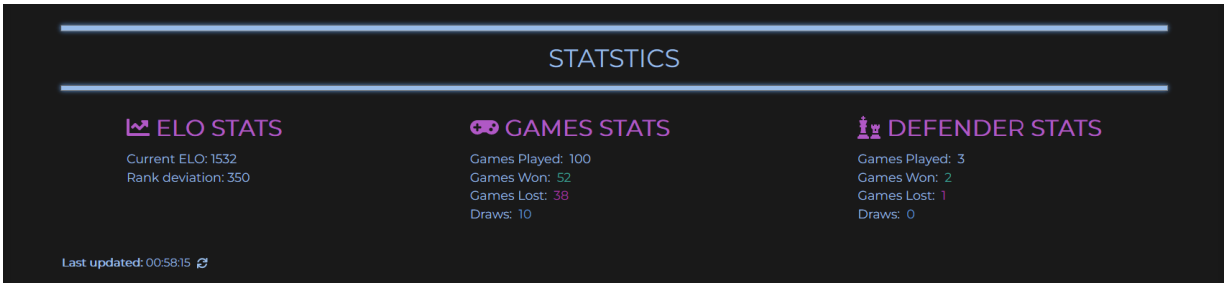
Dokładny sposób, w jaki serwer jest informowany o dołączeniu przez klienta do kolejki oraz sposób, w jaki klient jest informowany o aktualizacji zmiennej zakresu poszukiwań oraz znalezieniu przeciwnika, został opisany w dalszej części pracy (patrz sekcja 4.15.2).

[4] dokładne znaczenie zmiennych rozwinięte w sekcji 4.1.1 Struktury danych po stronie serwera - struktura ServerState

4.13 Wgląd w statystyki użytkownika

Na statystyki użytkownika składają się liczba punktów ELO, odchylenie rankingowe, liczby przegranych, wygranych i zremisowanych gier w każdym z dostępnych trybów. Dane te (pomijając liczbę punktów ELO) są dostępne do wglądu jedynie dla danego użytkownika. Dokładny sposób w jaki ta wyłączność jest implementowana został opisany w dalszej części pracy (patrz sekcja 4.15.1).

Po udanym zalogowaniu użytkownik ma wgląd w swoje statystyki na stronie głównej. Użytkownik może także użyć przycisku odświeżania aby pobrać z serwera najnowsze dane (patrz rysunek Rys. 25).



Rys. 25. Interfejs użytkownika - wgląd w statystyki

4.14 Wgląd w historię rozgrywek użytkownika

Historia rozgrywek użytkownika składa się z rozegranych przez niego rozgrywek w wieloosobowych trybach rozgrywki. Dla każdej z rozgrywek w historii gracz ma wgląd w:

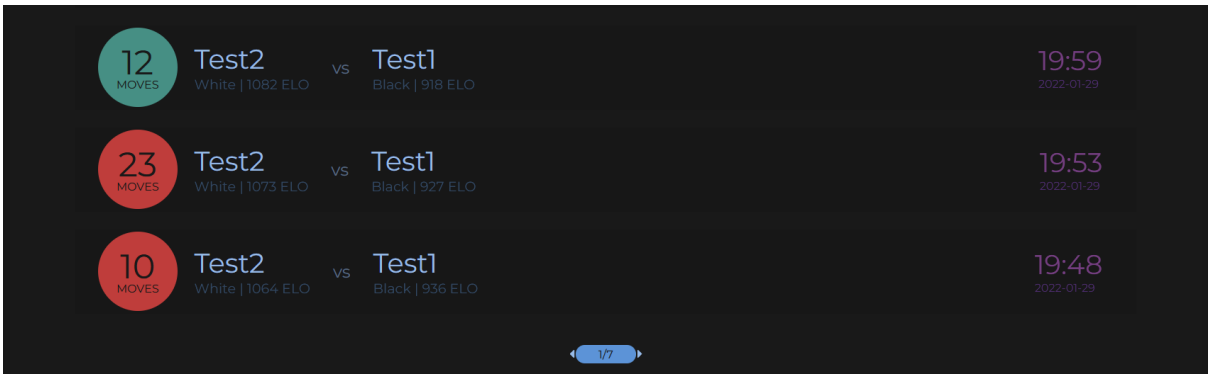
- wynik rozgrywki,
- liczbę wykonanych przez uczestników ruchów,
- liczbę punktów ELO swoją i przeciwnika w momencie rozpoczęcia rozgrywki,
- kolory bierek każdego z graczy,
- datę i godzinę rozpoczęcia rozgrywki.

Dane te są dostępne do wglądu jedynie dla danego użytkownika. Dokładny sposób, w jaki ta wyłączność jest implementowana został opisany w dalszej części pracy (patrz sekcja 4.15.1).

Po udanym zalogowaniu użytkownik ma wgląd w swoją historię rozgrywek w dolnej części strony głównej. W przypadku gdy użytkownik nie rozegrał jeszcze żadnych rozgrywek historia zostaje zastąpiona zaślepką (patrz rysunek Rys. 26).

Rys. 26. Interfejs użytkownika - pusta historia rozgrywek

Ponieważ historia rozgrywek każdego użytkownika może potencjalnie posiadać setki lub tysiące pozycji, aby uniknąć nadmiernego obciążania serwera, rozgrywki w historii dzielone są na strony. Każda strona to 10 rozgrywek, strony numerowane są od 1, gdzie na pierwszej stronie znajduje się nie więcej niż 10 najnowszych rozgrywek danego użytkownika. Użytkownik może przełączać między stronami używając przycisków znajdujących się pod historią rozgrywek (patrz Rys. 27.).



Rys. 27. Interfejs użytkownika - paginacja historii rozgrywek

Sposób w jaki kolejne strony historii są pobierane od serwera został opisany w dalszej części pracy (patrz sekcja 4.15.1).

4.15 Przebieg komunikacji aplikacji klienckiej z serwerem

Komunikacja między klientem a serwerem odbywa się w dwóch trybach, asynchronicznym oraz synchronicznym. Oba z nich zostały szczegółowo opisane w podsekcjach 4.15.1 oraz 4.15.2.

4.15.1 Komunikacja w trybie asynchronicznym

Przy komunikacji w trybie asynchronicznym używane jest REST API implementowane na serwerze Flask. REST API wykorzystywane jest do realizacji serwisu logowania oraz przy pobieraniu większości danych z serwera (historia gier, statystyki gracza itp.).

Odsłonięto następujące węzły końcowe (ang. *endpoint)* podane w tabeli 11. Kolumna “argumenty” odnosi się do argumentów podanych jako parametry URL (np. w www.example.com?userId=”1234” argumentem jest userId). Natomiast wartość kolumny “uwierzytelnianie” informuje czy po otrzymaniu zapytania do danego węzła końcowego serwer będzie wymagał okazania poprawnego tokena sesyjnego w nagłówku ‘Authorize’ zapytania. Dodatkowo, dla każdego zdefiniowanego węzła końcowego, dostępna jest metoda OPTIONS, jej dostępność pozwala realizować tak zwane *preflight requests* związane z *CORS [8].*

Tabela 11. Odsłonięte węzły końcowe

ścieżka adresu URL	argumenty	metody HTTP	argumenty w ciele zapytania	uwierzytelnianie
/register	--	POST, OPTIONS	*username*, *hashed_password*	NIE
/login	--	POST, OPTIONS	*username*, *hashed_password*	NIE
/check2Fa	--	POST, OPTIONS	*username, code*	NIE
/delete	*userId*	DELETE, OPTIONS	--	TAK
/getUserDetails	*userId*	GET, OPTIONS	--	TAK
/update	*userId*	POST, OPTIONS	*hashed_new_password, hashed_current_password, email, is_2_fa_enabled, two_fa_code*	TAK
/resent	*email / username*	GET, OPTIONS	--	NIE
/confirm	*token*	GET, OPTIONS	--	NIE
/reset	--	POST, OPTIONS	*token, hashed_password*	NIE
/forgotPassword	*email*	GET, OPTIONS	--	NIE
/logout	*userId*	GET, OPTIONS	--	TAK
/refresh_session	--	GET, OPTIONS	--	NIE
/is_in_game	*userId*	GET, OPTIONS	--	TAK
/get_game_info	*gameId*	GET, OPTIONS	--	NIE
/get_player_stats	*userId*	GET, OPTIONS	--	TAK
/match_history	*userId*	GET, OPTIONS	--	TAK
/get_available_game_modes	--	GET, OPTIONS	--	NIE

Poniżej przedstawiono wszystkie możliwe odpowiedzi, jakie klient może otrzymać od zdefiniowanych węzłów końcowych (patrz Tabela 12.). Zmienne w ciele odpowiedzi oznaczono kursywą.

Tabela 12. Możliwe odpowiedzi na zapytania do węzłów końcowych

Ścieżka adresu URL	Kod	Ciało odpowiedzi	Kiedy występuje
/register	200	{"registration": 'successful'}	Po poprawnej rejestracji.
/login	200	{ *user_id*, *session_token*}	Po przesłaniu poprawnych danych logowania.
/check2Fa	200	{“result”: true}	Po podaniu poprawnego kodu OTP
/delete	200	{“response”: “OK”}	Przy poprawnym usunięciu konta użytkownika
/getUserDetails	200	{“response” : user_data}	Przy poprawnym pobraniu danych użytkownika
/update	200	{“response”: “OK”}	Przy poprawnej edycji danych użytkownika
/resent	200	{“response”: “OK”}	Po poprawnym wysłaniu wiadomości pozwalającej na aktywację konta użytkownika
/reset	200	{“response”: “OK”}	Po poprawnym ustawieniu nowego hasła
/forgotPassword	200	{“response”: “OK”}	Po poprawnym wysłaniu wiadomości pozwalającej na ustawienie nowego hasła
/logout	200	{"logout": 'successful'}	Po poprawnym wylogowaniu.
/refresh_ session	200	{*sessionToken*}	Po podaniu poprawnego tokena *refresh*.
/is_in_game	200	{"inGame": False}	Gracz o podanym Id nie jest w grze.
/is_in_game	200	{"inGame": True, *gameId, gameMode, playingAs*}	Gracz o podanym Id jest w grze.
/get_game_ info	*200*	{*gameMode* ,*currentTurn*, *FEN*, "whitePlayer": {*w_username*, *w_ELO*,},	Podano *gameId* istniejącej gry.

		"blackPlayer":{ *b_username*, *b_ELO*}, *blackTime*, *whiteTime}*	
/get_player_stats	200	{*elo,deviation,gamesPlayed ,gamesWon, gamesLost, draws}*	Przy poprawnym tokenie sesyjnym w nagłówku ‘Authorize’.
/match_ history	200	‘Match’:{*matchResult,p1User name, , p1PlayedAs, p1ELO, p2Username, p2PlayedAs, p2ELO, nOfMoves, dateTime},* *‘Match’: {...}, ...*	Przy poprawnym tokenie sesyjnym w nagłówku ‘Authorize’.
/get_available_ga me_modes	200	[{*gameModeId,gameModeNa me,gameModeDesc,gameMod eTime,gameModeStartingFEN ,gameModeMultiplayer}* , *{...}]*	Zawsze.
/reset	400	{"error": "Token expired", "token": token}	Przy użyciu wygasłego tokenu
/reset	400	{"error": "Token invalid", "token": token}	Przy użyciu niepoprawnego tokenu
/delete	401	{"error": "Authorization failed."}	Przy błędnym tokenie sesyjnym w nagłówku ‘Authorize’.
/getUserDetails	401	{"error": "Authorization failed."}	Przy błędnym tokenie sesyjnym w nagłówku ‘Authorize’.
/update	401	{"error": "Authorization failed."}	Przy błędnym tokenie sesyjnym w nagłówku ‘Authorize’.
/logout	401	{"error": "Authorization failed."}	Przy błędnym tokenie sesyjnym w nagłówku ‘Authorize’.
/refresh_ session	401	{"error": "Wrong refresh token."}	Przy błędnym tokenie *refresh* przesłanym w ciasteczku.
/is_in_game	401	{"error": "Authorization failed."}	Przy błędnym tokenie sesyjnym w nagłówku ‘Authorize’.
/get_player_stats	401	{"error": "Authorization failed."}	Przy błędnym tokenie sesyjnym w nagłówku ‘Authorize’.
/match_ history	401	{"error": "Authorization failed."}	Przy błędnym tokenie sesyjnym w nagłówku ‘Authorize’.
/login	403	{"error": "Username doesn't exist"}	Po przesłaniu nazwy użytkownika, która nie istnieje w bazie danych.

/login	403	{"error": "Incorrect password"}	Po przesłaniu nieprawidłowego hasła dla istniejącej nazwy użytkownika.
/check2Fa	403	{“result”: false}	Po przesłaniu błędnego hasła jednorazowego.
/register	403	{"error": "Username already taken"}	Po próbie rejestracji zajętej nazwy użytkownika.
/register	403	{"error": "Email already taken”}	Po próbie rejestracji zajętego adresu email.
/delete	403	{"response": "Incorrect password"}	Podanie błędnego hasła przy próbie usunięcia konta użytkownika.
/delete	403	{"response": "Incorrect 2FA code"}	Podanie błędnego kodu jednorazowego przy próbie usunięcia konta użytkownika.
/register	403	{"error": "Username already taken"}	Po próbie zmiany nazwy użytkownika na zajętą.
/register	403	{"error": "Email already taken”}	Po próbie zmiany adresu e-mail na zajęty.
/login, /check2Fa, /register, /delete, /getUserDetails, /update, /resent, /reset, /forgotPassword,/ match_history, /get_ELO_change _in_last_game, /player_stats, /match_history,	503	{"error": "Can't fetch from db"}	Przy błędzie bazy danych.

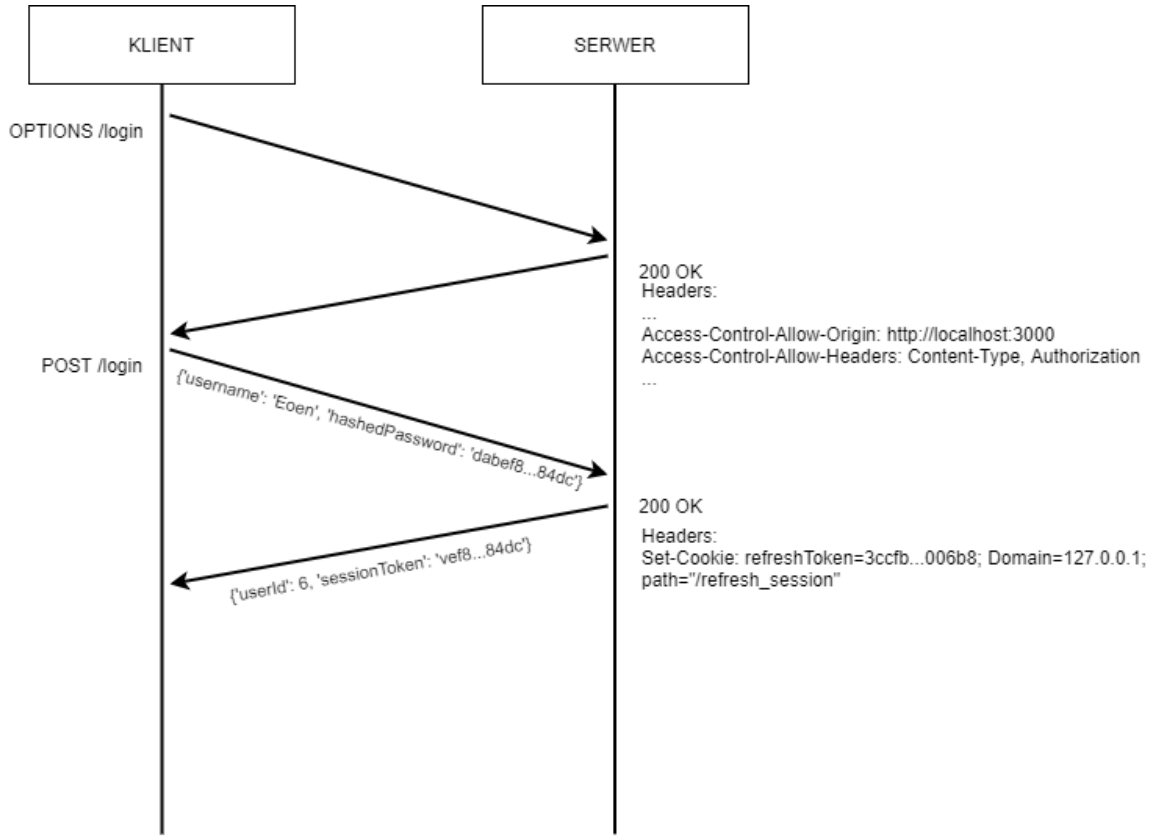
Poniżej opisano zastosowanie każdego z węzłów końcowych.

- /register - wykorzystywany przy rejestracji, użytkownik przesyła wybraną nazwę oraz hasło w formie skrótu w ciele zapytania.
- /login - wykorzystywany przy logowaniu, użytkownik przesyła swoją nazwę oraz skrót hasła w ciele zapytania. W odpowiedzi na poprawne zalogowanie, użytkownik otrzymuje *userId* oraz token sesyjny. Odpowiedź zawiera także nagłówek

‘Set-Cookie’ ustawiający po stronie klienta ciasteczko z tokenem odświeżania (patrz sekcja 4.11).

- /check2Fa - służy do sprawdzenia kodu dwuetapowej weryfikacji podczas logowania się na konto.
- /delete - wykorzystywany do usunięcia konta użytkownika z serwisu.
- /getUserDetails - pobieranie danych użytkownika
- /update - pozwala na edycję danych użytkownika m.in. hasła, adresu e-mail oraz weryfikacji dwuetapowej.
- /resent - ponowne wysłanie wiadomości zawierającej link do aktywacji konta o podanym adresie e-mail.
- /reset - pozwalający na zmianę zapomnianego hasła do konta.
- /forgotPassword - umożliwiający wysłanie wiadomości e-mail na podany adres zawierającą link do zresetowania aktualnego hasła.
- /logout - służy do wylogowania użytkownika po stronie serwera. Po otrzymaniu tego zapytania serwer autoryzuje użytkownika (aby niemożliwe było wylogowanie innego użytkownika) a następnie usuwa jego wpis ze słownika *sessions* (patrz sekcja 4.1.1) w ten sposób unieważniając jego tokeny. Odpowiedź zawiera także nagłówek ‘Set-Cookie’ ustawiający po stronie klienta ciasteczko z pustym tokenem odświeżania.
- /refresh_session - wykorzystywany do otrzymania nowego tokena sesyjnego przez użytkownika (np. przy odświeżeniu karty). Wymaga przedstawienia poprawnego tokena odświeżania przesyłanego w ciasteczku.
- /is_in_game - wykorzystywany przez gracza przy próbie ponownego dołączenia do gry (np. po zamknięciu karty lub wylogowaniu). W odpowiedzi gracz otrzymuje swój stan gry oraz podstawowe informacje o niej takie jak identyfikator gry czy kolor jego bierek.
- /get_game_info - wykorzystywany przez gracza do otrzymania szczegółowych informacji o grze o danym identyfikatorze. Nie wymaga autoryzacji tokenem sesyjnym w nagłówku ‘Authorize’, ponieważ w przyszłości będzie wykorzystywany przy implementacji oglądania gier innych graczy.
- /get_player_stats - wykorzystywany przez gracza do otrzymania statystyk jego gier.
- /match_history - wykorzystywany do otrzymania historii gier danego gracza.
- /get_available_game_modes - wykorzystywany przez aplikację kliencką do otrzymania listy dostępnych trybów gry.

Na rysunku Rys. 28 pokazano przykładowy przebieg procesu autoryzacji.



Rys. 28. Przykład komunikacji podczas logowania użytkownika

Gracz wysyła zapytanie OPTIONS do węzła końcowego ‘/login’, na które, ponieważ pochodzi ono z odpowiedniej domeny, otrzymuje odpowiedź z nagłówkami CORS [8]. Następnie wysyła zapytanie POST do tego samego węzła końcowego, w ciele zapytania umieszczając nazwę użytkownika i skrót hasła. Jeżeli w bazie danych istnieje użytkownik o podanej nazwie i haśle serwer odsyła do użytkownika komunikat z kodem 200 OK oraz nagłówkiem ‘Set-Cookie’. Nagłówek ten zawieraja token odświeżania, ciąg znaków, który wysłany do węzła końcowego ‘/refresh_session’ pozwala na wygenerowanie nowego tokena sesyjnego. Odebranie tego pakietu spowoduje ustawienie ciasteczka po stronie klienta.

W ciele odpowiedzi serwera przekazywany jest także token sesyjny dla klienta oraz jego identyfikator pobrany z bazy danych. Token sesyjny będzie wysyłany w nagłówku ‘Authorization’ każdego następnego zapytania, wysłanego przez klienta i sprawdzany w celu autoryzacji. Identyfikator klienta natomiast, pozwoli mu zidentyfikować się na serwerze, w celu uzyskania odpowiednich zasobów (np. jego historii gier).

4.15.2 Komunikacja w trybie synchronicznym

Komunikacja między klientem a serwerem odbywa się w trybie synchronicznym (tj. w czasie rzeczywistym). Taka komunikacja jest przydatna do sprawnej realizacji doboru

przeciwnika oraz gry szachowej. Do jej implementacji wykorzystano technologię *websocket*, zapewniającą pełno dupleksową komunikację przy użyciu jednego połączenia TCP.

Z tą technologią związany jest pewien narzut, który nie występowałby przy użyciu "czystego" gniazda TCP. Przy rozpoczęciu sesji komunikacyjnej jest to jednorazowe przesłanie około ~700 dodatkowych bajtów (patrz załącznik 1 w sekcji 8). Po ustanowieniu połączenia narzut z każdą wysłaną wiadomością jest niewielki, ponieważ ich treści są krótsze niż 160 bajtów. Wiadomości od serwera do klienta mają długość dwóch bajtów. Wiadomości od klienta do serwera mają długość sześciu bajtów [3]. Dodatkowym powodem skorzystania z technologii *websocket* jest łatwość użycia oraz implementacji w przeglądarkach internetowych.

Poniżej przedstawiono przykładowy komunikat websocket podsłuchany w programie wireshark.

```
["update_timers",{"whiteTime":598.9999995208345,"blackTime":600}]
```

Każdy komunikat ma formę ["typ",{dane komunikatu}]. Dane komunikatów składają się z par "klucz":"wartość" oddzielonych od siebie przecinkami. Pozwala to na łatwe odczytywanie ich zawartości przy użyciu bibliotek do obsługi formatu *JSON*. Poniżej wymieniono wszystkie zdefiniowane typy komunikatów podzielone ze względu na zastosowanie (patrz Tabela 13, Tabela 14 oraz Tabela 15).

Tabela 13. Komunikaty websocket związane z łączeniem i autoryzacją

Typ	Argumenty	Opis
connect	--	Komunikat wysyłany po podłączeniu się do serwera, inicjalizuje sesje komunikacyjną, ma puste pole danych. Klient jest identyfikowany przez identyfikator gniazda *(*ang. *socket identificator*, w skrócie *sid)* przypisany na stałe do danego *websocket'*u.
connected	--	Komunikat wysyłany do klienta przez serwer po poprawnym połączeniu, ma puste pole danych.
authorize	*playerId*, *sessionToken*	Komunikat wysyłany przez klienta po zalogowaniu w celu autoryzacji *websocket'u* o danym identyfikatorze gniazda. Przekazany token sesyjny *(ang. sessionToken*) jest porównywany z tokenem w tablicy aktualnych sesji graczy na serwerze *sessions* (patrz sekcja 4.1.1). Jeżeli zgadza się *sid websocket*'u wpisywany on jest do słownika *authorized_sockets* (patrz sekcja 4.1.1) z kluczem równym identyfikatorowi gracza (ang. *playerId)*.

authorized	--	Komunikat odsyłany w odpowiedzi na poprawną autoryzację, ma puste pole danych.
unauthorized	--	Komunikat odsyłany w odpowiedzi na niepowodzenie w autoryzacji, próbę użycia nieautoryzowanego gniazda lub wykonania nieautoryzowanego działania (np. wysłania ruchu w nie swojej grze) ma puste pole danych.

Próba nawiązania połączenia *websocket'u* rozpoczyna się po zalogowaniu użytkownika od wysłania komunikatu *connect*. Po nieudanej próbie połączenia, czyli takiej, na jaką w odpowiedzi nie otrzymano komunikatu *connected*, klient próbuje połączyć się z serwerem ponownie, najpierw po sekundzie, a następnie, w interwałach czasu rosnących dwukrotnie z każdą próbą.

Po udanym połączeniu następuje próba autoryzacji przez wysłanie pakietu *authorize* z tokenem sesyjnym użytkownika, nieudana otrzymuje w odpowiedzi *unauthorized*, natomiast udana *authorized*. Nieudana próba autoryzacji skutkuje próbą uzyskania nowego tokena sesji, jeżeli to także nie wyjdzie, użytkownik jest wylogowywany.

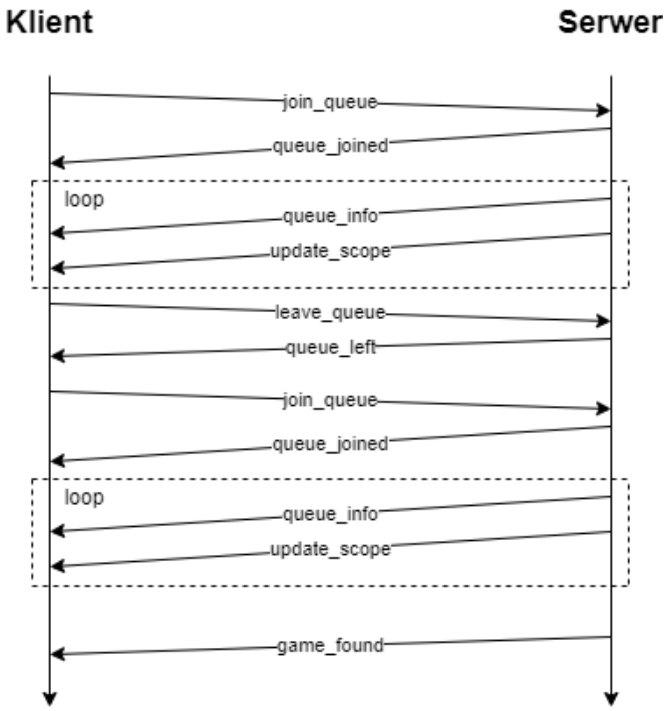
Po odebraniu każdego komunikatu poza c*onnect* i *authorize.* serwer sprawdza czy użytkownik, od którego go otrzymał, jest zautoryzowany (wpisany do słownika *authorized_sockets*).

Tabela 14. Komunikaty websocket związane z doborem przeciwnika

Typ	Argumenty	Opis
join_queue	*playerId*	Komunikat wysyłany przez klienta o danym *playerId* do serwera w celu dołączenia do kolejki oczekiwania na grę.
queue_joined	--	Komunikat wysyłany przez serwer do klienta po poprawnym dołączeniu do kolejki (dodaniu do tablicy *queue*).
leave_queue	*playerId*	Komunikat wysyłany przez klienta o danym *playerId* do serwera w celu opuszczenia kolejki oczekiwania na grę.
queue_left	--	Komunikat wysyłany przez serwer do klienta po poprawnym opuszczeniu kolejki (usunięciu z tablicy *queue*).
already_in_queue	--	Komunikat wysyłany przez serwer do klienta w odpowiedzi na *join_queue,* jeżeli jest on już w kolejce oczekiwania.

queue_info	*playersInQueue*	Komunikat wysyłany przez serwer do wszystkich klientów oczekujących w kolejce po dołączeniu nowego klienta do kolejki lub opuszczeniu jej. Zawiera informacje o ilości klientów aktualnie oczekujących w kolejce.
update_scope	*scope*	Komunikat wysyłany przez serwer do klienta po rozszerzeniu jego zakresu poszukiwań przeciwników.
game_found	*gameId,playingAs*	Komunikat wysyłany przez serwer do graczy których dobrano do gry. Zawiera identyfikator gry oraz kolor bierek wylosowany dla danego gracza.

Na rysunku Rys. 29 przedstawiono przykładowy diagram czasowy komunikacji z serwerem gracza chcącego dobrać przeciwnika.



Rys. 29. Diagram czasowy doboru przeciwnika

Tabela 15. Komunikaty websocket związane z przebiegiem gry

Typ	Argumenty	Opis
make_move	*playerId,gameId, move*	Komunikat wysyłany przez gracza do serwera po wykonaniu ruchu w aplikacji klienckiej. Zawiera identyfikator gracza wykonującego ruch, identyfikator gry, w

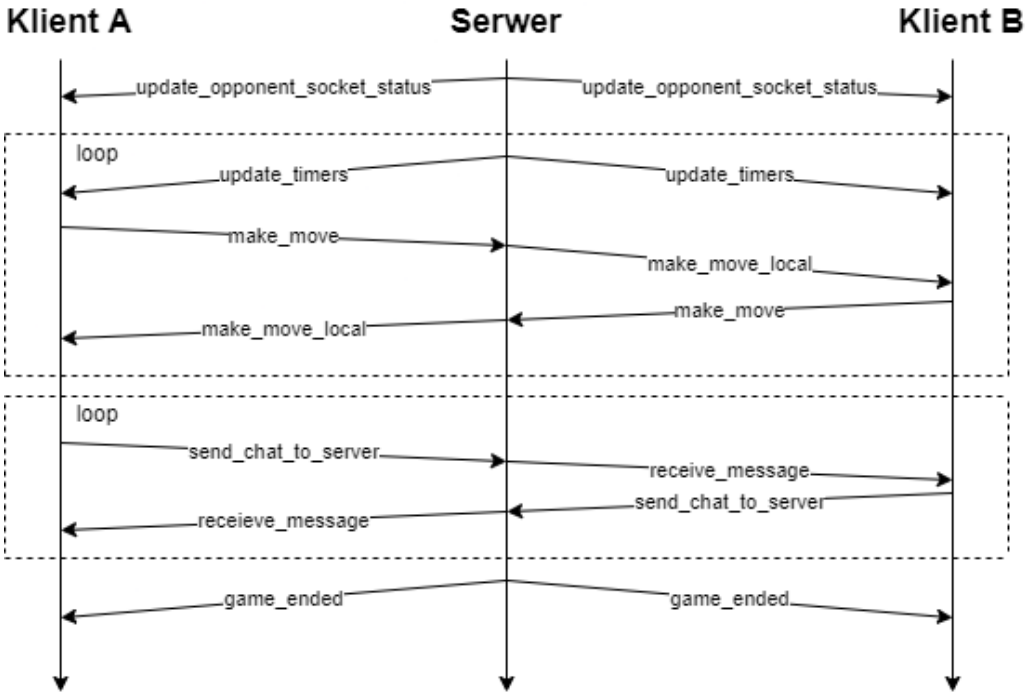
		której ruch ma być wykonany oraz sam ruch[5].
illegal_move	*move*	Komunikat wysyłany przez serwer do gracza po otrzymaniu komunikatu *make_move* zawierającego niedozwolony ruch szachowy, ruch w grze, w której nie jest graczem lub ruch w nie swojej turze. Otrzymanie tego komunikatu po stronie klienta skutkuje cofnięciem ruchu podanego w ciele odpowiedzi.
make_move_local	*move*	Komunikat wysyłany przez serwer do gracza po otrzymaniu komunikatu *make_move* (zawierającego dozwolony ruch) od przeciwnika. Otrzymanie tego komunikatu po stronie klienta skutkuje wykonaniem go na planszy.
place_defender_piece	*playerId,gameId, FEN*	Komunikat wysyłany przez gracza do serwera w celu ustawienia nowej figury w fazie pierwszej rozgrywki w trybie defender. Jako argument przekazywany jest zaktualizowany FEN z żądanym nowym ustawieniem.
place_defender_piece_local	*FEN, white_points,black_points*	Komunikat wysyłany przez serwer do graczy po otrzymaniu przez niego komunikatu *place_defender_piece*.
propose_draw	*playerId,gameId*	Komunikat wysyłany przez gracza do serwera w celu zaproponowania drugiemu remisu w rozgrywce.
draw_proposed		Komunikat wysyłany przez serwer do gracza, któremu zaproponowano remis.
draw_response	*accepted*	Komunikat wysyłany przez gracza, któremu zaproponowano remis do serwera. A następnie kopiowany przez serwer do gracza, który zaproponował remis. Jako *accepted* argument gracz podaje *true*

[5] W postaci struktury opisanej w sekcji 4.1.2 Struktury danych po stronie klienta.

		(zgadza się na remis) lub *false* (odrzuca remis).
update_timers	*whiteTime, blackTime*	Komunikat wysyłany przez serwer do graczy po upływie pełnej sekundy czasu któregokolwiek z nich. Otrzymanie tego komunikatu po stronie klienta skutkuje aktualizacją zegarów szachowych graczy.
send_chat_to_server	*playerId. gameId, playerName, text,*	Komunikat wysyłany przez gracza do serwera w celu wysłaniu wiadomości czatu do przeciwnika.
receive_message	*text, playerName*	Komunikat wysyłany przez serwer do przeciwnika gracza, od którego serwer otrzymał komunikat *send_chat_to_server.* Otrzymanie tego komunikatu po stronie klienta skutkuje dodaniem wiadomości z pola *text* do czatu.
update_opponents_socket_status	*status*	Komunikat wysyłany przez serwer do przeciwnika gracza, którego stan połączenia z serwerem się zmienił (np. wskutek rozłączenia, lub ponownego dołączenia). Otrzymanie tego komunikatu po stronie klienta skutkuje zmianą stanu widżetu połączenia przeciwnika na ten z pola *status (*możliwe stany to ‘disconnected’ oraz ‘connected’).
surrender	*playerId. gameId*	Komunikat wysyłany przez gracza do serwera w celu poddania gry. Zawiera identyfikator gracza, który chce się poddać oraz identyfikator poddawanej gry.

game_ended	*result*	Komunikat wysyłany przez serwer do graczy po wykryciu zakończenia gry (np. wskutek mata czy upływu czasu jednego z graczy). Zawiera pole result określające wynik dla każdego z graczy (możliwe wartości *result* to ‘win’, ‘lose’ lub ‘draw’). Otrzymanie tego komunikatu po stronie klienta skutkuje wyświetleniem ekranu zakończenia gry (patrz rys. 30.).

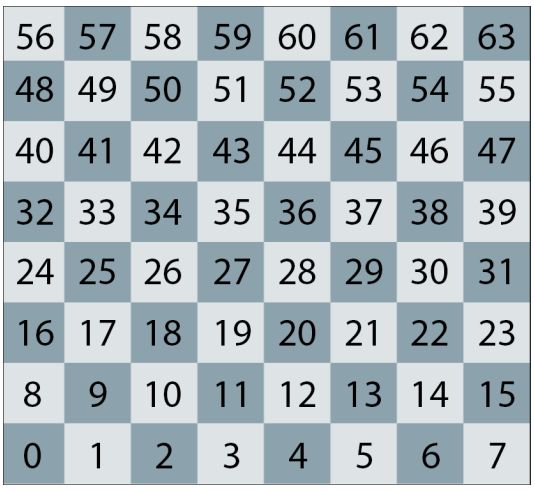
Na rysunku Rys. 30 przedstawiono przykładowy diagram czasowy komunikacji z serwerem graczy w trakcie gry.



Rys 30. Diagram czasu przebiegu gry

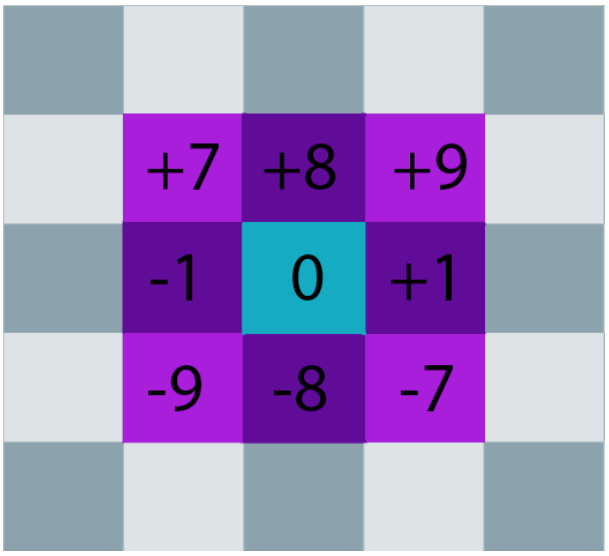
4.16 Generowanie możliwych do wykonania ruchów po stronie klienta

Aplikacja kliencka generuje wszystkie możliwe do wykonania ruchy, które potem są wykorzystywane w ramach podpowiedzi dla użytkownika. Pola na szachownicy są oznaczone indeksami od 0 do 63 patrząc od strony gracza grającego bierkami białymi (patrz rysunek Rys. 31).



Rys. 31. Numerycznie opisane pola na szachownicy

Na tak zaindeksowanej planszy, ruch niezbędny do przemieszczenia bierki między danym polem startowym, a końcowym, możemy oznaczyć jako różnicę indeksów pola końcowego i startowego (dalej *offset)*. Na rysunku 32 przedstawiono *offset* sąsiednich pól dla bierki znajdującej się na polu oznaczonym ‘0’.



Rys. 32. Kierunki przedstawione za pomocą *offset’u*

W ten sposób wszystkie możliwe kierunki poruszania bierki można zapisać za pomocą *offsetu*. Aby poruszyć się od danego pola w prawo lub w lewo, należy odpowiednio odjąć lub dodać jeden. Aby poruszać się w górę lub dół, należy dodać lub odjąć osiem. Aby poruszać się po skosach w lewo, należy dodać siedem lub odjąć dziewięć. Natomiast aby poruszać się po skosach w prawo, należy dodać dziewięć lub odjąć siedem. Wyjątkiem są pola, dla których dodanie *offset’u* wiązałoby się z przekroczeniem krawędzi planszy (na przykład pola po skrajnie lewej stronie planszy, nie mają dalszych pól po lewej).

Do ułatwienia dalszej implementacji kierunki przedstawione za pomocą *offset*’u, zostały przechowane w tablicy w następującej kolejności: 8, -8, -1, 1, 7, -7, 9, -9. Kolejność ta jest ważna przy późniejszej generacji ruchów.

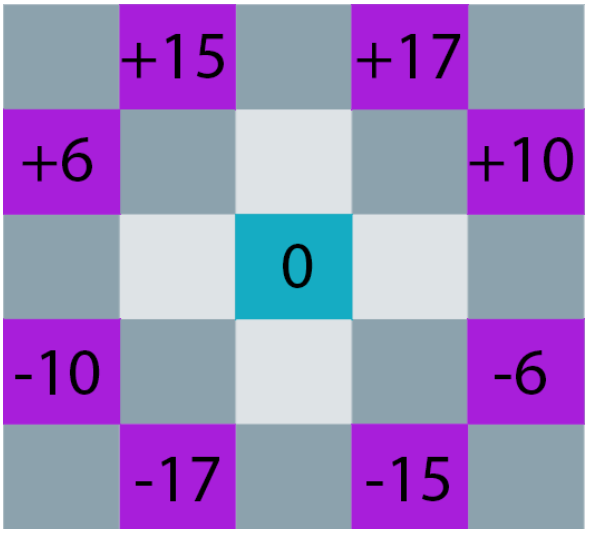
4.16.1 Generowanie ruchów królowej, wieży oraz gońca

Aby wygenerować ruchy królowej należy najpierw znaleźć jej indeks na planszy. Następnie dla każdego z kierunków dodawać do początkowego indeksu królowej wartość kierunku, aż do napotkania końca planszy lub innej bierki. Jeżeli napotkana bierka, jest bierką przeciwnika, dodaje się dodatkowo ruch pozwalający na jej zbicie.

Generowanie ruchów dla wież oraz gońców wygląda w taki sam sposób z tym, że kierunki, w które mogą poruszać się wieże, to tylko pierwsze cztery z tablicy kierunków, natomiast dla gońców ostatnie cztery.

4.16.2 Generowanie ruchów skoczka

Aby wygenerować ruchy skoczka, zostały przygotowane specjalnie kierunki (patrz rysunek Rys. 33). W tym przypadku jeżeli skoczek stoi na polu oznaczonym ‘0’ a na docelowym polu nie znajduje się żadna bierka, lub znajduje się bierka przeciwnika, należy odpowiednio dodać ruch umożliwiający przejście na dane pole lub zbicie bierki.



Rys. 33. Pola na które może poruszać się skoczek przedstawione za pomocą offset’u

4.16.3 Generowanie ruchów piona

Aby wygenerować podstawowe ruchy piona, należy sprawdzać trzy pola. Jeżeli pole przed pionem jest puste, dodawany jest ruch pionem do przodu o jedno pole. Jeżeli na polach przed pionem po skosie, znajduje się bierka przeciwnika, należy dodać ruch umożliwiający jej bicie.

Piony, które jeszcze nie wykonały żadnego ruchu, mają również możliwość ruchu o dwa pola do przodu, wtedy w analogiczny sposób jest sprawdzane pole, znajdujące się dwa pola nad pionem. Jeżeli to pole jest puste, ruch zostaje dodany do listy możliwych ruchów.

Jeżeli ruch piona będzie kończył się na krawędzi planszy po stronie przeciwnika, po wykonaniu ruchu pion jest promowany do królowej (bierka piona zastąpiona jest bierką królowej tego samego koloru).

Wyjątkowym ruchem jest bicie w przelocie. Ten ruch dodajemy tylko i wyłącznie wtedy, gdy pion wykonał ruch o dwa pola do przodu, a pole, które minął było atakowane

przez piona przeciwnika. W takiej sytuacji pionowi atakującemu minięte pole, należy dodać ruch umożliwiający zbicie piona poruszonego o dwa pola. W tej wyjątkowej sytuacji, gdy jest możliwe bicie w przelocie do zapisu FEN dodaje się pole, na którym wylądowałby bijący pion po wykonaniu tego ruchu. Jest to potrzebne, ponieważ w momencie wznawiania partii, po samym ułożeniu pionów, nie jesteśmy wstanie powiedzieć, czy bicie w przelocie powinno być możliwe.

4.16.4 Generowanie ruchów króla

Król może poruszać się w każdym kierunku, o ile pole nie jest atakowane przez żadną z przeciwnych bierek ani na danym polu nie znajduje się inna bierka. Jeżeli jednak na polu znajduje się bierka przeciwnika, dodajemy również ruch umożliwiający jej zbicie, o ile wykonanie takiego ruchu nie pozostawi króla w pozycji atakowanej.

Dodatkowo jeżeli między królem a wieżą tego samego koloru, nie znajdują się żadne bierki oraz ani król ani wieża nie został jeszcze poruszone, należy dodać ruch umożliwiający zrobienie roszady. Możliwość roszady jest zapisana w zapisie FEN, ponieważ po samym ułożeniu króla oraz wież, nie jesteśmy wstanie powiedzieć, czy istnieje możliwość roszady.

4.16.5 Wyjątki od wyżej wymienionych zasad

W momencie, gdy król jest szachowany, w liście wygenerowanych ruchów, zostają zachowane tylko ruchy, które pozwalają na przerwanie szachu. Na przykład może to być zasłonięcie się inną bierką, zbicie bierki atakującej, czy odsunięcie się królem.

Dodatkowo dla każdego wygenerowanego ruchu, należy sprawdzać czy po jego wykonaniu, król nie będzie znajdował się w szachu. Dlatego też, przy generacji ruchów, symulowany jest jeden ruch do przodu, aby sprawdzić czy taka sytuacja nie zaistnieje.

4.16.6 Pierwsza faza rozgrywki trybu *ChessDefender*

W pierwszej fazie rozgrywki w trybie *ChessDefender* gracze mogą stawiać bierki na trzech pierwszych rzędach planszy patrząc z perspektywy danego gracza, o ile posiadają wystarczającą liczbę punktów *defender* i na docelowym polu nie jest już położona inna bierka. Jedynym wyjątkiem jest stawianie króla, który dodatkowo nie może zostać wystawiony w szachu.

4.17 Szachownica - graficzny interfejs użytkownika

Graficzny interfejs szachownicy został w pełni stworzony przy pomocy biblioteki p5.js. (patrz sekcja 3.2).

Sama szachownica generowana jest przy pomocy wbudowanych w bibliotekę funkcji. Jest ona stworzona z 64 obiektów *square*, które posiadają dynamicznie przypisywaną pozycję oraz wielkość, w zależności od wielkości okna przeglądarki. Następnie, obiekty te kolorowane są na przemian, jednym z dwóch kolorów. Ostatecznie szachownica jest renderowana.

Następnie pobierany z serwera jest FEN, na podstawie którego są tworzone obiekty “Piece” (patrz sekcja 4.1.2), czyli bierki. Na podstawie pola *type* każdego z obiektów, jest renderowana bierka, z wcześniej załadowanych tekstur.

Jeżeli pozycja kursora będzie znajdowała się na bierce i zostanie wykryte wciśnięcie przycisku myszy, to do momentu jego puszczenia, będzie można ją przesuwać za pomocą myszy. W takim wypadku bierka jest ponownie renderowana na pozycji kursora. Jeżeli bierkę przesuwa gracz, którego jest aktualnie tura, na szachownicy zostaną podświetlone wszystkie dozwolone ruchy, które można wykonać przenoszoną bierką. Po puszczeniu przycisku myszy, pobrana zostaje aktualna pozycja kursora i jeżeli znajduje się ona na polu, na który można wykonać ruch, wysyłana jest informacja do serwera, o wykonanym ruchu. Jeżeli serwer potwierdzi to, że wykonywany ruch jest możliwy bierka zostanie przeniesiona na to pole. Jeżeli jednak kursor nie znajduje się na polu, na który można wykonać ruch lub poprawność ruchu nie zostanie potwierdzona przez serwer, bierka zostanie przywrócona do pozycji sprzed podniesienia.

W trybie *ChessDefender* dodatkowo, po prawej stronie planszy są renderowane obiekty “Piece”, po jednym dla każdej z bierek. Wykorzystywane są one w fazie inicjalizacji pozycji. W razie niewystarczającej liczby punktów do wystawienia bierki, obiekt jest usuwany, a pod nim renderowana jest częściowo przezroczysta tekstura tej bierki, która ma na celu sygnalizowanie braku możliwości wystawienia tej bierki. Nad bierkami możliwymi do wystawienia, wyświetlany jest też stan aktualnie posiadanych punktów.

4.18 Wykonywanie ruchów przez sztuczną inteligencję

W trybie *ChessDefender* dla jednego gracza, jeżeli aktualnie jest tura gracza komputerowego, aplikacja kliencka oczekuje, aż serwer prześle ruch wykonany przez silnik *StockFish.*

Po stronie serwera, ruch jest generowany za pomocą silnika szachowego *StockFish*, z ograniczeniem głębokości, przy pomocy biblioteki python-chess. Taki ruch jest przesyłany w szachowej notacji algebraicznej do aplikacji klienckiej, za pomocą komunikatu *websocket - make_move_local* (patrz sekcja 4.15.2).

4.19 Wdrożenie aplikacji do środowiska produkcyjnego

Ostatnim krokiem w implementacji aplikacji jest wdrożenie jej do środowiska produkcyjnego, które umożliwi dostęp do aplikacji z Internetu. Jeżeli wyniki wykonanych testów są zadowalające, aplikacja jest wdrażana na serwer zewnętrzny *VPS*. Pierwszym krokiem jest odpowiednie ustawienie wymaganych adresów *IP*, tj. po stronie aplikacji klienckiej, adresu *API* oraz po stronie *API*, adresu *IP*, na którym serwer ma nasłuchiwać żądania od klientów. Baza danych nie wymaga dodatkowej konfiguracji, ponieważ jest przechowywana w postaci skonteneryzowanej. Następnym krokiem jest odpowiednie skonfigurowanie serwera *HTTP*, aby poprawnie interpretował wysyłane żądania. Ostatnim krokiem jest instalacja certyfikatu *SSL* pozwalającego na zapewnienie poufności danych transmitowanych przez internet. Aplikacja dostępna jest pod adresem https://neochess.pl.

5. TESTY

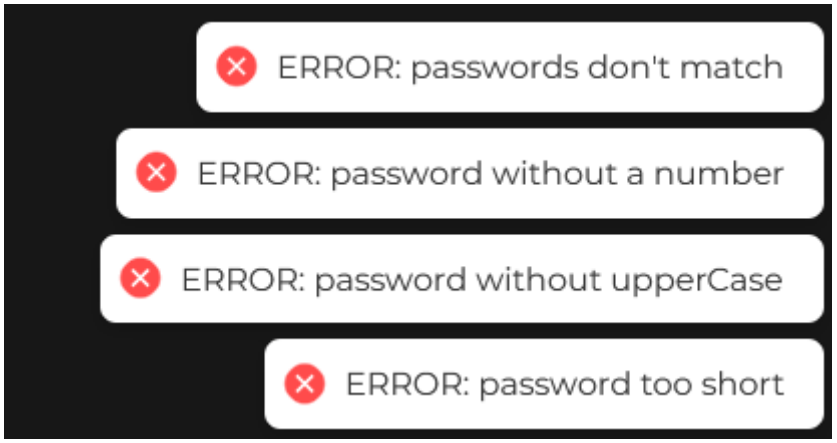
Wykonane testy aplikacji klienckiej zostały przeprowadzone na przeglądarkach Firefox oraz Google Chrome.

5.1 Testy serwisu logowania i rejestracji

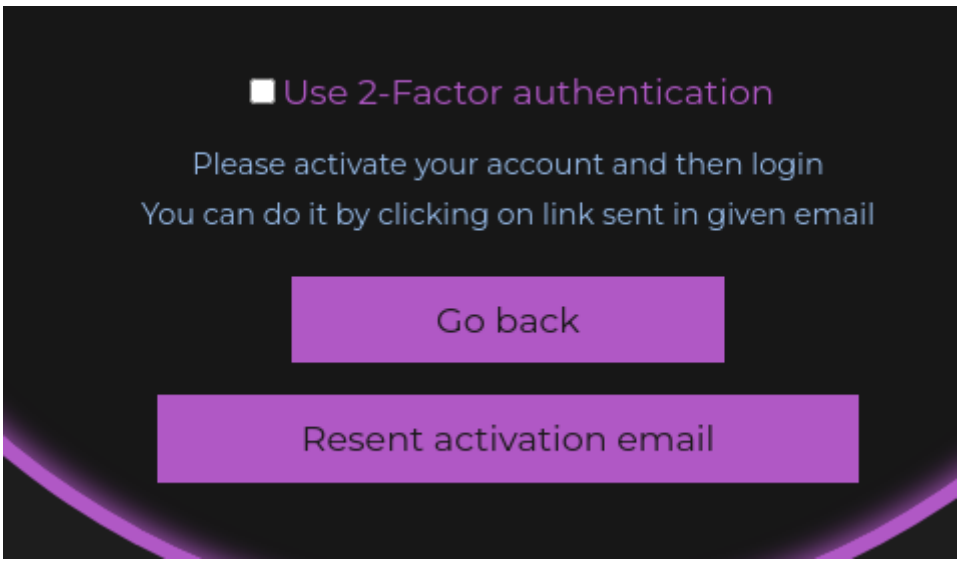
Testy sprawdzające poprawność działania mechanizmów logowania i rejestracji przedstawionych poniżej.

- Rejestracja przy użyciu błędnych danych.
- Rejestracja przy użyciu poprawnych danych.
- Logowanie przy użyciu błędnych danych.
- Logowanie przy użyciu poprawnych danych.
- Logowanie na nieaktywne konto.
- Aktywowanie konta wygasłym / nieprawidłowym tokenem
- Weryfikacja działania mechanizmu *CAPTCHA*

W celu przetestowania mechanizmu rejestracji, wykonana została próba rejestracji przy użyciu hasła nie spełniającego polityki haseł aplikacji, zakończona wyświetleniem odpowiednich błędów (patrz rysunek Rys. 34.). Została przeprowadzona próba rejestracji przy użyciu poprawnych danych (patrz rysunek Rys. 35.), zakończona sukcesem, czyli utworzeniem konta oraz wyświetlaniem wiadomości o potrzebie jego aktywacji.

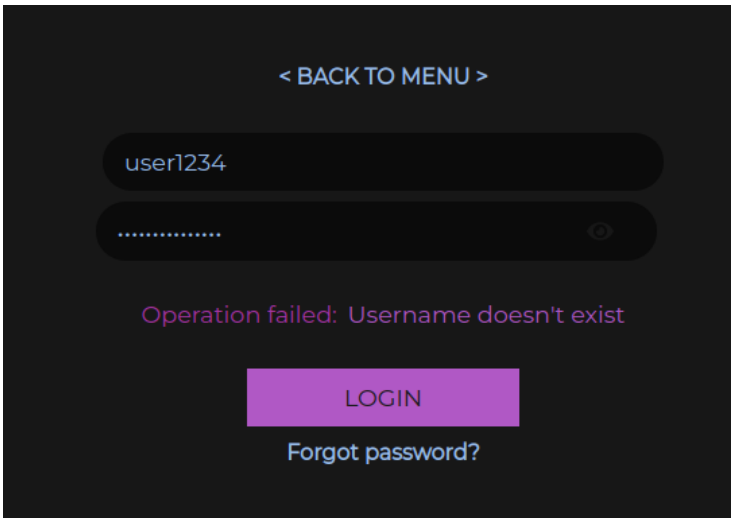


Rys. 34. Nieudana próba rejestracji danymi niespełniającymi zasad

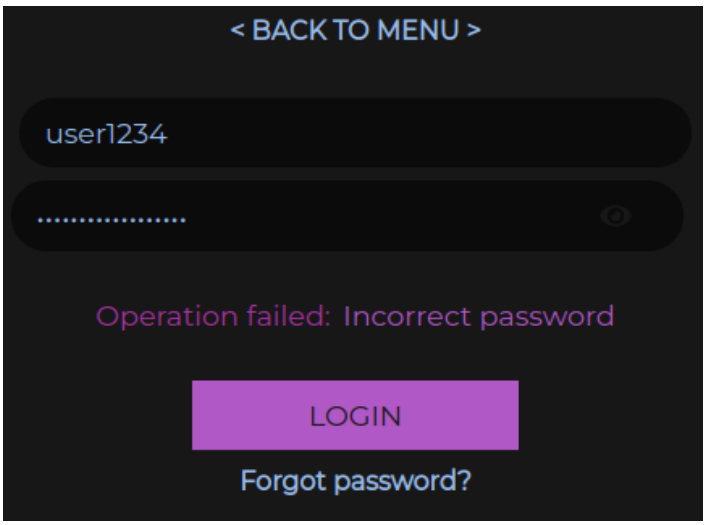


Rys. 35. Udana próba rejestracji przy użyciu poprawnych danych

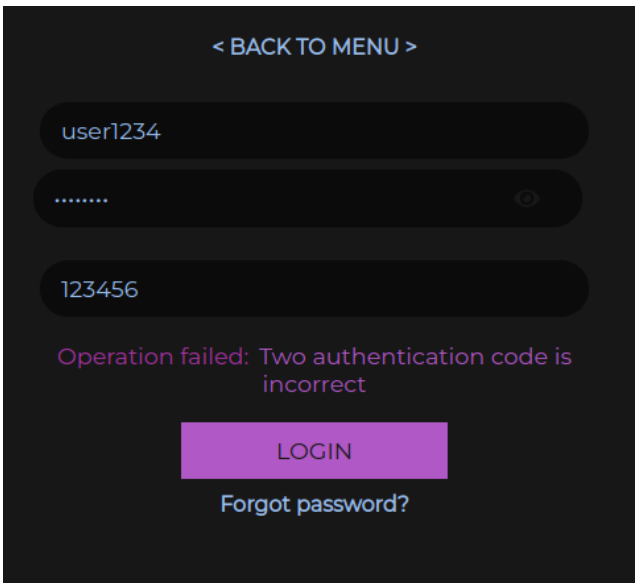
Następnie, aby sprawdzić działanie mechanizmu logowania, wykonano próbę logowania na nieistniejące konto użytkownika (patrz rysunek Rys. 36), próbę logowania przy użyciu niepoprawnego hasła (patrz rysunek Rys. 37), próbę logowania z użyciem błędnego kodu weryfikacji dwuetapowej (patrz rysunek Rys. 38) oraz próbę logowania na nieaktywne konto (patrz rysunek Rys. 39). W każdym z powyższych przypadków system wyświetla odpowiedni błąd. Na podstawie poniższych rysunków można stwierdzić, iż omawiany test zakończył się sukcesem.



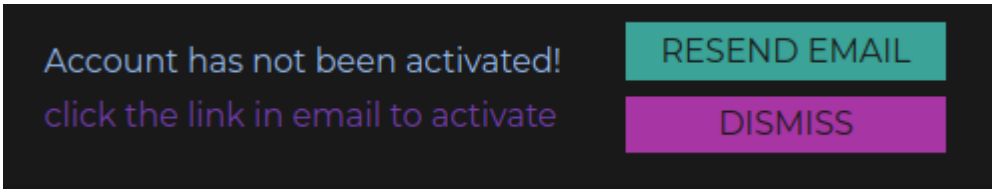
Rys. 36. Próba logowania na nieistniejącego użytkownika



Rys. 37. Próba logowania przy użyciu niepoprawnego hasła



Rys. 38. Próba logowania przy użyciu niepoprawnego kodu weryfikacji dwuetapowej

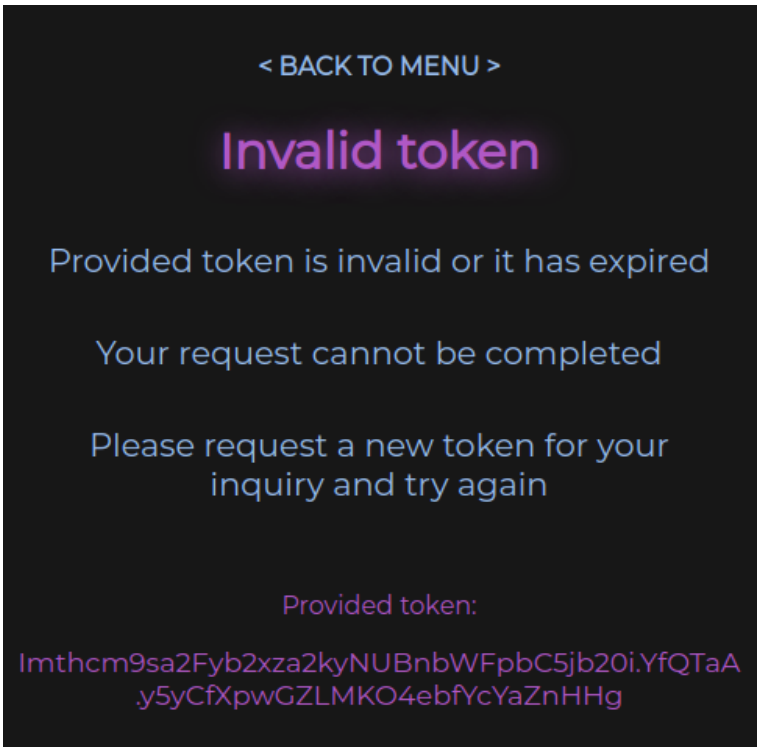


Rys. 39. Próba logowania na nieaktywne konto

5.2 Testy weryfikacji tokenu

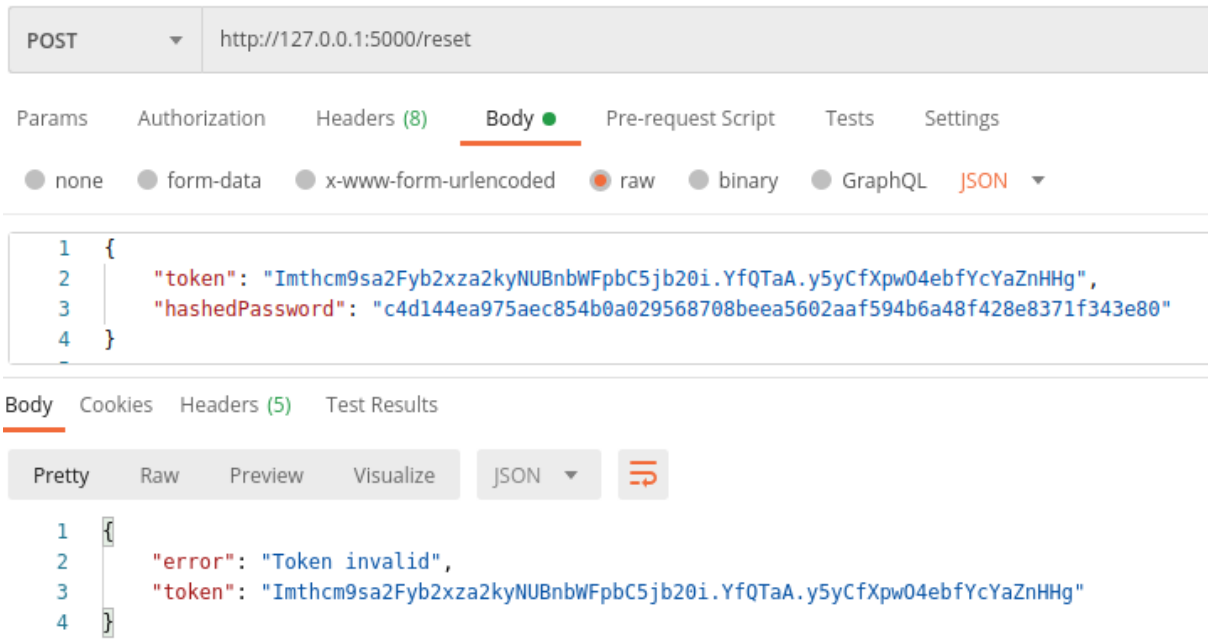
Wyniki testów przeprowadzonych pod kątem weryfikacji poprawności wygenerowanego tokenu pozwalającego na aktywację oraz resetowanie hasła zostały przedstawione poniżej. Jako pierwsza przeprowadzona została próba aktywacji konta (patrz rysunek Rys. 40), a następnie próba zmiany hasła (patrz rysunek Rys. 41, rysunek Rys. 42 oraz rysunek Rys. 43).

5.2.1 Test aktywacji konta z użyciem błędnego tokenu

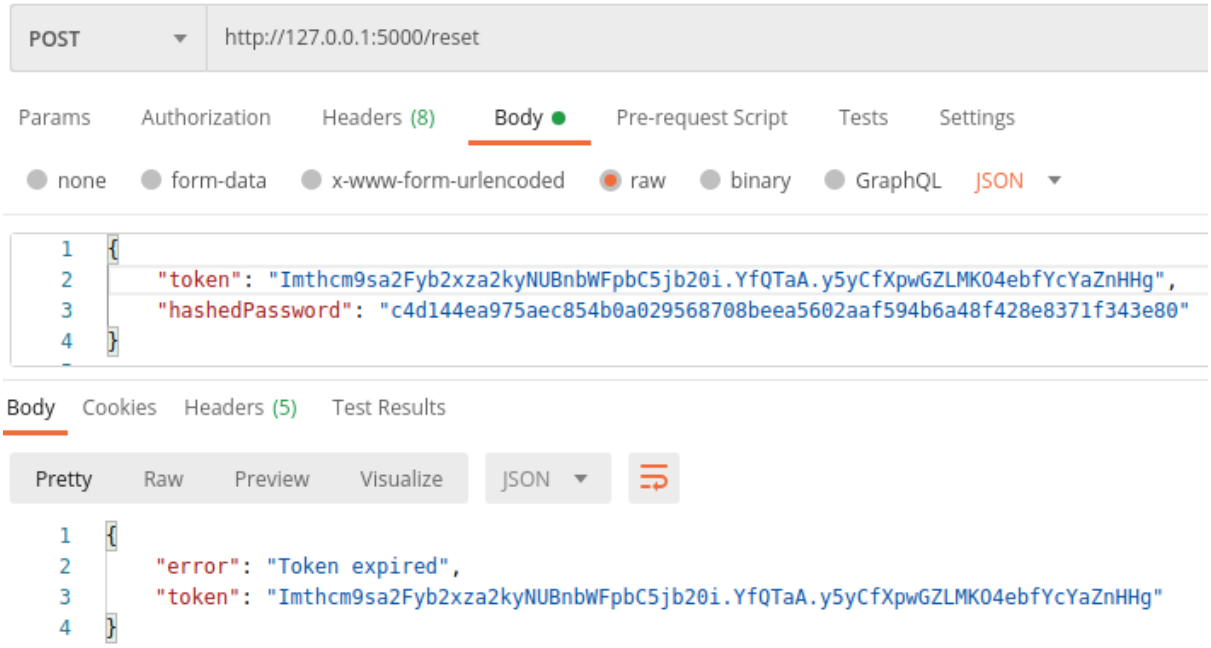


Rys. 40. Aktywowanie konta niepoprawnym tokenem

5.2.2 Test resetowania hasła z użyciem błędnego tokenu

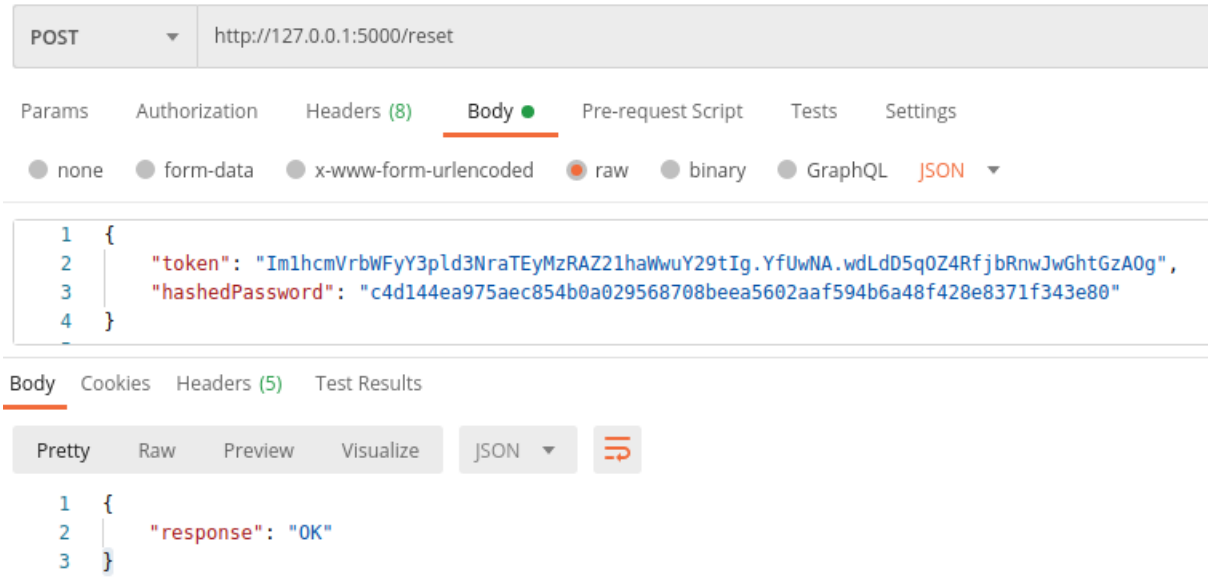


Rys. 41. Resetowanie hasła niepoprawnym tokenem



Rys. 42. Resetowanie hasła wygasłym tokenem

5.2.3 Test resetowania hasła z użyciem poprawnego tokenu



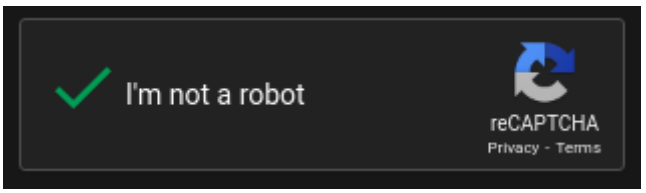
Rys. 43. Resetowanie hasła poprawnym tokenem

5.3 Weryfikacja mechanizmu *CAPTCHA*

Kolejnym przeprowadzonym testem był test pozwalający zweryfikować działanie mechanizmu *CAPTCHA* podczas procesu tworzenia nowego konta. Na rysunkach Rys. 44 oraz Rys. 45 został pokazany wynik testu.

Rys. 44. Błąd weryfikacji

Na powyższym rysunku można zauważyć, iż próba utworzenia nowego konta w serwisie bez zaznaczania odpowiedniego pola w mechanizmie *CAPTCHA* spowodowało wyświetlenie odpowiedniego komunikatu, a nowe konto nie zostało utworzone.



Rys. 45. Poprawna weryfikacja

Po potwierdzeniu, że nowo zakładane konto nie jest tworzone przez program komputerowy, zostanie ono pomyśle założone, zakładając, że wprowadzone dane są poprawne.

5.4 Testy doboru przeciwnika

W ramach testu poprawności działania mechanizmu doboru przeciwnika zalogowano się równocześnie na trzy testowe konta graczy o różnej liczbie punktów ELO (patrz Tabela 16).

Tabela 16. Testowe konta użytkowników

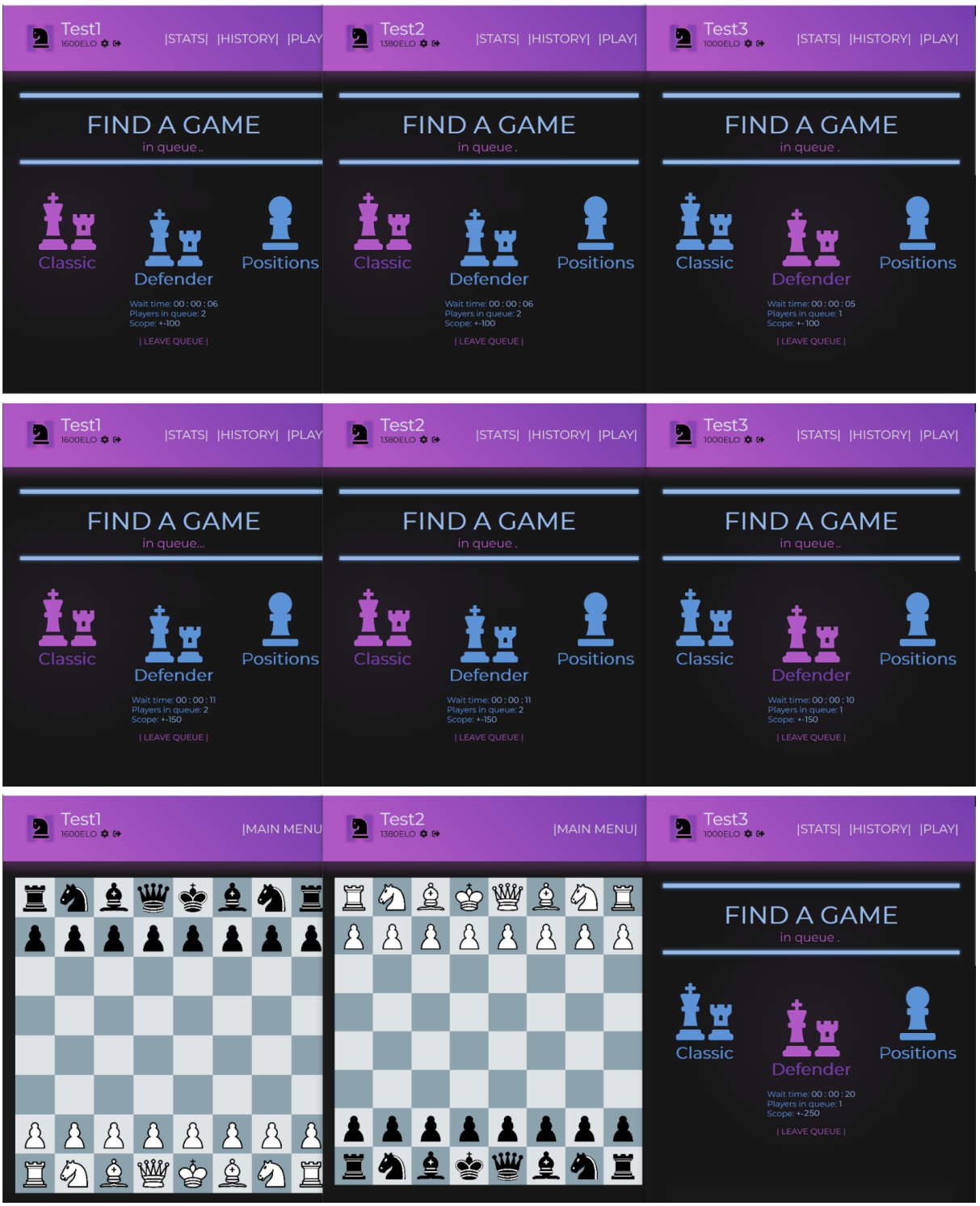
Nazwa użytkownika	Liczba punktów ELO
Test1	1600
Test2	1390
Test3	1000

Po stronie serwera ustawiono następujące wartość zmiennych kolejkowania (patrz sekcja 4.1.1):

- *InitialScope* - 50,
- *ScopeUpdateInterval* - 5000ms (5s),
- *ScopeUpdateAmmount* - 50.

Następnie jako ‘Test 1’ i ‘Test 2’ dołączono do kolejki trybu klasycznego, natomiast jako ‘Test3’ do kolejki trybu *defender*. Spodziewanym, oraz zaobserwowanym, zachowaniem

było dobranie do rozgrywki gracza ‘Test1’ i ‘Test2’ po upływie 20 sekund, podczas gdy gracz ‘Test3’ wciąż oczekiwał na przeciwnika (patrz Rys. 46.).



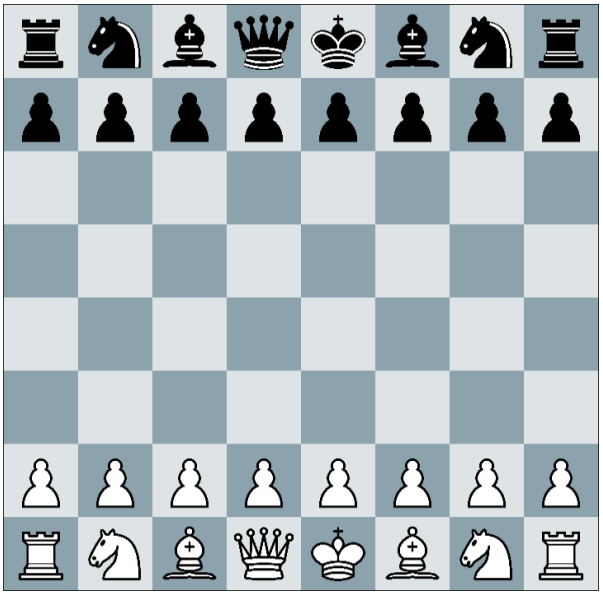
Rys. 46. Przebieg testu doboru przeciwnika

5.5 Testy silnika szachowego po stronie klienckiej

Do testowania silnika szachowego zostały wykonane wybrane testy *PERFT* [9] (ang. *performance test, move path enumeration*). W tych testach dla danych pozycji startowych

porównuje się liczbę możliwych do wykonania ruchów, do tych, generowanych przez inne silniki. Sprawdzane są kolejne tzw. głębokości przewidywań, czyli ilość kolejnych rozpatrywanych tur. W przypadku omawianej pracy, przez to w jaki sposób są generowane ruchy, zostaną sprawdzone trzy głębokości. Poniżej przedstawiono kilka z wykonanych testów *PERFT*.

Wykonano test dla pozycji startowej klasycznych szachów, zapisanej jako FEN - rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 (patrz Rys. 47.). Wyniki testu i porównanie z wynikami generowanymi przez inne silniki zostały przedstawione w tabeli 17.

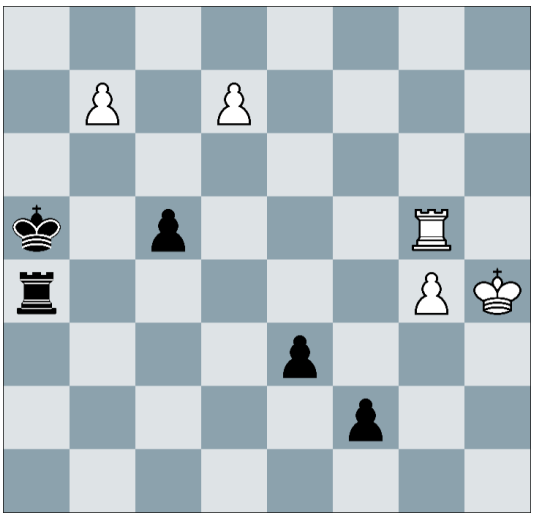


Rys. 47. FEN rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1 na szachownicy

Tabela 17. Wynik testu PERF dla pozycji startowej klasycznych szachów

Głębokość	Uzyskana ilość ruchów	Prawidłowa ilość ruchów
1	**20**	20
2	**400**	400
3	**8902**	8902

Wykonano też test dla pozycji, zapisanej jako FEN - 8/2p5/3p4/KP5r/1R3p1k/8/4P1P1/8 w - - (patrz Rys. 48.). Wyniki testu i porównanie z wynikami generowanymi przez inne silniki zostały przedstawione w tabeli 18.

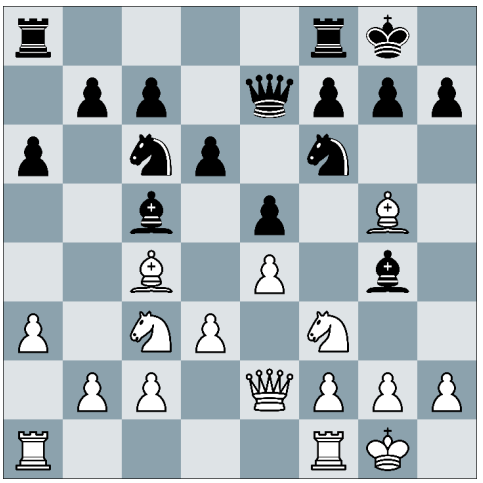


Rys. 48. FEN 8/2p5/3p4/KP5r/1R3p1k/8/4P1P1/8 w - - na szachownicy

Tabela 18. Wynik testu PERF dla FEN 8/2p5/3p4/KP5r/1R3p1k/8/4P1P1/8 w - -

Głębokość	Uzyskana ilość ruchów	Prawidłowa ilość ruchów
1	**14**	14
2	**191**	191
3	**2812**	2812

Wykonano także test dla pozycji zapisanej jako FEN - r3k2r/Pppp1ppp/1b3nbN/nP6/BBP1P3/q4N2/Pp1P2PP/R2Q1RK1 w kq - 0 1 (patrz Rys. 49.). Wyniki testu i porównanie z wynikami generowanymi przez inne silniki zostały przedstawione w tabeli 19.



Rys. 49. FEN r3k2r/Pppp1ppp/1b3nbN/nP6/BBP1P3/q4N2/Pp1P2PP/R2Q1RK1 w kq - 0 1 na szachownicy

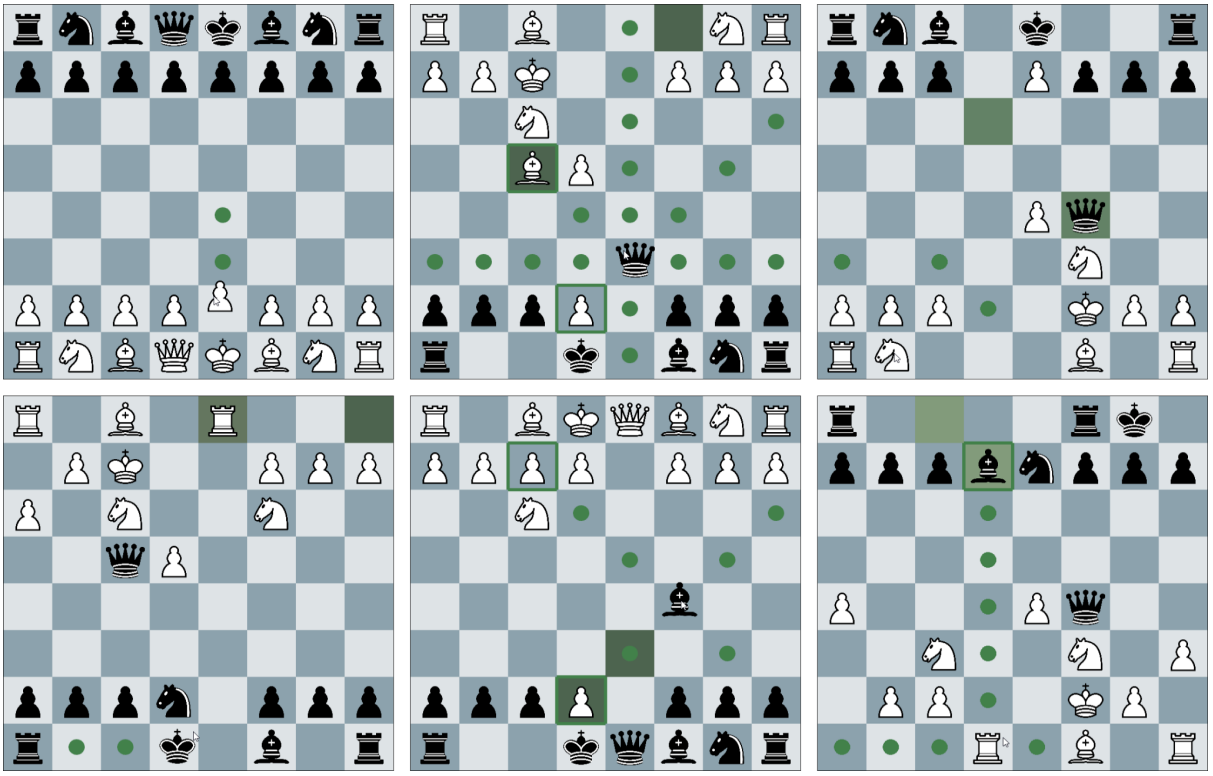
Tabela 19. Wynik testu PERF dla FEN
r3k2r/Pppp1ppp/1b3nbN/nP6/BBP1P3/q4N2/Pp1P2PP/R2Q1RK1 w kq - 0 1

Głębokość	Uzyskana ilość ruchów	Prawidłowa ilość ruchów
1	**46**	46
2	**2079**	2079
3	**89890**	89890

Początkowe przeprowadzenie testów PERF pozwoliło wykryć kilka z niedoskonałości w zaimplementowanym silniku szachowym. Po naprawie tych błędów, testy zostały przeprowadzone ponownie, tym razem z pozytywnym wynikiem.

5.6 Testy graficznego interfejsu rozgrywki szachowej

W ramach testów graficznego interfejsu rozgrywki szachowej została ręcznie przeprowadzona seria partii. W każdej z partii zwracano uwagę, na responsywność planszy, oraz podświetlanie możliwych ruchów (patrz Rys. 49.). W trakcie przeprowadzania testów, nie zostały wykryte żadne błędy.



Rys. 50. Ruchy podpowiadane w interfejsie graficznym dla każdej z bierek

6. PODSUMOWANIE

Głównym celem projektu było zaprojektowanie oraz zaimplementowanie aplikacji pozwalającej na rozegranie gry opartej na zasadach szachowych za pośrednictwem strony internetowej.

Aplikacja pozwala na rozgrywkę szachową dla dwóch graczy w trybie klasycznym oraz w trybie *defender*. Zaimplementowany został także tryb dla jednego gracza, umożliwiający rozgrywkę ze sztuczną inteligencją.

Użytkownik może się zarejestrować, zalogować i wylogować. Aplikacja umożliwia zarządzanie danymi podanymi w momencie rejestracji. Użytkownik ma możliwość edycji hasła, maila czy założenie dwuetapowej weryfikacji. Dodatkowo użytkownik ma prawo do "bycia zapomnianym", czyli usunięcia wszelkich przechowywanych danych na jego temat.

Aplikacja pozwala na dołączenie do kolejki dla każdego z trybów rozgrywki, aby potem dobrać przeciwnika na jak najbardziej zbliżonym poziomie umiejętności. Jeżeli użytkownik zostanie rozłączony w trakcie gry, będzie miał możliwość powrotu do rozgrywki za pomocą ciasteczek ustawianych w przeglądarce.

Podsumowując, śmiało można stwierdzić, że wszystkie początkowe założenia zostały zrealizowane, a nawet aplikacja została rozszerzona o nowe pomysły, powstałe w trakcie implementowania projektu.

6.1 Napotkane problemy

Podczas rozwijania aplikacji, każdy z uczestników projektu napotkał na mniej lub bardziej skomplikowane problemy, które starał się rozwiązać w jak najlepszy sposób. Niekiedy napotkane problemy wymagały konsultacji z innymi członkami zespołu.

W początkowej fazie implementacji komunikacji między klientem a serwerem (patrz sekcja 4.15) dużym problemem była obsługa *CORS* wysyłanych zapytań. Serwer odrzucał wszystkie wysyłane zapytania, ponieważ były niezgodne z polityką *CORS*. Zmotywowało to zespół projektowy do poszerzenia swojej wiedzy na ich temat. Co ostatecznie pozwoliło nie tylko usunąć występujące błędy ale także pozwoliło członkom zespołu docenić znaczenie jakie zgodność z polityką *CORS* odgrywa z bezpieczeństwie sieciowym.

Problematyczny okazał się także interfejs graficzny szachownicy, który początkowo został stworzony bez komunikacji z serwerem. Okazało się, że przerobienie tak zaimplementowanego interfejsu było trudniejsze niż zakładano ponieważ aplikacja kliencka musiała pobierać z serwera dużo więcej danych do prawidłowego przeprowadzenia rozgrywki niż zakładano. Jednym z takich przykładów, może być opisane wcześniej bicie w przelocie (patrz sekcja 4.16.3). W trakcie przeprowadzania testów, okazało się, że samo ułożenie bierek nie jest wystarczające, aby powiedzieć czy bicie w przelocie jest możliwe. Dlatego też, informacja o tym musiała być przechowywana po stronie serwera, i w momencie odświeżenia strony, od niego pobierana. Zmieniony musiał być też sposób przechowywania informacji o tym, kto aktualnie powinien wykonać turę, oraz ruchy wykonane po stronie klienckiej musiały być potwierdzanie po stronie serwera. Opisane zmiany były tylko niektórymi z tych, które trzeba było zaimplementować aby umożliwić poprawne przeprowadzenie rozgrywki przy pomocy serwera. Można więc stwierdzić,

że implementowanie rozgrywki bez komunikacji z serwerem, było błędem, ponieważ zamiast zaoszczędzić czas, dołożyła więcej problemów.

6.2 Perspektywy dalszego rozwoju

Projekt będący tematem pracy jest aplikacją użytkową, co wpływa na fakt, iż aplikacja może zostać rozszerzona o kolejne funkcjonalności mające na celu wzbogacić system oraz pozyskać kolejnych użytkowników.

Istotnym udogodnieniem dla nowych użytkowników jest możliwość logowania się do konta za pomocą innych platform takich jak Google, Facebook, itp. Dzięki takiemu systemowi proces logowania jest skrócony do minimum.

Podczas prowadzenia rozgrywki szachowej, użytkownicy mają możliwość komunikowania się tylko za pomocą czatu tekstowego. Innowacją wspomagającą porozumiewanie się między graczami byłoby dodanie czatu głosowego udoskonalającego proces komunikacji między sobą.

Należy zaznaczyć, iż aplikacja jest dostarczona w formie aplikacji internetowej przeznaczonej na przeglądarki. Przyszłościowym rozwiązaniem mogłaby się okazać aplikacja przeznaczona na telefony komórkowe, posiadająca przejrzysty interfejs użytkownika.

Przy obecnie zastosowanej architekturze aplikacji (klient-serwer), portal przystosowany jest do rozbudowę oraz dodawanie kolejnych funkcjonalności, bądź udoskonalanie już istniejących.

7. LITERATURA

1. International Chess Federation, FIDE handbook, https://handbook.fide.com/chapter/E012018Opi, 2013, dostęp 12 czerwca 2021

2. Glickman M. Example of the Glicko-2 system, http://www.glicko.net/glicko/glicko2.pdf, , dostęp 30 listopada 2013

3. Kellogg T., Websockets are not magical - Tim Kellogg, https://timkellogg.me/blog/2015/03/01/websockets-are-not-magic, 1 marca 2015, dostęp 15 czerwca 2021

4. Knuth D.E. i Moore R.W. An analysis of alpha-beta pruning, http://www-public.imtbs-tsp.eu/~gibson/Teaching/CSC4504/ReadingMaterial/KnuthMoore75.pdf, 1975, dostęp 27 stycznia 2022

5. Network Working Group. Hypertext Transfer Protocol – HTTP/1.1s. czerwiec 1999. https://tools.ietf.org/html/rfc2616 dostęp 23 stycznia 2022

6. Network Working Group. Simple Mail Transfer Protocol. październik 2008. https://datatracker.ietf.org/doc/html/rfc5321 dostęp 23 stycznia 2022

7. Postman, Inc., Postman, https://www.postman.com/product/tools/, nieznana, dostęp 27 stycznia 2022

8. Romstad T. Kiiski J. oraz Costalba M, About - Stockfish - Open source chess engine, https://stockfishchess.org/about/, nieznana, dostęp 12 czerwca 2021

9. Schultz Brad, Written in Cobol - Program Written as Chess Buff's Research Aid, https://news.google.com/newspapers?nid=849&dat=19780417&id=h8lOAAAAIBAJ&sjid=DEoDAAAAIBAJ&pg=6180,1080528&hl=pl, 17 kwietnia 1978, dostęp 27 stycznia 2022

10. Steven J. Edwards, Portable Game Notation Specification and Implementation Guide, https://www.thechessdrum.net/PGN_Reference.txt, 12 marca 1994, dostęp 12 czerwca 2021

11. nieznany, Czym jest CORS (Cross-Origin Resource Sharing) i jak wpływa na bezpieczeństwo, https://sekurak.pl/czym-jest-cors-cross-origin-resource-sharing-i-jak-wplywa-na-bezpieczenstwo/, 19 grudnia 2018, dostęp 15 czerwca 2021

12. nieznany, Flaga cookie - HttpOnly, https://sekurak.pl/flaga-cookie-httponly/, 4 lipca 2013, dostęp 15 czerwca 2021

13. nieznany, Notacja algebraiczna (FIDE), http://szachy.info.pl/przepisy-gry-w-szachy-fide/notacja-algebraiczna-fide/, nieznana, dostęp 12 czerwca 2021

14. nieznany, Store, https://redux.js.org/api/store, nieznana, dostęp 15 czerwca 2021

15. nieznany, Tokens, https://auth0.com/docs/tokens, nieznana, dostęp 15 czerwca 2021

16. nieznany, Zasady gry w szachy - LubimySzachy.pl, https://lubimyszachy.pl/szachy-zasady/, nieznana, dostęp 12 czerwca 2021

17. nieznany ,Ogólne rozporządzenie o ochronie danych osobowych - *UODO*, https://www.uodo.gov.pl/pl/131/224, 23 Maj 2018, dostęp 27 Stycznia 2022.

8. ZAŁĄCZNIKI

ZAŁĄCZNIK 1

Przykład nagłówków związanych z rozpoczęciem komunikacji *Websocket*

Zapytanie klienta

```
GET /socket.io/?EIO=4&transport=websocket&sid=DxbfsNxRhceO8moiAAAE HTTP/1.1
Host: 127.0.0.1:5000
Connection: Upgrade
Pragma: no-cache
Cache-Control: no-cache
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/91.0.4472.101 Safari/537.36
Upgrade: websocket
Origin: http://localhost:3000
Sec-WebSocket-Version: 13
Accept-Encoding: gzip, deflate, br
Accept-Language: pl-PL,pl;q=0.9,en-US;q=0.8,en;q=0.7
Sec-WebSocket-Key: +YY+9uk2JZDZdLuXWmAc/w==
Sec-WebSocket-Extensions: permessage-deflate; client_max_window_bits
```

Odpowiedź serwera

```
HTTP/1.1 101
Upgrade: WebSocket
Connection: Upgrade
Sec-WebSocket-Accept: 7RbsrEtIrcgbOaycVPDR4vDYC4Q=
```

Całkowita długość: 665 bajtów

ZAŁĄCZNIK 2

Przykład użycia zapisu FEN

Poniżej znajduje się przykład przebiegu gry zakończonej matem zapisany przy pomocy szachowej notacji algebraicznej:

1. e4 d5 2. exd5 Qxd5 3. Nc3 Qa5 4. d4 c6 5. Bc4 Bf5 6. f3 e6 7. Ne2 Nd7 8. Bd2 Qc7 9. g4 Bg6 10. h4 h6 11. Bd3 Bxd3 12. cxd3 Bb4 13. Qb3 Bxc3 14. a3 Bxd2+ 15. Kf2 Qf4 16. g5 Qxf3+ 17. Kg1 Be3+ 18. Kh2 Qxe2+ 19. Kg3 Qf3+ 20. Kh2 hxg5 21. a4 Rxh4#

ZAŁĄCZNIK 3

Przykład użycia zapisu FEN

Poniżej został przedstawiony sposób zmieniania się FEN po wykonaniu konkretnych ruchów, pogrubiono zmieniające się segmenty.

FEN pozycji początkowej:
rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1

po wykonaniu ruchu 1. e4:
rnbqkbnr/pppppppp/8/8/**4P3**/8/**PPPP1PPP**/RNBQKBNR **b** KQkq - 0 1

po wykonaniu ruchu 1. e4 d5:
rnbqkbnr/**ppp1pppp**/8/**3p4**/4P3/8/PPPP1PPP/RNBQKBNR **w** KQkq - 0 **2**

po wykonaniu ruchu 2. Sc3:
rnbqkbnr/ppp1pppp/8/3p4/4P3/**2N5**/PPPP1PPP/**R1BQKBNR b** KQkq - **1** 2

ZAŁĄCZNIK 4

Sesja komunikacji w czasie rzeczywistym podsłuchana w wireshark

..2....`....2..;S....2...66....2..j.5.Y..2..^P..m..2..9&..
..2...8.]....F.7.t...d.X.(.F.#.R.j.L.d.[.?.E.".....d.V.#.X.#.S.d...d..#42["update_scope",{"scope"
:"1000"}].'42["queue_info",{"playersInQueue":"1"}]..4324[]..2.....2....e.3..2..`8M.S..2..H.H
.{.!42["update_scope",{"scope":1050}]..2...^....'42["queue_info",{"playersInQueue":"2"}].}4
2["game_found",{"gameId":"560aa3e6e94314c78236109e209ac79e15e05ec8bf2dcb78300ae
65e720edf9e","playingAs":"w","gameMode":"0"}].;42["update_opponents_socket_status",{"
status":"connected"}].C42["update_timers",{"whiteTime":599.9996812969912,"blackTime":
600}]..2..4.....C42["update_timers",{"whiteTime":598.9999995208345,"blackTime":600}].C
42["update_timers",{"whiteTime":597.9241981427767,"blackTime":600}].C42["update_tim
ers",{"whiteTime":596.9999993253732,"blackTime":600}].B42["update_timers",{"whiteTim
e":595.895346211968,"blackTime":600}].C42["update_timers",{"whiteTime":594.96227928
74579,"blackTime":600}]..2....Q...C42["update_timers",{"whiteTime":593.8966383463121,"
blackTime":600}].C42["update_timers",{"whiteTime":592.9403163719107,"blackTime":600
}].@42["update_timers",{"whiteTime":591.9876045204,"blackTime":600}].C42["update_ti
mers",{"whiteTime":590.9999990608776,"blackTime":600}].C42["update_timers",{"whiteTi
me":589.9999996077386,"blackTime":600}]...
..n@..\u...!..1-...b.....6..Lz..L3...4.. ....2..Lz..B....2.....2..Lz..B....9..2b..L....l..
!.../..'$..T...Xp..]%..Wt..Z#..\s..^y..^y..Yy..[%...#...r...w..^p..Xu..\p...y..Ll...,...2..2b..Lr..Ll..(...L
z....."..A0...0...o..Vo..Zo..>...>...A...?...<`..%...N$..^`..L=....2..../...Q42["update_timers",{"wh
iteTime":589.7779584054952,"blackTime":599.9933410820086}]..4325[].Q42["update_time

rs",{"whiteTime":589.7779584054952,"blackTime":598.9952003523358}].>42["receive_message",{"text":"Witam!","playerName":"Wojcioo"}].Q42["update_timers",{"whiteTime":589.7779584054952,"blackTime":597.9267089670757}].I42["make_move_local",{"startingSquare":12,"targetSquare":28,"mtype":"P"}]]..2.4..1./i!Gb@qQyVfF5.!OK.dUzWq[x_JPK.9h5.5.vS0Q!W:.$.7W1.!.3r.3vQ4r.6Q'.fW/Pe.sQa./.3.vW5.r.1.rVern!.K.sXvKfF^V_.-n!.n!I5o."42["game_ended",{"result":"lost"}]..2.....4328[]..2.....".

Spis rysunków

Spis tabel

Dodatek A

Wraz z pracą zostały dołączone niżej wymienione załączniki w postaci spakowanych archiwów.

- backend.zip - Archiwum zawierające kod aplikacji serwerowej odpowiadający za funkcjonowanie API, zarządzaniem bazą danych, obsługujący ranking szachowy, itp.
- frontend.zip - Archiwum zawierające kod aplikacji klienckiej m.in. pozwalający na prowadzenie rozgrywki szachowej, zarządzanie kontem użytkownika, itp.
- docker.zip - Archwium zawierające plik docker-compose.yml pozwalający na uruchomienie bazy danych w wirtualnym środowisku.