|  |
| --- |
| **Politechnika Wrocławska Wydział Informatyki i Zarządzania Katedra Informatyki** |
| **STUDIA PODYPLOMOWE Technologie Internetowe** |
| **Projekt i implementacja systemu społecznościowego do gry w szachy w sieci internetowej** |
|  |
| Marcin Kużdowicz |

|  |
| --- |
| Wrocław 2015 |

Opiekun pracy: Doc. dr inż. Zbigniew Szpunar

Spis treści

[Streszczenie 5](#_Toc432979831)

[1. Cel i zakres pracy 6](#_Toc432979832)

[2. Opis problemu 6](#_Toc432979833)

[2.1. Wstęp 6](#_Toc432979834)

[2.2. Słownik pojęć biznesowych 7](#_Toc432979835)

[2.3. Opis wycinka rzeczywistości 8](#_Toc432979836)

[3. Specyfikacja wymagań 9](#_Toc432979837)

[3.1 Wymagania funkcjonalne 9](#_Toc432979838)

[3.1.1 Ogólne wymagania systemowe 9](#_Toc432979839)

[3.1.2. Perspektywa administratora 10](#_Toc432979840)

[3.1.3. Perspektywa Użytkownika 10](#_Toc432979841)

[3.2. Wymagania niefunkcjonalne 11](#_Toc432979842)

[3.3. Diagram przypadków użycia 12](#_Toc432979843)

[4. Projekt systemu 13](#_Toc432979845)

[4.1 Wybór technologii 13](#_Toc432979846)

[4.2 Architektura aplikacji 16](#_Toc432979847)

[4.2.1. Ogólna architektura 16](#_Toc432979848)

[4.2.2. Drzewo plików 17](#_Toc432979849)

[4.2.3 Diagram klas 22](#_Toc432979850)

[4.2.4 Model bazy danych 24](#_Toc432979851)

[4.2.5. Operacje na danych 25](#_Toc432979852)

[5. Realizacja systemu 29](#_Toc432979853)

[5.1. Tworzenie kont użytkowników 29](#_Toc432979854)

[5.2 Autentykacja i uwierzytelnianie 29](#_Toc432979855)

[5.2.1. Dostęp do zasobów całej podstrony z pozycji url 30](#_Toc432979856)

[5.2.2. Dostęp do zasobów na stronie internetowej 32](#_Toc432979857)

[5.3 Implementacja gry w szachy 33](#_Toc432979858)

[5.4 Komunikator do gry w szachy 37](#_Toc432979859)

[5.4.1. Odbiór wiadomości po stronie serwera i po stronie klienta 37](#_Toc432979860)

[5.4.2. Protokół komunikacji 44](#_Toc432979861)

[5.4.2.1 Opis przebiegu komunikacji pomiędzy graczami 47](#_Toc432979862)

[5.4.2.2 Projekt komunikacji pomiędzy graczami 47](#_Toc432979863)

[6. Testy 50](#_Toc432979864)

[7. Prezentacja aplikacji 54](#_Toc432979865)

[8. Podsumowanie 62](#_Toc432979866)

[Literatura 64](#_Toc432979867)

[Spis ilustracji 65](#_Toc432979868)

[Załączniki 65](#_Toc432979869)

# Streszczenie

Celem pracy jest projekt i implementacja systemu społecznościowego do gry w szachy w sieci internetowej . W pierwszym rozdziale został omówiony cel i zakres pracy. W drugim rozdziale został zdefiniowany zakres problemu, zaprezentowany słownik pojęć biznesowych oraz opisany wycinek rzeczywistości. W rozdziale trzecim zostały zdefiniowane wymagania funkcjonalne i nie funkcjonalne programu oraz przedstawiono diagram przypadków użycia. Rozdział czwarty omawia technologie jakie zostały użyte do implementacji systemu, prezentuje drzewo plików aplikacji oraz przedstawia modele użyte w procesie projektowania systemu, w tym model architektoniczny MVC, diagram klas oraz model bazy danych. Rozdział piąty omawia wybrane elementy implementacji aplikacji, w tym podstawowych elementów systemów webowych takich jak tworzenie konta użytkownika, logowanie, autentykacja, uwierzytelnianie. Następnie są w nim opisane implementacje elementów bardziej jednostkowych dla systemu, takich jak gra w szachy, komunikator do gry, protokół komunikatora do gry w szachy. Rozdział szósty przybliża proces testowania systemu w tym testy jednostkowe, walidacja kodu oraz testy bezpieczeństwa. Pracę kończy rozdział siódmy, który omawia zalety i wady systemu oraz perspektywy dalszego rozwoju.

# Cel i zakres pracy

Celem pracy jest zaprojektowanie i implementacja systemu informatycznego służącego do gry w szachy w sieci internetowej. Program ma być aplikacją webową, korzystającą z cech i właściwości systemów webowych, takich jak: gromadzenia danych w bazie danych, możliwość rejestracji kont użytkownika, logowanie do systemu, prezentowanie wybranych zasobów w zależności od tego jakie uprawnienia ma użytkownik aktualnie korzystający z programu oraz komunikacji w czasie rzeczywistym z innymi użytkownikami online. Założeniem jest iż anonimowy użytkownik aplikacji będzie miał dostęp do pewnej części wizualnej i demonstracyjnej aplikacji, natomiast użytkownik zarejestrowany będzie mógł grać z innymi użytkownikami w szachy w zależności od tego czy inni użytkownicy będą w danym momencie obecni przy swoich komputerach, gdyż ma to być aplikacja z możliwością komunikacji w czasie rzeczywistym. System ma również gromadzić dane statystyczne na temat wyników pojedynków szachowych i część z nich prezentować w interfejsie użytkownika.

# Opis problemu

## Wstęp

Jednym z najciekawszych funkcji współczesnego Internetu jest możliwość komunikacji w czasie rzeczywistym miedzy dwoma osobami lub grupą osób znajdujących się fizycznie w innych miejscach. Funkcjonalność ta może służyć do wymiany informacji między ludźmi w formie komunikatorów typu chat lub skype, przesyłać za ich pośrednictwem dane tekstowe, dźwiękowe, multimedialnych w postaci zdjęć lub filmów. Jak również funkcjonalność ta może one służyć do dalej idących interakcji jak wymianie zachowań na określone bodźce według określonych reguł, tak jak to ma miejsce w przypadku gier społecznościowych online.

Problemem podjętym w niniejszej pracy jest stworzenie takiego systemu, który będzie skupiał określoną grupę użytkowników i umożliwiał im prowadzenie pojedynków szachowych między sobą w czasie rzeczywistym.

Sam temat jest również interesujący wyzwaniem programistycznym, gdyż będzie wymagał użycia szerokiej palety technologii ze względu na osadzenie aplikacji w Internecie, jak i zastosowanie złożonej logiki mającej na celu obsłużenie komunikacji między graczami i zaprojektowanie obiektu wiadomości szachowej.

# Słownik pojęć biznesowych

W poniższej tabeli przedstawiono słownik pojęć biznesowych, który stanowi zbiór zdefiniowanych terminów używanych w opisie i projekcie systemu. Poniższe terminy będą się pojawiać w omawianiu opracowania systemu.

Tabela 1 - Słownik pojęć biznesowych

|  |  |
| --- | --- |
| Anonimowy użytkownik | Użytkownik Internetu nie zarejestrowany w systemie, korzystający z aplikacji |
| Zarejestrowany użytkownik | Użytkownik Internetu posiadający konto użytkownika w systemie, korzystający z aplikacji |
| Administrator | Użytkownik systemu mający uprawnienia do zarządzania innymi użytkownikami |
| Szachy | Rodzaj gry planszowej przeznaczonej dla dwóch osób |
| Szachownica | Obiekt/ strefa, w kontekście której poruszają się gracze |
| Figura szachowa | Obiekt/ pojedynczy element poruszający się po szachownicy. Reprezentowany w systemie w postaci ikony |
| Ruch szachowy | Ruch figury szachowej wykonany przez użytkownika |
| Pojedynek szachowy online | Gra w szachy pomiędzy dokładnie dwoma użytkownikami w czasie rzeczywistym za pomocą oprogramowania |
| Lista najlepszych graczy | Posortowana lista dziesięciu użytkowników, którzy wygrali najwięcej pojedynków szachowych |
| Kolor figur szachowych | Reprezentacja dwóch stanów, zbioru figur szachowych, przyporządkowanych do gracza, która może przyjąć stan: klor biały albo czarny. Figury białe zawsze rozpoczynają grę |
| Status gry | Aktualna sytuacja w jakiej znajduje się gra, np: ruch dla figur białych, szach, szach mat. |
| Fen | Notacja Forsytha-Edwardsa. zapis szachowy dla określenia pozycji gry szachowej. Celem FEN jest podanie wszystkich niezbędnych informacji do ponownego rozpoczęcia gry od danej pozycji. |
| Komunikator do gry online | Część oprogramowania służąca do wysyłania ruchów szachowych użytkowników przez sieć internetową między dwoma użytkownikami. Umożliwiający również zapraszanie do gry oraz aktualizowania wyświetlanych aktualnie dostępnych użytkowników |
| Oczekiwanie na nową grę | Stan w jakim znajduje się użytkownik, który w danym momencie z nikim nie gra i jest online |
| Zaproszenie do gry | Stan w jakim znajduje się użytkownik, który został zaproszony do gry lub sam zaprosił do gry innego użytkownika |
| Stan w trakcie gry | Stan w jakim znajduje się użytkownik, który aktualnie prowadzi grę z innym użytkownikiem |
| Pojedynek szachowy | Dwóch użytkowników grających przeciwko sobie |
| Wykres najlepszych graczy | Wykres słupkowy 10 najlepszych graczy, posortowanych według ilości wygranych pojedynków szachowych |
| Wiadomość od gracza do gracza | Wysłany ruch szachowy i aktualny status gry od jednego użytkownika do drugiego po wykonanym ruchu |

# Opis wycinka rzeczywistości

**Profil działalności**

Użytkownik sieci internetowej, zainteresowany grą w szachy, oraz grami online.

**Wizja systemu**

Dostęp do aplikacji ma się odbywać poprzez sieć internetową. Każdy anonimowy użytkownik Internetu może odwiedzić stronę z aplikacją, lecz ma dostęp tylko do jej niektórych zasobów. Do wszystkich zasobów ma dostęp dopiero po założeniu konta i zalogowaniu. Do zasobów administracyjnych ma dostęp wyłącznie użytkownik z rolą administratora.

System przewiduje dwie role dla zarejestrowanych użytkowników:

1. Administrator
2. Użytkownik

Użytkownikiem można się stać poprzez rejestrację w systemie, za pomocą formularza do rejestracji obecnego w interfejsie użytkownika.

# Specyfikacja wymagań

## 3.1 Wymagania funkcjonalne

### 3.1.1 Ogólne wymagania systemowe

Na podstawie przeprowadzonych badań polegających na zapoznaniu się z aplikacjami pełniącymi podobne przeznaczenie do opracowywanego systemu sformułowano następujące wymagania funkcjonalne:

* dodawanie/usuwanie użytkownika przez administratora
* administrator może przeglądać listę użytkowników
* administrator może tworzyć użytkowników
* administrator może usuwać użytkowników
* możliwość stworzenia konta użytkownika za pomocą formularza rejestracji przez użytkownika anonimowego
* do rejestracji niezbędne będzie podanie: loginu, hasła, potwierdzenia hasła, adresu email
* konto użytkownika po rejestracji musi być zweryfikowanych za pomocą linka aktywacyjnego wysyłanego, na wcześniej podany adres email
* użytkownik po rejestracji nie może zmienić swojego loginu, gdyż login jest identyfikatorem w grze online dla innych użytkowników
* użytkownik może edytować po rejestracji swoje dane takie jak: adres email, imię, nazwisko.
* użytkownik może zmienić hasło
* możliwość gry w szachy z komputerem
* wyniki pojedynków z komputerem nie będą zapisywane do bazy, pełnią rolę warstwy demonstracyjnej.
* możliwość gry w szachy z innym użytkownikiem w czasie rzeczywistym
* zalogowany użytkownik będzie mógł przeglądać historie odbytych przez siebie wcześniej pojedynków szachowych
* w interfejsie użytkownika będzie dostępny rankingu najlepszych dziesięciu graczy w postaci wykresu słupkowego
* pojedynek szachowy ma występować w postaci: pojedynczy użytkownik kontra pojedynczy użytkownik
* użytkownik może prowadzić maksymalnie jedną grę w tym samym czasie
* użytkownik nie może prowadzić gry z kilkoma użytkownikami jednocześnie
* użytkownik przed rozpoczęciem pojedynku musi zaprosić drugiego użytkownika do gry.
* Zaproszony użytkownik ma 15 sekund na decyzje
* w momencie trwania zaproszenia, dana para użytkowników jest zablokowana dla innych użytkowników i inni użytkownicy nie mogą do nich wysłać wiadomości.
* W momencie odmowy na zaproszenie dana para użytkowników zostaje odblokowana dla innych użytkowników pod względem możliwości wysyłania do nich wiadomości
* w momencie trwania gry między dwoma użytkownikami, dana para użytkowników jest zablokowana dla innych użytkowników i inni użytkownicy nie mogą do nich wysłać wiadomości.
* W momencie zakończenia gry dana para użytkowników zostaje odblokowana dla innych użytkowników pod względem możliwości wysyłania do nich wiadomości

### Perspektywa administratora

System w formie aplikacji webowej potrzebuje użytkownika posiadającego uprawnienia do zarządzania utworzonymi kontami użytkowników. Założono iż rola administratora będzie wyposażona w następujące funkcjonalności.

* ma te same uprawnienia co zalogowany użytkownik
* ma dostęp do tych samych zasobów co zalogowany użytkownik
* może dodawać nowych użytkowników
* może usuwać istniejące konta.
* może tworzyć konta z uprawnieniami administratora
* może nadawać uprawnienia administratora lub je odbierać.
* może edytować konta użytkownika

### Perspektywa Użytkownika

Użytkowników systemu nie będący administratorem może przybrać dwie postaci: anonimową oraz użytkownika systemu. Zdefiniowano dla obu typów użytkownika założenia funkcjonalne.

Anonimowy użytkownik ma dostęp do:

* Strony domowej
* możliwości zagrania w szachy z komputerem, aby zapoznać się z systemem.
* wykresu słupkowego z maksymalnie 10 najlepszymi graczami (przy założeniu że w bazie mamy dziesięciu graczy, którzy odbyli zwycięskie pojedynki szachowe. Jeżeli jest ich mniej, to wyświetlana jest mniejsza liczba użytkowników)
* formularza logowania
* formularza z możliwością utworzenia konta

Zalogowany użytkownik ma dostęp do:

* Tych samych zasobów co użytkownik niezalogowany
* Możliwość grania w szachy z innymi użytkownikami online.
* Danych na temat swojego konta i możliwością ich edycji z wyłączeniem zmiany loginu.
* Historii swoich odbytych pojedynków szachowych

## Wymagania niefunkcjonalne

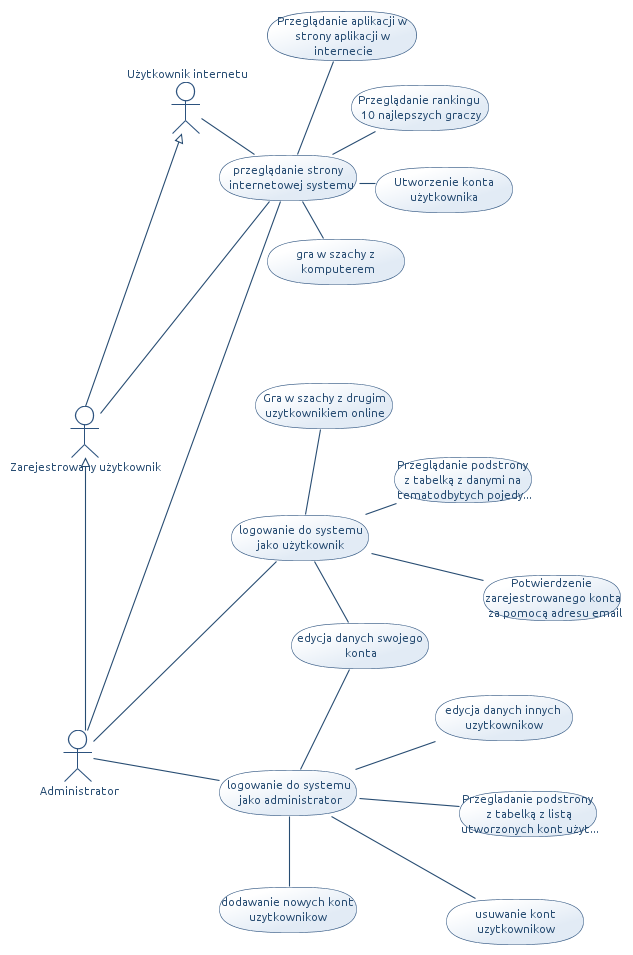
Oprócz wymagań funkcjonalnych założono iż przed systemem stoją następujące wymagania niefunkcjonalne.

* Program ma być aplikacją webową
* Program ma posiadać czytelny interfejs dla użytkownika
* Program ma służyć rozrywce
* Program ma propagować grę w szachy w sieci
* Program ma rozwijać umiejętności logicznego myślenia użytkowników

* 1. Diagram przypadków użycia

Na poniższym rysunku przedstawiono model zdarzeń oraz ich konsekwencji, jakie mogą zaistnieć w systemie z perspektywy trzech rodzajów aktorów, użytkownika anonimowego, użytkownika zalogowanego i administratora systemu.

Rysunek - Diagram przyopadków użycia



# Projekt systemu

## 4.1 Wybór technologii

W aplikacji zastosowano następujące technologie.

* Java – obiektowy język programowania, główna technologia użyta w aplikacji do stworzenia kodu działającego po stronie serwera.
* Spring MVC – framework stworzony w języku java wspierający architekturę Model-View-Controller oraz zawierający gotowe komponenty, które mogą być wykorzystane do implementacji aplikacji webowych.
* JSP (Java server pages) - to technologia umożliwiająca tworzenie dynamicznych dokumentów www z wykorzystaniem języka Java, wplecionego w kod HTML danej strony. Jsp jest formatem pliku w którym można wykorzystać język zarówno HTML jak i Java. Składnia jsp składa się na elementy: html, elementy skryptowe: skryplety (kod w języku Java kontrolujący proces generowania dokumentu), elementy składniowe tzw. Expression Language oraz tagi XML wywołujące określone predefiniowane metody. W przypadku JSP korzysta się najczęściej z predefiniowanych tagów JSTL i Spring security
* JSTL (Java Standard Tag Library) – kolekcja tagów z przypisanymi funkcjonalnościami. JSTL wspiera typowe zadania takie jak iteracja czy konstrukcje warunkowe, wykonywane w warstwie widoku.
* Jetty **-** serwer www oraz kontener servletów Javy.
* Maven **-** narzędzie do zarządzania projektem Javowym. Projekt mavenowy definiuje się poprzez stworzenie i utrzymywanie pliku pom.xml (POM – ang. Project Object Model). **pom.xml** jest głównym miejscem pracy z projektem i zawiera wszystkie istotne elementy definiujące projekt, jego strukturę, sposób budowania i przede wszystkim zależności. Zależności to m.in. zewnętrzne biblioteki importowane do projektu w postaci plików .jar lub zewnętrzne projekty.
* JSON – (Javascript Object Notation) format wymiany danych komputerowych, przypominający swoją strukturą hashmapę, organizując dane w postaci *klucz = wartość*. Jest to format transportowy, który został użyty w aplikacji do wysyłania ruchów szachowych pomiędzy użytkownikami, aktualnego stanu gry, zaproszenia do gry czy zrezygnowania z gry.
* XML (Extensible Markup Language) - rozszerzalny język znaczników. W projektach javowoych bardzo często używany do plików konfiguracyjnych, oraz w plikach jsp w postaci tagów z predefiniowanymi funkcjonalnościami za pomocą zaimportowanych przestrzeni nazw.
* Websocket - jest technologią zapewniającą dwukierunkowy kanał komunikacji (full-duplex) za pośrednictwem jednego połączenia TCP. Websocket jest niezależnym protokółem opartym na protokole TCP. W niniejszej aplikacji ta technologia została użyta do stworzenia komunikatora wysyłającego ruchy szachowe, zaproszenia do gry, rezygnacje z gry, koniec gry jak i listę aktualnie połączonych użytkowników. Dzięki tej technologii gra w szachy zaimplementowana w programie przybrała charakter tzw. *real-time games* (gra czasu rzeczywistego);
* JavaScript - język skryptowy wykonujący się po stronie klienta w przeglądarce, dostarczający programiście możliwości programowania interfejsu użytkownika zdarzeniowo. Interfejs do gry w szachy oraz interfejs komunikatora do gry są oprogramowane zdarzeniowo za pomocą javascript.
* JQuery – biblioteka JavaScript dostarczająca dodatkowe narzędzia i umożliwiająca zaprogramowanie tych samych funkcjonalności za pomocą mniejszej ilości kodu.
* Bootstrap 3 – biblioteka CSS i JavaScript do tworzenia interfejsów użytkownika stworzona przez programistów Twittera. Główną cechą bootsrapa są dostarczane gotowe klasy css, gwarantujące dobry wygląd interfejsu i jego responsywność.
* MongoDB – baza danych typu NoSql.   Charakteryzująca się dużą skalowalnością, wydajnością oraz brakiem ściśle zdefiniowanej struktury, co w przypadku implementacji niniejszego systemu było zaletą. W MongoDb dane składowane są jako dokumenty typu JSON. Do aplikacji w rodzaju live-game baza danych tego typu stanowi dobry wybór ze względu na szybkość działania i możliwość dynamicznego dostosowywania się tabel/kolekcji w bazie pod wpływem zmian jakie są dyktowane przez kod Java, bez konieczności zmiany struktury bazy danych za pomocą dodatkowych zapytań modelujących. Przykładowo gdy zachodzi potzreba do tzw. Kolekcji (odpowiednik tabelki w bazach typu sql) dodać dodatkowe pole, to nie jest konieczne uprzednie wykonywanie polecenia w rodzaju ALTER TABLE, ono samo się utworzy przy pierwszym zapisie obiektu posiadającego dodatkowe pole, co jest bardzo wygodne.
* Html 5 - jest najnowszą wersją standardu opisującego język HTML. Zalety tej technologii zostały użyte przy walidacji formularzy po stronie klienta np. za pomocą atrybutu *required* czy *type=”email”* oraz w przekazywaniu danych do javascriptu za pomocą *data attributes* np. *data-username=”admin”*.
* CSS 3 – kaskadowe arkusze stylów, język realizujący wygląd aplikacji.
* DataTables – plugin do jquery umożliwający szybkie oprogramowanie tabelki wraz z paginacją zaimplementowaną po stronie klienta w javascript.
* Canvasjs – biblioteka javascript umożliwiająca wyświetlenie responsywnych histogramów, słupków, wykresów prezentujących dane statystyczne w przyjaznym dla użytkownika formacie. W programie użyta do wyświetlenia rankingu 10 najlepszych graczy w postaci wykresu słupkowego.
* Chessboardjs - biblioteka javascript dostarczająca interfejs graficzny do gry w szachy.
* Chess.js - biblioteka javascript implementująca logikę gry w szachy umożliwiająca integrację z Chessboardjs oraz zaprogramowanie gry w szachy zdarzeniowo.
* GIT – System kontroli wersji umożliwiający zarządzanie poszczególnymi wersjami aplikacji i przechowywanie wszystkich wersji na lokalnym i zdalnym repozytorium.
* JUnit – napisana w języku Java służąca do przeprowadzania testów jednostkowych
* Skipfish – skaner bezpieczeństwa dla aplikacji webowych
* Hibernate validator – biblioteka napisana w języku Java wspierająca efektywną walidacje formularzy w aplikacjach webowych

## 4.2 Architektura aplikacji

### 4.2.1. Ogólna architektura

Program jest oparty na wzorcu architektonicznym MVC (model view controller) zaimplementowanym przy pomocy komponentów dostarczonych przez framework Spring MVC. Model-View-Controller zakłada podział aplikacji na trzy główne części:

* **Model** – jest pewną reprezentacją problemu bądź logiki aplikacji.
* **Widok** – opisuje, jak wyświetlić pewną część modelu w ramach interfejsu użytkownika. Może składać się z podwidoków odpowiedzialnych za mniejsze części interfejsu.
* **Kontroler** – przyjmuje dane wejściowe od użytkownika i reaguje na jego poczynania, zarządzając aktualizacje modelu oraz odświeżenie widoków.

Obraz modelu MVC widać na poniższym rysunku.

Rysunek - model MVC



### 4.2.2. Drzewo plików

Poniższy schemat prezentuje jak wygląda architektura systemu z punktu widzenie podziału kodu na foldery pakiety, foldery z bibliotekami i pliki z kodem źródłowym. Na zaprezentowanym modelu widać wpływ architektury MVC na drzewo plików w projekcie, co zostanie przybliżone w kolejnym podrozdziale.

**src.**

├── main

│   ├── java

│   │   └── com

│   │   └── chessApp

│   │   ├── controllers

│   │   │   ├── AdminPanelController**.**java

│   │   │   ├── HomeController**.**java

│   │   │   ├── LoginController**.**java

│   │   │   ├── PlayChessSitesController**.**java

│   │   │   ├── RegistrationConfirmationController**.**java

│   │   │   ├── SignInController**.**java

│   │   │   ├── UserPanelController**.**java

│   │   │   └── UsersAjaxController**.**java

│   │   ├── dao

│   │   │   ├── ChessGamesRepository**.**java

│   │   │   └── UsersRepository**.**java

│   │   ├── daoimpl

│   │   │   ├── ChessGamesRepositoryImpl**.**java

│   │   │   └── UsersRepositoryImpl**.**java

│   │   ├── exceptions

│   │   │   └── UserNotConfirmedException**.**java

│   │   ├── gameProtocol

│   │   │   ├── ChessColor**.**java

│   │   │   ├── ChessMoveStatus**.**java

│   │   │   ├── GameMessageProtocol**.**java

│   │   │   ├── GameMessageType**.**java

│   │   │   └── GameUserCommunicationStatus**.**java

│   │   ├── mailService

│   │   │   ├── MailService**.**java

│   │   │   └── MailSubjectPL**.**java

│   │   ├── model

│   │   │   ├── ChessAppUser**.**java

│   │   │   ├── ChessGame**.**java

│   │   │   ├── ChessMove**.**java

│   │   │   ├── GameMessage**.**java

│   │   │   ├── GameUser**.**java

│   │   │   └── UserAccount**.**java

│   │   ├── props

│   │   │   └── ChessAppProperties**.**java

│   │   ├── security

│   │   │   ├── LocalAuthenticationProvider**.**java

│   │   │   ├── PasswordEncryptor**.**java

│   │   │   └── UserRoles**.**java

│   │   ├── taskService

│   │   │   └── ScheduledTaskService**.**java

│   │   └── websocket

│   │   ├── ChessGamesHandler**.**java

│   │   ├── GameUsersHandler**.**java

│   │   ├── WebSocketServer**.**java

│   │   └── WebSocketSessionHandler**.**java

│   ├── resources

│   │   ├── chessApp**.**properties

│   │   └── log4j**.**xml

│   └── webapp

│   ├── assets

│   │   ├── css

│   │   │   ├── dataTable**.**css

│   │   │   ├── fonts

│   │   │   │   ├── glyphicons**-**halflings**-**regular**.**eot

│   │   │   │   ├── glyphicons**-**halflings**-**regular**.**svg

│   │   │   │   ├── glyphicons**-**halflings**-**regular**.**ttf

│   │   │   │   ├── glyphicons**-**halflings**-**regular**.**woff

│   │   │   │   └── glyphicons**-**halflings**-**regular**.**woff2

│   │   │   ├── images

│   │   │   │   ├── favicon**.**ico

│   │   │   │   ├── sort\_asc\_disabled**.**png

│   │   │   │   ├── sort\_asc**.**png

│   │   │   │   ├── sort\_both**.**png

│   │   │   │   ├── sort\_desc\_disabled**.**png

│   │   │   │   ├── sort\_desc**.**png

│   │   │   │   └── Sorting icons**.**psd

│   │   │   ├── lib

│   │   │   │   ├── bootstrap**.**min**.**css

│   │   │   │   ├── bootstrap**-**theme**.**min**.**css

│   │   │   │   ├── chessboard**-**0.3.0.css

│   │   │   │   └── jquery**.**dataTables**.**min**.**css

│   │   │   └── style**.**css

│   │   ├── icons

│   │   │   └── myicon**.**png

│   │   ├── images

│   │   │   ├── chess2**.**jpg

│   │   │   ├── chess**.**jpg

│   │   │   └── chesspieces

│   │   │   └── wikipedia

│   │   │   ├── bB**.**png

│   │   │   ├── bK**.**png

│   │   │   ├── bN**.**png

│   │   │   ├── bP**.**png

│   │   │   ├── bQ**.**png

│   │   │   ├── bR**.**png

│   │   │   ├── wB**.**png

│   │   │   ├── wK**.**png

│   │   │   ├── wN**.**png

│   │   │   ├── wP**.**png

│   │   │   ├── wQ**.**png

│   │   │   └── wR**.**png

│   │   └── js

│   │   ├── bestPlayersMain**.**js

│   │   ├── chessComputerVsComputer**.**js

│   │   ├── chessUserVsComputer**.**js

│   │   ├── chessUserVsUser**.**js

│   │   ├── lib

│   │   │   ├── bootstrap**.**min**.**js

│   │   │   ├── chessboard**-**0.3.0.js

│   │   │   ├── chess**.**js

│   │   │   ├── jquery**-**2.1.4.min.js

│   │   │   ├── jquery**.**canvasjs**.**min**.**js

│   │   │   └── jquery**.**dataTables**.**min**.**js

│   │   ├── main**.**js

│   │   ├── playChessWithUserMain**.**js

│   │   ├── userGamesHistory**.**js

│   │   ├── websocketClientEndpoint**.**js

│   │   ├── wsClientEndpointFunctions**.**js

│   │   └── wsClientMessageExchangeProtocol**.**js

│   ├── index**.**jsp

│   └── WEB**-**INF

│   ├── application**-**context**.**xml

│   ├── mongo**-**config**.**xml

│   ├── mvc**-**config**.**xml

│   ├── scheduled**-**tasks**-**config**.**xml

│   ├── security**-**config**.**xml

│   ├── views

│   │   ├── adminAccount**.**jsp

│   │   ├── bestPlayers**.**jsp

│   │   ├── confirmRegistrationMessage**.**jsp

│   │   ├── creatAccountMessage**.**jsp

│   │   ├── editUser**.**jsp

│   │   ├── error**.**jsp

│   │   ├── home**.**jsp

│   │   ├── includes

│   │   │   ├── footer**.**jsp

│   │   │   ├── forms

│   │   │   │   ├── addUserForm**.**jsp

│   │   │   │   ├── editUserForm**.**jsp

│   │   │   │   ├── editYourAccountFormADMIN**.**jsp

│   │   │   │   ├── editYourAccountForm**.**jsp

│   │   │   │   ├── logInForm**.**jsp

│   │   │   │   └── signInForm**.**jsp

│   │   │   ├── header**.**jsp

│   │   │   ├── menu**.**jsp

│   │   │   ├── modal\_boxes

│   │   │   │   ├── game\_handshake\_invitaion\_modal**.**jsp

│   │   │   │   ├── game\_handshake\_response\_modal**.**jsp

│   │   │   │   ├── removeUserModal**.**jsp

│   │   │   │   ├── user\_end\_chess\_game\_pos\_modal**.**jsp

│   │   │   │   └── user\_info\_modal**.**jsp

│   │   │   ├── tables

│   │   │   │   ├── userGamesTable**.**jsp

│   │   │   │   └── usersTable**.**jsp

│   │   │   ├── usersSiteWithAddUserFormActive**.**jsp

│   │   │   └── usersSiteWithUsersTableActive**.**jsp

│   │   ├── login**.**jsp

│   │   ├── playChessWithComputer**.**jsp

│   │   ├── playChessWithUser**.**jsp

│   │   ├── signIn**.**jsp

│   │   ├── userGames**.**jsp

│   │   ├── users**.**jsp

│   │   └── yourAccount**.**jsp

│   └── web**.**xml

└── test

├── java

└── resources

└── log4j**.**xml

4.2.2.1 Zastosowanie Modelu MVC w systemie

W niniejszym podrozdziale zostanie omówione rozmieszczenie plików implementujących model MVC w projekcie. Wszystkie klasy Javowe znajdują się w pakiecie **src/main/Java**

**Kontroler**

Klasy implementujące kontroler znajdują się w pakiecie **com.chessApp.controllers**.

Klasy te zawierają anotację @Controller co informuje framework Spring o tym, że ta klasa jest kontrolerem i ma obsługiwać żądania http w zależności od tego jaki url zostanie wprowadzony przez użytkownika.

Przykład kodu kontrolera:

@Controller

public class PlayChessSitesController **{**

private static final Logger logger **=** Logger

**.**getLogger**(**PlayChessSitesController**.**class**);**

@RequestMapping**(**value **=** "/play-chess-with-computer"**,** method **=** RequestMethod**.**GET**)**

public ModelAndView playChessWithComputer**()** **{**

logger**.**info**(**"playChessWithComputer()"**);**

ModelAndView playChessWithCompPageModel **=** **new** ModelAndView**(**"playChessWithComputer"**);**

addBasicObjectsToModelAndView**(**playChessWithCompPageModel**);**

**return** playChessWithCompPageModel**;**

**}**

@RequestMapping**(**value **=** "/play-chess-with-user"**,** method **=** RequestMethod**.**GET**)**

public ModelAndView playChessWithUser**()** **{**

logger**.**info**(**"playChessWithUser()"**);**

ModelAndView playChessWithUserPageModel **=** **new** ModelAndView**(**

"playChessWithUser"**);**

addBasicObjectsToModelAndView**(**playChessWithUserPageModel**);**

**return** playChessWithUserPageModel**;**

**}**

private void addBasicObjectsToModelAndView**(**ModelAndView modelAndView**)** **{**

Authentication auth **=** SecurityContextHolder**.**getContext**()**

**.**getAuthentication**();**

String userLogin **=** auth**.**getName**();**

modelAndView**.**addObject**(**"currentUserName"**,** userLogin**);**

**}**

**}**

**Widok**

Zrealizowany za pomocą technologii: Jsp (java server pages), Jstl (JavaServer Pages Standard Tag Library), HTML 5, CSS 3, JavaScript i Jquery oraz Bootstrap . Pliki widoków znajdują się w głównym folderze: **/src/main/webapp/ WEB-INF/** **views**

w folderze **/src/main/webapp/assets** znajdują się pliki css i javascript

w folderze **assets/js** znajdują się pliki źródłowe javascript, w folderze **assets/js/lib** są biblioteki javascript

w folderze **assets/css** znajdują się pliki źródłowe css, w folderze **assets/css/lib** są biblioteki css

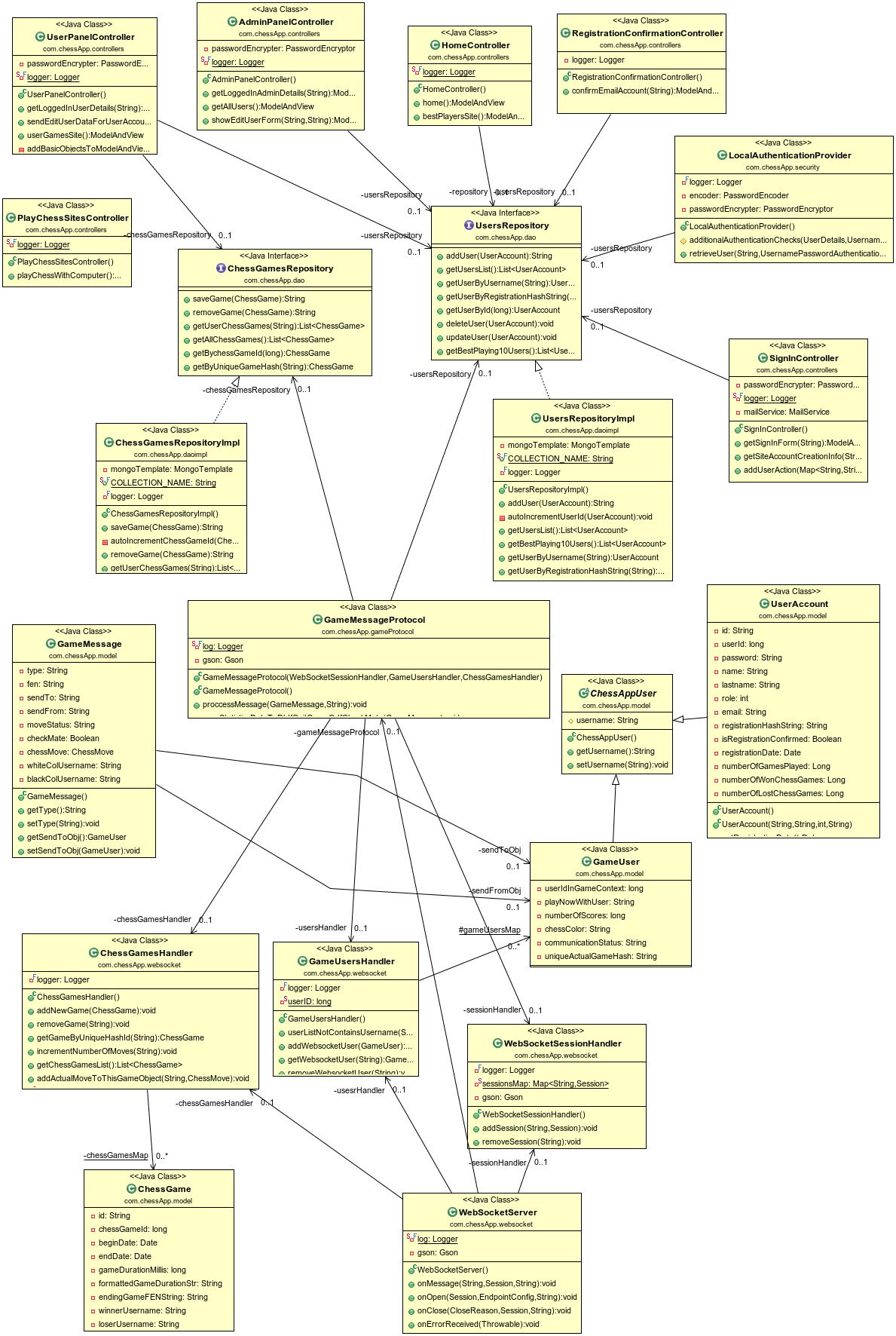
**Model**

Model jest reprezentacją perzystencji aplikacji, warstwy reprezentującej bazę danych w kodzie Java oraz obiektów używanych do transportowania danych pomiędzy komponentami systemu. Implementacja modelu znajduje się w pakiecie **com.chessApp.model.**

4.2.3 Diagram klas

Poniższy rysunek prezentuje, skrócony diagram klas aplikacji. Pełny diagram klas znajduje się w załącznikach w pliku: ChessAppDiagramKlas.png

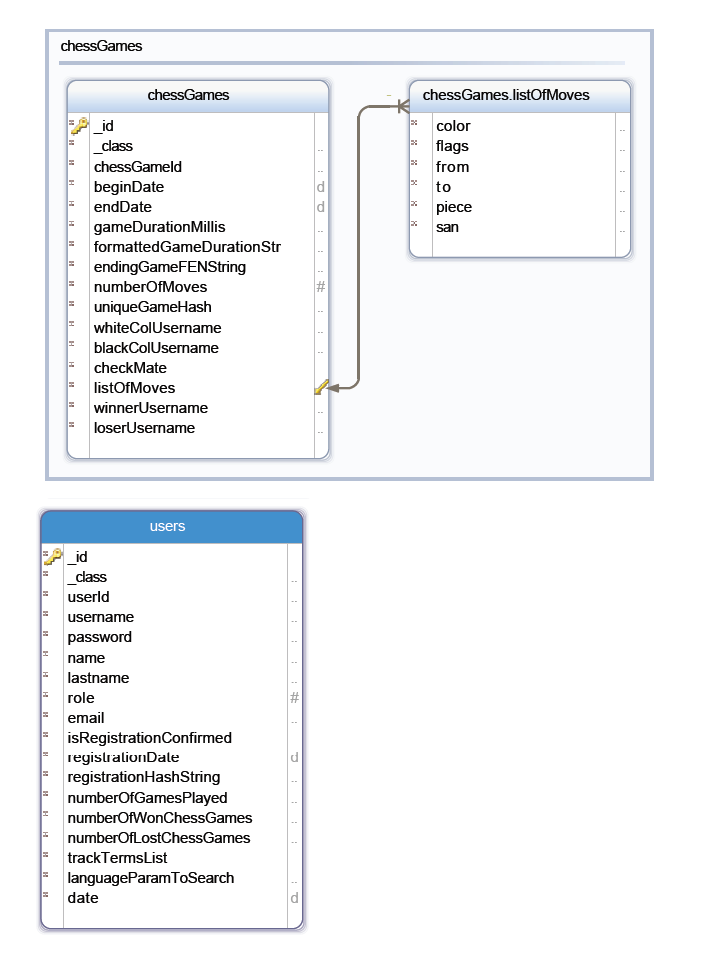
Rysunek - Diagram klas



### 4.2.4 Model bazy danych

Aplikacja wykorzystuje bazę danych MonogDB, która jest bazą typu NoSQL. MongoDB jest określana bazą *document-oriented*, przez co podstawową jednostką przechowującą dane w bazie nie określane mianem tabeli a kolekcji lub dokumentu. Nazwa bazy danych użytej w systemie to **chessapp\_db**. Na poniższym rysunku przedstawiono model bazy danych, na którym widać dwie kolekcje chessGames (z zagnieżdżonym obiektem move) i users.

Rysunek - - Model bazy danych

****

Konfiguracja połączenia z bazą danych znajduje się w pliku **mongo-config.xml** pod ścieżką **/src/main/webapp/ WEB-INF**

MongoDb ja wspomniano powyżej nie jest bazą relacyjną, nie występuje tu koncepcja kluczy obcych, natomiast jest możliwość przechowywania zagnieżdżonych obiektów oraz list typów prostych, stringów jak również list obiektów. W MongoDb zamiast pojęcia tabelka stosuje się pojęcie kolekcja bądź dokument. Zagnieżdząnie obiektów zostało zimplementowane w kolekcji chessGames, która przechowuje listę obiektów Move. Obiekt Move reprezentuje ruch szachowy.

Między obiektami ChessGame i Move występuje relacja has-a w postaci agregacji. Klasa **ChessGames** zawiera listę obiektów Move, która może mieć zero lub wiele elementów.

### 4.2.5. Operacje na danych

W niniejszym systemie zastosowano koncepcję **ORM** (object relational mapping) za pomocą frameworku Spring MVC i jego komponentu Spring Data MongoDB. Spring Data mapuje obiekty i kolekcje z bazy danych MongoDB do klas java. Dzięki temu zapytania do bazy danych można stosować za pomocą metod java a nie w postaci wysyłania stringów do bazy zawierających zapytania w języku zapytań (MognoDb ma swój język zapytań podobnie jak bazy SQL mają język zapytań SQL). Każda kolekcja w bazie ma swoją reprezentację w postaci klasy. Kolekcje z bazy chessapp\_db zostały zmapowane do klas java UserAccount, ChessApp, Move, na podstawie których mogą być tworzone instancje obiektów i te obiekty mogą być później zapisywane do bazy. Aby spring MVC identyfikował je jako mapowanie do bazy danych mongodb stosuje się anotacje: @Documen nad nazwą klasy oraz dodaje się argument do anotacji, który odnosi się do nazwy kolekcji w mongodb (collection = "users").

Kolekcja users zmapowana do klasy:

@Document**(**collection **=** "users"**)**

public class UserAccount **extends** ChessAppUser **implements**

Comparable**<**UserAccount**>** **{**

@Id

private String id**;**

private long userId**;**

private String password**;**

private String name**;**

private String lastname**;**

private int role**;**

private String email**;**

private String registrationHashString**;**

private Boolean isRegistrationConfirmed**;**

private Date registrationDate**;**

private Long numberOfGamesPlayed**;**

private Long numberOfWonChessGames**;**

private Long numberOfLostChessGames**;**

Do operacji typu CRUD (*create, read, update, delete*) na zasobach w bazie danych został również użyty komponent Spring Data MongoDB. Zastosowano wzorzec DAO : Data Access Object, czyli koncepcja stworzenia odseparowanej klasy skupiającej się na interakcji z bazą danych, w postaci działań typu CRUD. Projekt objął zdefiniowanie interfejsów: **UsersRepository** i **ChessGamesRepository** i ich implementacji **UsersRepositoryImpl** i **ChessRepositoryImpl.**

**Interfejs:**

public interface UsersRepository **{**

public String addUser**(**UserAccount user**);**

public List**<**UserAccount**>** getUsersList**();**

public UserAccount getUserByUsername**(**String username**);**

public UserAccount getUserByRegistrationHashString**(**

String registrationHashString**);**

public UserAccount getUserById**(**long userId**);**

public void deleteUser**(**UserAccount user**);**

public void updateUser**(**UserAccount user**);**

public List**<**UserAccount**>** getBestPlaying10Users**();**

**}**

**Implementacja interfejsu:**

@Repository

public class UsersRepositoryImpl **implements** UsersRepository **{**

@Autowired

private MongoTemplate mongoTemplate**;**

public static final String COLLECTION\_NAME **=** "users"**;**

private final Logger logger **=** Logger**.**getLogger**(**UsersRepositoryImpl**.**class**);**

public String addUser**(**UserAccount user**)** **{**

logger**.**debug**(**"addUser()"**);**

**if** **(!**mongoTemplate**.**collectionExists**(**UserAccount**.**class**))** **{**

mongoTemplate**.**createCollection**(**UserAccount**.**class**);**

**}**

UserAccount testUser **=** getUserByUsername**(**user**.**getUsername**());**

**if** **(**testUser **==** **null)** **{**

autoIncrementUserId**(**user**);**

mongoTemplate**.**insert**(**user**,** COLLECTION\_NAME**);**

logger**.**info**(**"user added"**);**

**return** "ok"**;**

**}** **else** **{**

logger**.**info**(**"login allready exist"**);**

**return** "fail"**;**

**}**

**}**

private void autoIncrementUserId**(**UserAccount usertoUpdate**)** **{**

logger**.**debug**(**"incrementUserId()"**);**

Query query **=** **new** Query**();**

UserAccount user **=** mongoTemplate**.**findOne**(**query**,** UserAccount**.**class**,**

COLLECTION\_NAME**);**

**if** **(**user **==** **null)** **{**

usertoUpdate**.**setUserId**(**0**);**

**}** **else** **{**

List**<**UserAccount**>** users **=** mongoTemplate**.**findAll**(**UserAccount**.**class**,**

COLLECTION\_NAME**);**

UserAccount lastUserBefore **=** users**.**get**(**users**.**size**()** **-** 1**);**

long lastUserBeforeId **=** lastUserBefore**.**getUserId**();**

logger**.**info**(**"lastUser id before sorting"**);**

logger**.**info**(**lastUserBeforeId **+** ""**);**

Collections**.**sort**(**users**);**

UserAccount lastUser **=** users**.**get**(**users**.**size**()** **-** 1**);**

long lastUserId **=** lastUser**.**getUserId**();**

logger**.**info**(**"lastUser id after sorting"**);**

logger**.**info**(**lastUserId **+** ""**);**

usertoUpdate**.**setUserId**(**lastUserId **+** 1**);**

**}**

**}**

public List**<**UserAccount**>** getUsersList**()** **{**

logger**.**debug**(**"getUsersList()"**);**

Query query **=** **new** Query**();**

query**.**with**(new** Sort**(**Sort**.**Direction**.**DESC**,** "userId"**));**

**return** mongoTemplate**.**find**(**query**,** UserAccount**.**class**,** COLLECTION\_NAME**);**

**}**

public List**<**UserAccount**>** getBestPlaying10Users**()** **{**

logger**.**debug**(**"getBestPlaying10Users()"**);**

Query query **=** **new** Query**();**

query**.**addCriteria**(**Criteria**.**where**(**"numberOfWonChessGames"**).**ne**(null)**

**.**gt**(**0**));**

query**.**with**(new** Sort**(**Sort**.**Direction**.**DESC**,** "numberOfWonChessGames"**));**

query**.**limit**(**10**);**

**return** mongoTemplate**.**find**(**query**,** UserAccount**.**class**,** COLLECTION\_NAME**);**

**}**

public UserAccount getUserByUsername**(**String username**)** **{**

logger**.**debug**(**"getUserByUsername()"**);**

Query query **=** **new** Query**();**

query**.**addCriteria**(**Criteria**.**where**(**"username"**).**is**(**username**));**

UserAccount user **=** mongoTemplate**.**findOne**(**query**,** UserAccount**.**class**,**

COLLECTION\_NAME**);**

**return** user**;**

**}**

public UserAccount getUserByRegistrationHashString**(**

String registrationHashString**)** **{**

logger**.**debug**(**"getUserByRegistrationHashString()"**);**

Query query **=** **new** Query**();**

query**.**addCriteria**(**Criteria**.**where**(**"registrationHashString"**).**is**(**

registrationHashString**));**

UserAccount user **=** mongoTemplate**.**findOne**(**query**,** UserAccount**.**class**,**

COLLECTION\_NAME**);**

**return** user**;**

**}**

public UserAccount getUserById**(**long userId**)** **{**

logger**.**debug**(**"getUserById()"**);**

Query query **=** **new** Query**();**

query**.**addCriteria**(**Criteria**.**where**(**"userId"**).**is**(**userId**));**

UserAccount user **=** mongoTemplate**.**findOne**(**query**,** UserAccount**.**class**,**

COLLECTION\_NAME**);**

**return** user**;**

**}**

public void deleteUser**(**UserAccount user**)** **{**

logger**.**debug**(**"deleteUser()"**);**

mongoTemplate**.**remove**(**user**,** COLLECTION\_NAME**);**

**}**

// UPDATE

public void updateUser**(**UserAccount user**)** **{**

logger**.**debug**(**"updateUser()"**);**

Update updateUserData **=** **new** Update**();**

updateUserData**.**set**(**"name"**,** user**.**getName**());**

updateUserData**.**set**(**"lastname"**,** user**.**getLastname**());**

updateUserData**.**set**(**"role"**,** user**.**getRole**());**

updateUserData**.**set**(**"email"**,** user**.**getEmail**());**

updateUserData**.**set**(**"isRegistrationConfirmed"**,**

user**.**getIsRegistrationConfirmed**());**

updateUserData**.**set**(**"registrationDate"**,** user**.**getRegistrationDate**());**

updateUserData**.**set**(**"registrationHashString"**,**

user**.**getRegistrationHashString**());**

**if** **(**user**.**getNumberOfGamesPlayed**()** **!=** **null)** **{**

updateUserData**.**set**(**"numberOfGamesPlayed"**,**

user**.**getNumberOfGamesPlayed**());**

**}**

**if** **(**user**.**getNumberOfWonChessGames**()** **!=** **null)** **{**

updateUserData**.**set**(**"numberOfWonChessGames"**,**

user**.**getNumberOfWonChessGames**());**

**}**

**if** **(**user**.**getNumberOfLostChessGames**()** **!=** **null)** **{**

updateUserData**.**set**(**"numberOfLostChessGames"**,**

user**.**getNumberOfLostChessGames**());**

**}**

**if** **(**user**.**getPassword**()** **!=** **null**

**&&** **!**user**.**getPassword**().**equalsIgnoreCase**(**""**))** **{**

updateUserData**.**set**(**"password"**,** user**.**getPassword**());**

**}**

Query query **=** **new** Query**();**

query**.**addCriteria**(**Criteria**.**where**(**"username"**).**is**(**user**.**getUsername**()));**

mongoTemplate**.**updateFirst**(**query**,** updateUserData**,** COLLECTION\_NAME**);**

**}**

**}**

# Realizacja systemu

## 5.1. Tworzenie kont użytkowników

Aplikacja omawiana w niniejszej pracy jest aplikacją webową. Może być dostępna dla każdego użytkownika podłączonego do sieci internetowej. Część zasobów jest dostępna dla użytkowników anonimowych, natomiast kluczowa funkcjonalność – gra z innymi użytkownikami online jest dostępna dla zalogowanego użytkownika. Konto użytkownika może utworzyć administrator w panelu administratora. Jak również konto użytkownika może utworzyć każdy użytkownik sieci internetowej za pomocą formularza do rejestracji. Jednak aby mieć dostęp do jej zasobów musi podać podczas rejestracji istniejący adres email, na który zostaje wysłany link aktywacyjny. Proces rejestracji od strony serwera realizuje klasa **SignInController**. Użytkownik w formularzu musi podać login, hasło, powtórzyć hasło, adres emial. Hasła są poddawane procesowi hashowania za pomocą klasy **PasswordEncryptor**. Login musi być unikalny w kontekście całego systemu, jeżeli użytkownik wprowadzi do formularza istniejący już w bazie login i kliknie *utwórz*, to w kontrolerze ta sytuacja zostanie przechwycona i zostanie wyświetlony komunikat walidacyjny w rodzaju: *taki login już istnieje proszę utworzyć inny*. Wysyłanie do użytkowników maila z linkiem aktywacyjnym realizuje klasa **MailService**. Dla każdego użytkownika link aktywacyjny jest unikatowy, jest on generowany za pomocą predefiniowanej klasy Java **UUID**. Użytkownik ma tydzień czasu na aktywacje linka. Jeżeli konto nie jest aktywowane to jego dane są usuwane z bazy. Tą funkcjonalność realizuje klasa **ScheduledTaskService**, która korzysta z anotacji @Scheduled dostępnej w frameworku Spring MVC, umożliwiającej wykonywanie zadań w określonych odstępach czasowych.

Użytkownicy są przechowywani w bazie danych w kolekcji users. Reprezentacją obiektową konta zarejestrowanego użytkownika zapisanego do bazy jest klasa **UserAccount**.

## 5.2 Autentykacja i uwierzytelnianie

Logowanie do serwisu jest możliwe jeżeli użytkownik ma potwierdzone konto. Bezpieczeństwo logowania zapewnia komponent Spring MVC - Spring security. Implementacja bezpieczeństwa przy użyciu tego komponentu polega na zdefiniowaniu ról użytkowników

i zdefiniowaniu jakie ścieżki url są dostępne dla określonych ról użytkownika. Za pomocą tagów xml można zdefiniować które zasoby mają być wyświetlane dla użytkownika o danej roli a które nie.

Przykłady kodu:

### 5.2.1. Dostęp do zasobów całej podstrony z pozycji url

Przykład z pliku **security-config.xml:**

<http auto-config=**"true"** disable-url-rewriting=**"false"**>

<headers disabled=**"true"** />

<csrf disabled=**"true"** />

<intercept-url pattern=**"/admin/\*\*"** access=**"hasRole('ROLE\_ADMIN')"** />

<intercept-url pattern=**"/user/\*\*"** access=**"hasRole('ROLE\_USER')"** />

<form-login login-page=**"/"** authentication-failure-url=**"/fail/"**

default-target-url=**"/"** />

<logout logout-success-url=**"/"** />

<form-login login-page=**"/login"** />

<logout logout-url=**"/j\_spring\_security\_logout"** />

<session-management>

<concurrency-control max-sessions=**"1"**

session-registry-alias=**"sessionRegistry"** />

</session-management>

</http>

<beans:bean id=**"userModel"** class=**"com.chessApp.model.UserAccount"**></beans:bean>

<!-- import resources/ beans from servlet config file -->

<beans:import resource=**"./mvc-config.xml"** />

<beans:import resource=**"./application-context.xml"** />

<beans:bean id=**"authenticationProvider"**

class=**"com.chessApp.security.LocalAuthenticationProvider"**></beans:bean>

<beans:bean id=**"usersRepositoryImpl"** class=**"com.chessApp.daoimpl.UsersRepositoryImpl"**></beans:bean>

<beans:bean id=**"passwordEncoder"**

class=**"org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder"** />

<authentication-manager alias=**"authenticationManager"**>

<authentication-provider ref=**"authenticationProvider"** />

</authentication-manager>

**<interceptor>** to tag definiujący jaki rodzaj użytkownika może mieć dostęp do danego adresu url.

Atrybut *pattern* reprezentuje url, atrybut *access* przyjmuje funkcje *hasRole()*, która zwraca wartość logiczną czy osoba odwiedzająca dany url ma uprawnienia do zasobów. Tag **<form-login>** przyjmuje atrybut *login-page*, który definiuje link akcji logowania. Link ten musi być przywołany w formularzu logowania w atrybucie *action* w tagu **<form>** . Tag **<authentication-manager>** odwołuje się do menadżera autentykacji predefiniowanego w Spring security, który oddelegowuje akcje logowania do tzw. Providera, który wymaga implementacji ze strony programisty, w tym przypadku klasy **LocalAuthenticationProvider**, która jest zaimplementowana w projekcie i zajmuje się procesem autentykacji.

Klasa LocalAuthenticationProvider:

@Component

public class LocalAuthenticationProvider **extends**

AbstractUserDetailsAuthenticationProvider **{**

private final Logger logger **=** Logger

**.**getLogger**(**LocalAuthenticationProvider**.**class**);**

@Autowired

private UsersRepository usersRepository**;**

@Autowired

private PasswordEncoder encoder**;**

private PasswordEncryptor passwordEncrypter **=** **new** PasswordEncryptor**();**

@Override

protected void additionalAuthenticationChecks**(**UserDetails userDetails**,**

UsernamePasswordAuthenticationToken authentication**)**

**throws** AuthenticationException **{**

**}**

@Override

public UserDetails retrieveUser**(**String username**,**

UsernamePasswordAuthenticationToken authentication**)**

**throws** AuthenticationException **{**

String password **=** **(**String**)** authentication**.**getCredentials**();**

String hashPassword **=** **null;**

**try** **{**

hashPassword **=** passwordEncrypter**.**encryptUserPassword**(**password**);**

**}** **catch** **(**Exception e**)** **{**

e**.**printStackTrace**();**

**}**

**if** **(!**StringUtils**.**hasText**(**password**))** **{**

logger**.**warn**(**username **+** ": no password provided"**);**

**throw** **new** BadCredentialsException**(**"Please enter password"**);**

**}**

**if** **(!**StringUtils**.**hasText**(**username**))** **{**

logger**.**warn**(**username **+** ": no login provided"**);**

**throw** **new** BadCredentialsException**(**"Please enter login"**);**

**}**

UserAccount user **=** usersRepository**.**getUserByUsername**(**username**);**

**if** **(**user **==** **null)** **{**

logger**.**warn**(**username **+** ": user not found"**);**

**throw** **new** UsernameNotFoundException**(**"Invalid Login"**);**

**}**

**if** **(**user**.**getIsRegistrationConfirmed**()** **==** **false)** **{**

logger**.**warn**(**username **+** ": not confirmed"**);**

**throw** **new** UserNotConfirmedException**(**username **+** ": not confirmed"**);**

**}**

**if** **(**username**.**equals**(**user**.**getUsername**())**

**&&** hashPassword**.**equals**(**user**.**getPassword**())**

**&&** hashPassword **!=** **null)** **{**

final List**<**GrantedAuthority**>** auths **=** UserRoles**.**getUserRoles**(**user

**.**getRole**());**

**return** **new** User**(**user**.**getUsername**(),** user**.**getPassword**(),** **true,** // enabled

**true,** // account not expired

**true,** // credentials not expired

**true,** // account not locked

auths**);**

**}** **else** **{**

**throw** **new** UsernameNotFoundException**(**"Invalid Login and password"**);**

**}**

**}**

**}**

### 5.2.2. Dostęp do zasobów na stronie internetowej

Przykład z pliku **menu.jsp:**

<%@ page import="com.chessApp.props.ChessAppProperties"%>

<%

String contextURL = ChessAppProperties

.getProperty("app.contextpath");

%>

<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>

<%@ taglib uri="http://www.springframework.org/security/tags"

prefix="security"%>

<ul id=**"navButtons"** class=**"nav nav-tabs navView"**>

<li><a href=**"<%=**contextURL**%>/"**>**home**</a></li>

<li><a href=**"<%=**contextURL**%>/play-chess-with-computer"**>**play-with-computer**</a></li>

<li><a href=**"<%=**contextURL**%>/play-chess-with-user"**>**play-with-others**</a></li>

<security:authorize access="hasRole('ROLE\_ADMIN')">

<li><a href=**"<%=**contextURL**%>/admin/users"**>**all-users**</a></li>

</security:authorize>

<security:authorize access="hasRole('ROLE\_ADMIN')" var="isAdmin" />

<c:choose>

<c:when test="${isAdmin}">

<li><a href=**"<%=**contextURL**%>/admin/your-account"**>**your-profile**</a></li>

</c:when>

<c:otherwise>

<security:authorize access="hasRole('ROLE\_USER')">

<li><a href=**"<%=**contextURL**%>/user/your-account"**>**your-profile**</a></li>

</security:authorize>

</c:otherwise>

</c:choose>

<security:authorize access="hasRole('ROLE\_USER')">

<li><a href=**"<%=**contextURL**%>/user/your-chessgames"**>**your-games-history**</a></li>

</security:authorize>

<li><a href=**"<%=**contextURL**%>/home/best-players"**>**best-players**</a></li>

<li class=**"pull-right logOutBtn text-capitalize"**><a

href=**"<%=**contextURL**%>/logout"**>**log out**</a></li>

<li class=**"pull-right logInBtn text-capitalize"**><a

href=**"<%=**contextURL**%>/login"**>**log in**</a></li>

</ul>

<div class=**"welcome-user-line"**>

<security:authorize access="hasAnyRole('ROLE\_ADMIN, ROLE\_USER')">

<h4 class=**"text-right logged-user-name"**>

**Welcome:** <span class=**"text-primary"**><b>**${currentUserName}**</b></span>

</h4>

</security:authorize>

</div>

Tag **<security:authorize>** określa jakie zasoby w pliku jsp mogą być zaprezentowane dla określonego użytkownika. Plik jsp na serwerze wykonuje się jako kod Java, który buduje strukturę pliku HTML według określonych instrukcji zdefiniowanych w pliku jsp za pomocą tagów xml. W powyższym przykładzie widzimy Tag xml **<security:authorize>** zdefiniowany przez bibliotekę spring security. Na początku zacytowanego kodu widać instrukcje <%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>

<%@ taglib uri=<http://www.springframework.org/security/tags> prefix="security"%>

które importują określone przestrzenie nazw dla tagów xml. Tag **<security:authorize>** przyjmuje atrybut *acces*, który przyjmuje funkcje *hasRole()* zwracającą wartość logiczną w zależności od tego jakie uprawnienia ma użytkownik odwiedzający stronę.

## 5.3 Implementacja gry w szachy

Interfejs do gry w szachy oraz logika do gry jest zaimplementowana po stronie klienta w języku javascript. Zostało to zrealizowane przy użyciu bibliotek javascript:

**chessboard.js** – biblioteka implementująca interfejs graficzny do gry w szachy oraz **chess.js** – biblioteka implementująca silnik do gry w szachy. Pliki źródłowe z tymi bibliotekami są obecne w katalogu: **/src/main/webapp/assets/js/lib/**

W katalogu **/src/main/webapp/assets/js/** umieszczone są pliki źródłowe z implementacją działania tych bibliotek w kontekście niniejszego systemu. Są to pliki:

* **chessComputerVsComputer.js –** implementacja funkcjonalności komputer versus komputer, widocznej na stronie domowej *home* jako warstwa prezentacyjna aplikacji.
* **chessUserVsComputer.js –** implementacja funkcjonalności użytkownik versus komputer, widocznej na podstronie *play-with-computer* jako warstwa demonstracyjna aplikacji.
* **chessUserVsUser.js –** implementacja funkcjonalności użytkownik versus komputer, widocznej na podstronie *play-with-users* jako warstwa prezentująca główną funkcjonalność aplikacji.

Kod z pliku **chessUserVsUser.js:**

***var*** startFENPosition **=** "rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1"**;**

// start new game

$**(**'#startPosBtn'**).**click**(*function*()** **{**

startNewGame**();**

**});**

***function*** startNewGame**()** **{**

board**.**position**(**startFENPosition**);**

game **=** ***new*** Chess**(**startFENPosition**);**

updateStatus**();**

**}**

// chess script ---------------------------------

***var*** board**,** game **=** ***new*** Chess**(),** statusEl **=** $**(**'#status'**),** fenEl **=** $**(**'#fen'**),** pgnEl **=** $**(**'#pgn'**);**

***var*** onDragStart **=** ***function*(**source**,** piece**,** position**,** orientation**)** **{**

console**.**log**(**"onDragStart()"**);**

console**.**log**(**'SENDED\_CHESS\_MOVE\_STATUS: ' **+** SENDED\_CHESS\_MOVE\_STATUS**);**

// if move does not belong to you

// pieces are blocked

// inviting user at start have "white piece"

// at start status is "White to move"

***if*** **(**statusEl**.**text**().**trim**()** **!=** SENDED\_CHESS\_MOVE\_STATUS**)** **{**

***return*** ***false*;**

**}**

// allow only one move

***if*** **(**CHESS\_MOVE\_COUNTER **>** 0**)** **{**

***return*** ***false*;**

**}**

// do not pick up pieces if the game is over

// only pick up pieces for the side to move

***if*** **(**game**.**game\_over**()** **===** ***true***

**||** **(**game**.**turn**()** **===** 'w' **&&** piece**.**search**(**/^b/**)** **!==** **-**1**)**

**||** **(**game**.**turn**()** **===** 'b' **&&** piece**.**search**(**/^w/**)** **!==** **-**1**))** **{**

***return*** ***false*;**

**}**

**};**

***var*** onDrop **=** ***function*(**source**,** target**)** **{**

console**.**log**(**"onDrop()"**);**

// see if the move is legal

***var*** move **=** game**.**move**({**

***from*** **:** source**,**

to **:** target**,**

promotion **:** 'q'

// simplicity

**});**

// illegal move

***if*** **(**move **===** null**)**

***return*** 'snapback'**;**

CHESS\_MOVE\_COUNTER**++;**

CURRENT\_CHESS\_MOVE **=** move**;**

updateStatus**();**

**};**

***var*** onSnapEnd **=** ***function*()** **{**

board**.**position**(**game**.**fen**());**

**};**

***var*** updateStatus **=** ***function*()** **{**

***var*** status **=** ''**;**

***var*** moveColor **=** 'White'**;**

***if*** **(**game**.**turn**()** **===** 'b'**)** **{**

moveColor **=** 'Black'**;**

**}**

// checkmate?

***if*** **(**game**.**in\_checkmate**()** **===** ***true*)** **{**

status **=** 'Game over, ' **+** moveColor **+** ' is in checkmate.'**;**

***var*** winnerColor **=** ""**;**

***var*** looserColor **=** ""**;**

***if*** **(**moveColor **==** 'White'**)** **{**

winnerColor **=** 'black'**;**

winnerUsername **=** BLACK\_COLOR\_USERNAME**;**

loserUsername **=** WHITE\_COLOR\_USERNAME**;**

**}** ***else*** **{**

winnerColor **=** 'white'**;**

winnerUsername **=** WHITE\_COLOR\_USERNAME**;**

loserUsername **=** BLACK\_COLOR\_USERNAME**;**

**}**

// only winner send message, to prevent duplicates

// browser client that create game.in\_checkmate() is winner

// by default

***if*** **(**WEBSOCKET\_CLIENT\_NAME **==** winnerUsername**)** **{**

***var*** fenString **=** fenFromYourMove**.**value**;**

webSocket**.**send**(**JSON**.**stringify**({**

type **:** "game-over"**,**

fen **:** fenString**,**

winnerColor **:** winnerColor**,**

winnerUsername **:** winnerUsername**,**

checkMate **:** ***true*,**

loserUsername **:** loserUsername**,**

sendFrom **:** WEBSOCKET\_CLIENT\_NAME**,**

sendTo **:** $**(**'#quit-game-btn'**).**data**(**"gamePartner"**)**

**}));**

**}**

$**(**'#move-for'**).**html**(**"<h1 class=\"text-success\">YOU WIN !</h1>"**);**

alert**(**"check mate, you Win with " **+** OPPONENT\_USERNAME **+** " !"**);**

$**(**'#startPosBtn'**).**show**();**

$**(**'#game-status'**).**data**(**'isPlaying'**,** ***false*);**

$**(**'#game-status'**).**html**(**''**);**

$**(**'#send-move-btn'**).**data**(**"opponentName"**,** ''**);**

$**(**'#quit-game-btn'**).**data**(**"gamePartner"**,** ''**);**

$**(**'#play-with-opponent-interface-actions'**).**attr**(**"hidden"**,** ***true*);**

OPPONENT\_USERNAME **=** ""**;**

clearParticipantsListView**();**

**}**

// draw?

else if (game.in\_draw() === true) {

status = 'Game over, drawn position';

}

// game still on

else {

status = moveColor + ' to move';

// check?

if (game.in\_check() === true) {

status += ', ' + moveColor + ' is in check';

}

}

statusEl.html(status);

fenEl.html(game.fen());

$('#fenFromYourMove').val(game.fen());

pgnEl.html(game.pgn());

};

var cfg = {

pieceTheme : '/resources/images/chesspieces/wikipedia/{piece}.png',

draggable : true,

position : 'start',

onDragStart : onDragStart,

onDrop : onDrop,

onSnapEnd : onSnapEnd

};

board = ChessBoard('board', cfg);

updateStatus();

W powyższym przykładzie widać zdarzeniową obsługę gry w szachy przy użyciu zdarzeń drag and drop. Podczas zdarzenia *onDragStart* jest sprawdzane czy możemy wykonać ruch, czy ruch należy do obecnego użytkownika, czy do przeciwnika. Przy zdarzeniu *onDrop*, jest tworzony obiekt *move* i aktualizowany status gry poprzez funkcję *updateStatus()*. Zdarzenie *onSnapEnd* buduje planszę na nowo po wykonanym ruchu za pomocą funkcji *position()* wykonywanej na obiekcie game przyjmującej stringa fen jao argument: *board.position(game.fen());*

Fen to Notacja Forsytha-Edwardsa. Funkcja *UpdateStatus()* wywołana w zdarzeniu *onDrop* aktualizuje obiekty przechowujące aktualny status gry w postaci ciągu znaków fen i jeżeli to szach mat to wysyłana jest o tym wiadomość na serwer w postaci instancji klasy GameMessage za pomocą Websocket:

var fenString **=** fenFromYourMove**.**value**;**

webSocket**.**send**(**JSON**.**stringify**({**

type **:** "game-over"**,**

fen **:** fenString**,**

winnerColor **:** winnerColor**,**

winnerUsername **:** winnerUsername**,**

checkMate **:** ***true*,**

loserUsername **:** loserUsername**,**

sendFrom **:** WEBSOCKET\_CLIENT\_NAME**,**

sendTo **:** $**(**'#quit-game-btn'**).**data**(**"gamePartner"**)**

**}));**

Na podstawie danych z tej wiadomości i danych przechowywanych na serwerze dokonuje się zapis gry do bazy danych;

## 5.4 Komunikator do gry w szachy

### 5.4.1. Odbiór wiadomości po stronie serwera i po stronie klienta

Funkcjonalność gry online była głównym tematem zainteresowań niniejszego projektu i jednym z najtrudniejszych zadań w procesie implementacji oprogramowania. Możliwość gry online wymagała zaprogramowania komunikatora działającego w czasie rzeczywistym, bez konieczności przeładowywania, czy odświeżania strony po zmianie statusu gry, czy wysłaniu aktualnej listy dostępnych użytkowników. Do implementacji systemu została użyta technologia Websocket. Klasy implementujące komunikator websocket znajdują się w pakietach: **com.chessApp.websocket** i **com.chessApp.gameProtocol**.Implementacja komunikatora opiera się na obsłudze zdarzeń po stronie serwera i po stronie klienta, oraz przechowywaniu wybranych danych w pamięci operacyjnej serwera. Serwer pełni rolę punktu końcowego/ przechwytującego wysłaną wiadomość, określanego mianem ServerEndpoint (**zaprogramowanego w Java**). Po stronie klienta mamy odpowiednik w postaci ClientEndpoint (**zaprogramowanego w JavaScript**).

Server endpoint został zaimplementowany w pliku **WebSocketServer.java** w pakiecie **com.chessApp.websocket.** Client ednpoint w pliku **websocketClientEndpoint.js** obecnym

w katalogu pod ścieżką **/src/main/webapp/assets/js/**

Fragmenty kodu **WebSocketServer.java:**

@Service

@ServerEndpoint**(**"/chessapp-live-game/{sender}"**)**

public class WebSocketServer **{**

private final static Logger log **=** Logger**.**getLogger**(**WebSocketServer**.**class**);**

private final WebSocketSessionHandler sessionHandler **=** **new** WebSocketSessionHandler**();**

private final GameUsersHandler usesrHandler **=** **new** GameUsersHandler**();**

private final ChessGamesHandler chessGamesHandler = new ChessGamesHandler();

private GameMessageProtocol gameMessageProtocol = new GameMessageProtocol(

sessionHandler, usesrHandler, chessGamesHandler);

private Gson gson = new Gson();

@OnMessage

public void onMessage(String msg, Session wsSession,

@PathParam("sender") String sender) throws IOException {

log.info("wiadomość odebrana przez server: ");

GameMessage message = gson.fromJson(msg, GameMessage.class);

gameMessageProtocol.proccessMessage(message, msg);

}

@OnOpen

public void onOpen(Session wsSession, EndpointConfig config,

@PathParam("sender") String sender) {

log.info("connection started, websocket session id: "

+ wsSession.getId() + " " + sender + " open connection ");

if (usesrHandler.userListNotContainsUsername(sender)) {

GameUser gameUser = new GameUser(sender);

gameUser.setCommunicationStatus(GameUserCommunicationStatus.WAIT\_FOR\_NEW\_GAME);

synchronized (this) {

wsSession.getUserProperties().put("sessionOwner",

gameUser.getUsername());

sessionHandler.addSession(gameUser.getUsername(), wsSession);

usesrHandler.addWebsocketUser(gameUser);

}

sessionHandler.sendToAllConnectedSessions(gameUser.getUsername());

}

}

@OnClose

public void onClose(CloseReason closeReason, Session wsSession,

@PathParam("sender") String sender) {

log.info("connection closed. Reason: " + closeReason.getReasonPhrase());

log.info(sender);

synchronized (this) {

GameUser cloesingConnectionUser = usesrHandler

.getWebsocketUser(sender);

if (cloesingConnectionUser.getPlayNowWithUser() != null

&& cloesingConnectionUser.getPlayNowWithUser() != "") {

GameUser cloesingConnectionUserGamePartner = usesrHandler

.getWebsocketUser(cloesingConnectionUser

.getPlayNowWithUser());

cloesingConnectionUserGamePartner.setPlayNowWithUser(null);

cloesingConnectionUserGamePartner

.setCommunicationStatus(GameUserCommunicationStatus.WAIT\_FOR\_NEW\_GAME);

GameMessage disconnectMsg = new GameMessage();

disconnectMsg.setType(GameMessageType.USER\_DISCONNECT);

sessionHandler.sendToSession(

cloesingConnectionUserGamePartner.getUsername(),

sender, gson.toJson(disconnectMsg));

}

usesrHandler.removeWebsocketUser(sender);

sessionHandler.removeSession(sender);

}

sessionHandler.sendToAllConnectedSessionsActualParticipantList();

}

@OnError

public void onErrorReceived(Throwable t) {

log.debug("there was an error with connection");

log.debug(t);

}

}

WebsocketServer jest klasą obsługującą zdarzenia: **onOpen**, **onMessage**, **onClose** i **onError**. Na podstawie tych zdarzeń wywoływanych przez klienta aplikacji wywoływane są określone działania. Serwer w swojej pamięci operacyjnej przechowuje listę użytkowników podłączonych do repozytorium gry online. Reprezentacją połączenia przez webSocket jest sesja, każdy użytkownik ma swoją sesję, jest ona innym obiektem niż sesja logowania, która przechowuje kontekst użytkownika dla bezstanowego protokołu http, te sesje są w kontekście połączenia poprzez websocket. Aby poszczególnych użytkowników dało się identyfikować w implementacji posłużono się kolekcjami dostępnymi w języku Java. Podstawową strukturą użytą do tego celu jest mapa. Zostały utworzone mapy:

* **Map<String, Session>**  **webSocketSessions –** mapa, w której kluczem jest nazwa użytkownika, wartością sesja polaczenia, do której można wysłać wiadomość przy zdarzeniu onMessage
* **Map<String, GameUser> gameUsers -** w tej kolekcji są przechowywaneszczegóły na temat użytkowników**.** Mapa gameUsers przechowuje użytkowników gry w postaci obiektów, przechowujących informacje potrzebne do tego aby prowadzić konwersację z innymi użytkownikami. Kluczami do wartości tej kolekcji są także nazwy użytkowników. Przy tym rozwiązaniu zarówno sesja danego użytkownika jak i jego reprezentacja w postaci obiektu maja ten sam klucz, dzięki czemu pozostają w relacji *one to one*. Konto użytkownika zarejestrowanego w aplikacji reprezentuje obiekt UserAccount. W kontekście gry ten sam użytkownik jest reprezentowany w obiekcie GameUser reprezentującym jego kontekst w repozytorium graczy online i prowadzonej grze szachowej (jeżeli jest prowadzona), który z kolei jest powiązany z obiektem Session reprezentującym jego połączenie z warstwą websocket. Wszystkie te obiekty są powiązane jednym kluczem, czyli nazwą użytkownika.
* **Map<String, ChessGame> chessGamesMap** – kolekcja, która przechowuje informacje na temat aktualnie toczących się gier. Kluczem jest tutaj pseudolosowo generowany unikatowy *hash*, wartością obiekt gry. Użytkownicy uczestniczący w pojedynku występujący w postaci instancji klas GameUser przechowują w zmiennej String uniqueActualGameHash, hash aktualnie toczonej przez siebie gry. Tworzy się w ten sposób unikatowa para powiązana kluczem uniqueActualGameHash, zamknięta w kontekście prowadzonej przez siebie gry. Ten sam unikatowy hash, jest kluczem do obiektu ich pojedynku przechowywanym w kolekcji chessGamesMap.

Akcje związane z obsługą tych kolekcji zostały przeniesione do osobnych klas: **GameUsersHandler**, **ChessGamesHandler**, **WebSocketSessionsHandler**. Zastosowanie tych kolekcji w postaci map oraz obiektów wykonujących na nich operacje dało możliwość wysyłania wiadomości do wszystkich użytkowników w formie *broadcastu*, co jest zastosowane w momencie gdy ktoś nowy podłącza się do repozytorium gry:

public synchronized void sendToAllConnectedSessionsActualParticipantList**()** **{**

String jsonUsersList **=** gson**.**toJson**(**GameUsersHandler**.**gameUsersMap

**.**values**());**

**for** **(**String username **:** sessionsMap**.**keySet**())** **{**

Session userSession **=** sessionsMap**.**get**(**username**);**

**try** **{**

userSession**.**getBasicRemote**().**sendText**(**jsonUsersList**);**

**}** **catch** **(**IOException e**)** **{**

logger**.**info**(**e**);**

**}**

**}**

**}**

Jak i możliwość wysyłania wiadomości w komunikacji *jeden do jeden*, kiedy jest prowadzona gra w parach, lub realizowane jest zaproszenie do gry.

public void sendToSession**(**String toUsernameName**,** String fromUsername**,**

String message**)** **{**

logger**.**info**(**"sendToSession()"**);**

Session userSession **=** sessionsMap**.**get**(**toUsernameName**);**

**if** **(**userSession **!=** **null)** **{**

**try** **{**

userSession**.**getBasicRemote**().**sendText**(**message**);**

**}** **catch** **(**IOException e**)** **{**

logger**.**debug**(**e**);**

**}**

**}**

**}**

Kliencki endpoint obsługuje te same zdarzenia co Serwer endpoint. Jest on obecny w katalogu

Kod z pliku **websocketClientEndpoint.js**:

***var*** TIMEOUT\_FOR\_HANDSHAKE **=** 15**;**

***var*** CLICK\_REFUSED\_FLAG **=** ***false*;**

***var*** CLICK\_AGREEMENT\_FLAG **=** ***false*;**

// ------CONNECT TO WEBSOCKET FUNCTION, WEBSOCKET EVENTS----------------------

***function*** connectToWebSocket**()** **{**

console**.**log**(**'connectToWebSocket()'**);**

// ----init websocket -------------------------------------

***var*** endpointUrl **=** "ws://" **+** document**.**location**.**host **+** "/chessapp-live-game/"

**+** WEBSOCKET\_CLIENT\_NAME**;**

webSocket **=** ***new*** WebSocket**(**endpointUrl**);**

// websocketClient events -----------------------------

webSocket**.**onopen **=** ***function*(**event**)** **{**

console**.**log**(**"Server connected \n"**);**

console**.**log**(**event**.**data**);**

$**(**'#disconnect'**).**attr**(**"disabled"**,** ***false*);**

webSocket**.**send**(**JSON**.**stringify**({**

type **:** "welcome-msg"**,**

sendFrom **:** WEBSOCKET\_CLIENT\_NAME

**}));**

$('#connection-status').html(

"<div class=\"alert alert-success connection-status-msg\">"

+ "<h2>You are connected!</h2></div>");

var disconnectBtn = $('#disconnect');

var connectBtn = $('#connectToWebSocket');

if (disconnectBtn.attr("disabled", true)) {

disconnectBtn.removeAttr("class");

disconnectBtn.attr("disabled", false);

disconnectBtn.attr("class", "btn btn-danger pull-right");

}

connectBtn.removeAttr("class");

connectBtn.attr("disabled", true);

connectBtn.attr("class", "btn btn-default");

};

// -----------------------------------

webSocket.onmessage = function(event) {

console.log("onmessage: ");

clientMsgProtocol.proccessMessage(event);

};

// -----------------------------------

webSocket.onclose = function(event) {

$('#connection-status').html(

"<div class=\"alert alert-warning connection-status-msg\">"

+ "<h2>You are disconnected!</h2></div>");

var disconnectBtn = $('#disconnect');

var connectBtn = $('#connectToWebSocket');

if (connectBtn.attr("disabled", true)) {

connectBtn.removeAttr("class");

connectBtn.removeAttr("disabled");

connectBtn.attr("disabled", false);

connectBtn.attr("class", "btn btn-primary");

}

disconnectBtn.removeAttr("class");

disconnectBtn.attr("disabled", true);

disconnectBtn.attr("class", "btn btn-default pull-right");

$('#game-status').html('');

$('#participants div ul').html('');

$('#disconnect').attr("disabled", true);

$('#play-with-opponent-interface').attr("hidden", true);

$('#startPosBtn').show();

OPPONENT\_USERNAME = "";

console.log(event);

};

// -----------------------------------

webSocket.onerror = function(event) {

webSocket.send("error: client disconnected");

console.log("Server disconnected \n");

console.log(event);

webSocket.close();

$('#startPosBtn').show();

$('#disconnect').attr("disabled", true);

$('#play-with-opponent-interface').attr("hidden", true);

OPPONENT\_USERNAME = "";

};

};

// close websocket when page reload --------------------------------------------

window.onbeforeunload = function() {

webSocket.onclose = function() {

webSocket.send("client disconnected");

console.log("Server disconnected \n");

}; // disable onclose handler first

webSocket.close();

$('#connectToWebSocket').attr("disabled", false);

OPPONENT\_USERNAME = "";

window.location.reload(false);

};

Jak widać powyżej są tu obsługiwane te same zdarzenia co na serwerze: **onOpen**, **onMessage**, **onClose** i **onError**.

Stan gry i ruchy poszczególnych użytkowników są przekazywane między nimi za pomocą notacji fen. **FEN** Jest to tak zwana **notacja Forsytha-Edwardsa.** Polega ona na zapisaniu położenia wszystkich figur oraz do kogo należy następny ruch w postaci ciągu znaków. Przykład notacji fen: „r1bqkbnr/pppp1ppp/2n5/1B2p3/4P3/5N2/PPPP1PPP/RNBQK2R”.

### 5.4.2. Protokół komunikacji

Jedną z większych trudności przy implementacji komunikatora do gry było zapewnienie parom użytkowników zamkniętej konwersacji, tak aby ich gra była bezpieczna od wtargnięcia i ingerencji innych użytkowników w grę. Zostało to osiągnięte poprzez wprowadzenie zaproszenia do gry, zanim zostanie wysłany jakikolwiek ruch szachowy. To spowodowało, że wiadomości stały się różnego typu. Innego typu jest wiadomość „zapraszam cię do gry” niż wiadomość w postaci ciągu znaków fen „r1bqkbnr/pppp1ppp/2n5/1B2p3/4P3/5N2/PPPP1PPP/RNBQK2R”. Wymagało to wprowadzenia typowania wiadomości oraz zaimplementowanie protokołu komunikacji przy zdarzeniu **onmessage**, który to w zależności od typu wiadomości delegowałby do innych zadań przy użyciu instrukcji warunkowych opartych na pytaniu „czy ta wiadomość jest wiadomością typu?”. Wiadomość wysyłaną przez użytkowników na serwer reprezentuje Obiekt **GameMessage**. Obiekt ten ma m.in. właściwośc **String type**, który przechowuje wspomniany powyżej typ. Oprócz tego obiekt wiadomość ma właściwości **String sendFrom** i **String sendTo**, które przechowują informacje o tym kto wysłał wiadomość i do kogo

Fragment kodu klasy GmaeMessage:

public class GameMessage **{**

private String type**;**

private String fen**;**

private String sendTo**;**

private String sendFrom**;**

private String moveStatus**;**

private GameUser sendToObj**;**

private GameUser sendFromObj**;**

private Boolean checkMate**;**

private ChessMove chessMove**;**

private String whiteColUsername**;**

private String blackColUsername**;**

public GameMessage**()** **{**

**}**

Typy wiadomości zostały zdefiniowane w klasie **GameMessageType** przechowującej pola statyczne w postaci stringów reprezentujące określone typy wiadomości.

public class GameMessageType **{**

public final static String USER\_CONNECT **=** "welcome-msg"**;**

public final static String GAME\_HANDSHAKE\_INVITATION **=** "game-handshake-invitation"**;**

public final static String GAME\_HANDSHAKE\_AGREEMENT **=** "game-handshake-agreement"**;**

public final static String GAME\_HANDSHAKE\_REFUSE **=** "game-handshake-refuse"**;**

public final static String CHESS\_MOVE **=** "chess-move"**;**

public final static String GAME\_OVER **=** "game-over"**;**

public final static String QUIT\_GAME **=** "quit-game"**;**

public final static String USER\_DISCONNECT **=** "goodbye-msg"**;**

public final static String TRY\_LATER **=** "try-later"**;**

**}**

Gdy dany użytkownik chce zagrać z drugim użytkownikiem, to musi wysłać mu zaproszenie GAME\_HANDSHAKE\_INVITATION, drugi może się zgodzić wysyłając: GAME\_HANDSHAKE\_AGREEMENT lub nie, wysyłając: GAME\_HANDSHAKE\_REFUSE. Jeżeli jest zgoda drugiego użytkownika to obiekty tych użytkowników przyjmują status IS-PLAYING. Typy statusów użytkowników są przechowywane w klasie **GameUserCommunicationStatus** również w postaci pól statycznych:

public class GameUserCommunicationStatus **{**

public final static String WAIT\_FOR\_NEW\_GAME **=** "wait-for-new-game"**;**

public final static String IS\_DURING\_HANDSHAKE **=** "during-handshake"**;**

public final static String IS\_PLAYING **=** "is-playing"**;**

**}**

Gdy użytkownik ma status WAIT\_FOR\_NEW\_GAME to każdy może się z nim skontaktować i on może z każdym się kontaktować. Gdy ma status IS\_DURING\_HANDSHAKE, może skontaktować sie tylko z użytkownikiem, którego zaprosił do gry bądź został przez niego zaproszony. Gdy ma status IS\_PLAYING, może wysyłać wiadomości tylko do użytkownika, z którym gra i może tylko odbierać wiadomości od użytkownika, z którym gra, co jest bardzo ważne i gwarantuje bezpieczeństwo gry.

**Pozostałe typy wiadomości:**

Ruch szachowy to wiadomość typu CHESS\_MOVE, koniec gry to GAME\_OVER lub QUIT\_GAME, wtedy automatycznie użytkownicy zmieniają status na WAIT\_FOR\_NEW\_GAME. Status USER\_CONNECT reprezentuje stan : gdy nowy użytkownik przyłącza sie do gry, wtedy do wszystkich użytkowników wysyłana jest aktualna lista użytkowników. Podobnie dzieje się przy USER\_DISCONNECT oznacza to iż użytkownik zakończył połączenie i do wszystkich wysyłana jest aktualna lista użytkowników. TRY\_LATER gdy użytkownik, z którym chcemy się skontaktować ma status IS\_DURING\_HANDSHAKE albo IS\_PLAYING, przy czym gdy użytkownik jest w jednym ze stanów IS\_DURING\_HANDSHAKE lub IS\_PLAYING to dla pozostałych użytkowników poza swoją para do gry są zablokowane akcje w interfejsie użytkownika umożliwiające kontakt z tym z daną jednostką.

Typy wiadomości są przechowywanie w instancjach klasy GameUser w właściwości **String communicationStatus** i aktualizowane w protokole komunikacji w zależności od sytuacji.

Fragment kodu klasy GameUser:

@Component

public class GameUser **extends** ChessAppUser **{**

private long userIdInGameContext**;**

private String playNowWithUser**;**

private long numberOfScores**;**

private String chessColor**;**

private String communicationStatus**;**

private String uniqueActualGameHash**;**

### 5.4.2.1 Opis przebiegu komunikacji pomiędzy graczami

Poniżej zostanie przybliżony proces komunikacji między graczami w systemie.

* Przyjmijmy że są obecni Uzytkownik1 i Uzytkownik2
* Użytkownik1 i Uzytkownik2 po zalogowaniu do serwisu przyłączają się do repozytorium gry online poprzez podstronę play-with-users i przycisk connect.
* Nowo połączony użytkownik ma status WAIT\_FOR\_NEW\_GAME.
* Uzytkownik1 zaprasza Użytkownika2,
* po tym zdarzeniu u obu użytkowników zmienia się status na IS\_DURING\_HANDSHAKE. Jezeli Użytkownika2 się zgadza rozpoczynaja gre i oboje zmieniają status na IS\_PLAYING. Dopóki, któryś z użytkowników nie zrezygnuje, lub nie nastąpi zdarzenie szach mat, toczy się gra, i obu użytkowników dalej ma status IS\_PLAYING.
* Jeżeli jeden z użytkowników wygrywa lub rezygnuje, gra się kończy i obu użytkowników zmienia status na WAIT\_FOR\_NEW\_GAME.
* Jeżeli Użytkownika2 odmówił po zaproszeniu oboje wracają do statusu początkowego WAIT\_FOR\_NEW\_GAME i zarówno proces gry jak i status IS\_PLAYING nie zaistniał.

### 5.4.2.2 Projekt komunikacji pomiędzy graczami

Projekt oraz implementacja powyższych założeń przebiegał poprzez zapis algorytmu w formie pseudokodu, następnie zilustrowaniu jego działania w formie schematu blokowego oraz implementacja w kodzie java.

#### 5.4.2.2.1. Pseudokod

Start**;**

GameUser user1 **=** **new** GameUser**(**WAIT\_FOR\_NEW\_GAME**);**

GameUser user2 **=** **new** GameUser**(**WAIT\_FOR\_NEW\_GAME**);**

user1**.**connect**();**

user2**.**connect**();**

user1**.**invite**(**user2**);**

user1**.**gameStatus **=** IS\_DURING\_HANDSHAKE**;**

user2**.**gameStatus **=** IS\_DURING\_HANDSHAKE**;**

**if** **(**user2**.**agreeToPlayWith**(**user1**))** **{**

user1**.**gameStatus **=** IS\_PLAYING**;**

user2**.**gameStatus **=** IS\_PLAYING**;**

**}** **else** **{**

user1**.**gameStatus **=** WAIT\_FOR\_NEW\_GAME**;**

user2**.**gameStatus **=** WAIT\_FOR\_NEW\_GAME**;**

**}**

**while(**user1**.**gameStatus **=** IS\_PLAYING **&&** user2**.**gameStatus **=** IS\_PLAYING**)** **{**

**if(**user1**.**gameOver**()** **||** user2**.**gameOVER**()** **||** checkMate**())** **{**

user1**.**gameStatus **=** WAIT\_FOR\_NEW\_GAME**;**

user2**.**gameStatus **=** WAIT\_FOR\_NEW\_GAME**;**

**}**

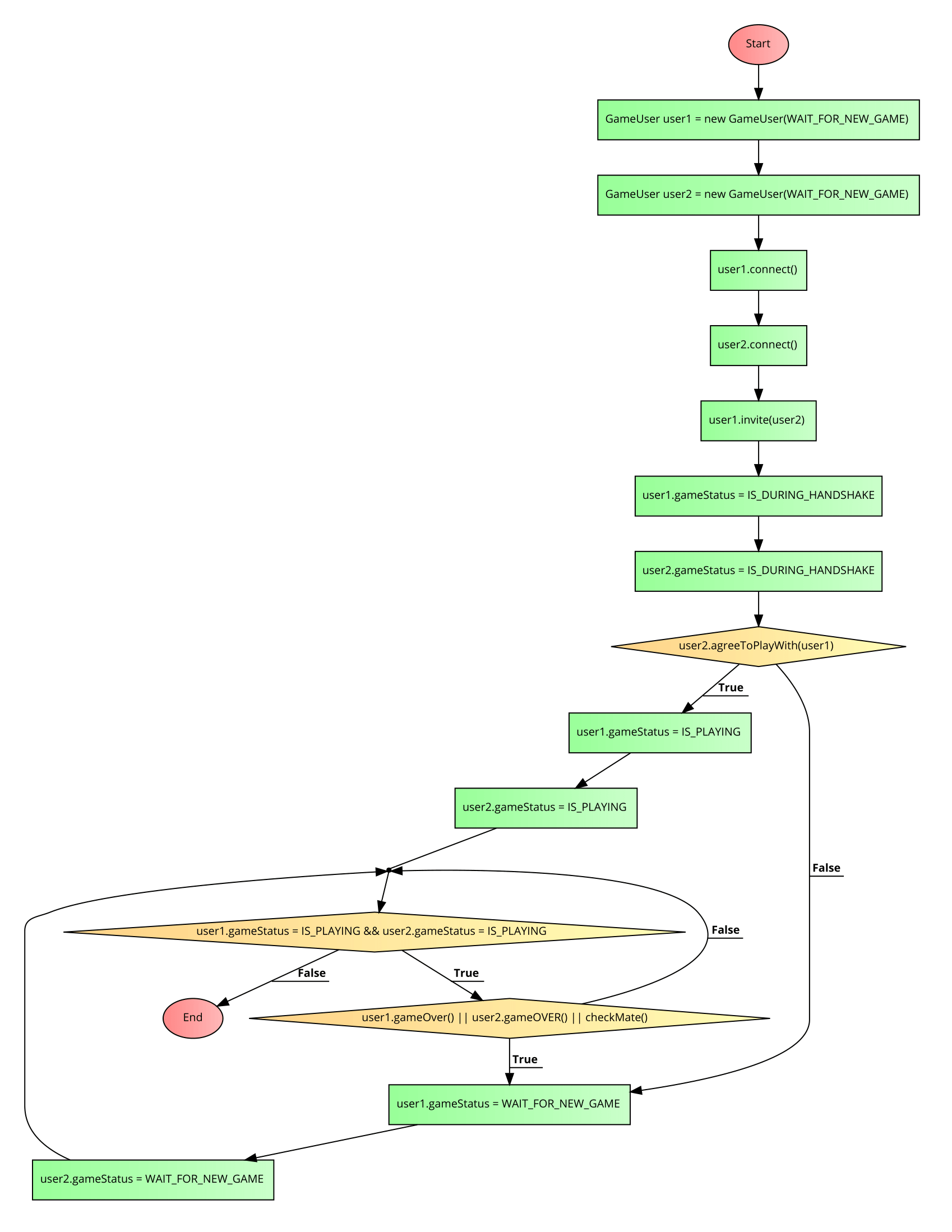
**}**

End**;**

##### 5.4.2.2.2 Schemat blokowy

Poniższy rysunek prezentuje schemat blokowy komunikacji pomiędzy użytkownikami korzystającego z komunikatora do gry w szachy w systemie.

Rysunek 5 - - Schemat blokowy komunikacji graczy



# Testy

System społecznościowy do gry w szachy został poddany kilku rodzajom testów, zarówno testom w procesie implementacji jak i testom końcowym. W przypadku złożonych systemów jakimi są aplikacje webowe korzystające najczęściej z kilkunastu technologii, zachodzi sytuacja iż nie każdą funkcjonalność można przetestować za pomocą narzędzie do testowania. Podobna sytuacja miała miejsce podczas testowania niniejszego systemu, na skutek czego przeważającym sposobem testowania programu, było testowanie aplikacji z punktu widzenia użytkownika w formie imitowania typowych dla niego zachowań. Natomiast funkcjonalności, które były możliwe do przetestowania za pomocą narzędzi zostały w ten sposób przetestowane.

W aplikacji zostały użyte testy jednostkowe za pomocą biblioteki **JUnit**. Kod testów jednostkowych jest obecny w pakiecie **com.chessApp.tests** Prezentując wybrany test jednostkowy, zostanie zaprezentowany fragment klasy

**CalculateTimeDurationBeetwenGameBeginAndEndMethodTest** testującej metodę zwracająca czas trwania gry w szachy pomiędzy użytkownikami.

Fragment kodu testu jednostkowego metody zwracającej czas trwania gry :

public class CalculateTimeDurationBeetwenGameBeginAndEndMethodTest **{**

private ChessGame game**;**

private DateTime dateTime**;**

@Before

public void prepare**()** **{**

game **=** **new** ChessGame**();**

**}**

@Test

public void durationIs10MinnutesTest**()** **{**

// given

long duration10MinutesExpected **=** 6000000**;**

Date beginDate **=** **new** Date**();**

dateTime **=** **new** DateTime**(**beginDate**);**

Date dateAfterTenMinnutes **=** dateTime**.**plus**(**duration10MinutesExpected**)**

**.**toDate**();**

game**.**setBeginDate**(**beginDate**);**

game**.**setEndDate**(**dateAfterTenMinnutes**);**

// when

ChessGamesHandler

**.**calculateAndSetTimeDurationBeetwenGameBeginAndEnd**(**game**);**

long durationResult **=** game**.**getGameDurationMillis**();**

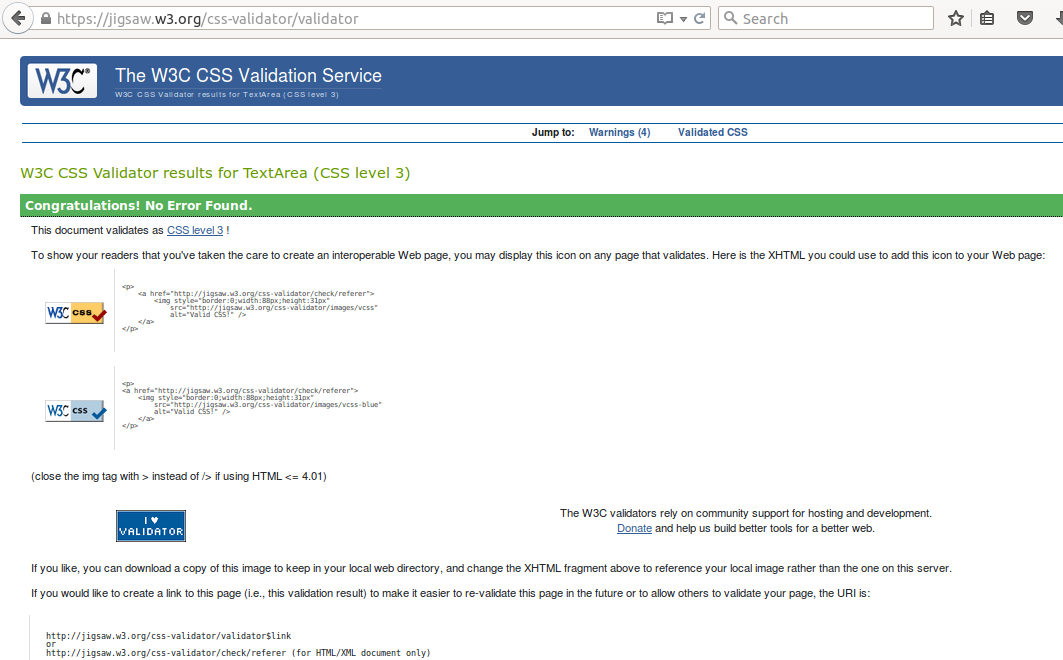
// then

assertEquals**(**duration10MinutesExpected**,** durationResult**);**

**}**

Oprócz testów jednostkowych zostały takie narzędzia jak walida tory kodu HTML i CSS.. Kod CSS został poddany walidacji za pomocą strony: <https://jigsaw.w3.org/css-validator/> Walidacja przebiegła pomyślnie, co prezentuje poniższa ilustracja:

Rysunek 6 – Wynik walidacja kodu CSS



Kod HTML został poddany walidacji za pomocą walidatora na stronie: <https://validator.w3.org/nu/> Jest to walidator html5. Walidacja HTML również przebiegła pomyślnie, pomijając niektóre ostrzeżenia. Ostrzeżenia pojawiały się w sytuacji gdy, struktura kodu html wynikała z zastosowanych funkcji javascript, bądź użycia tagów xml z biblioteki JSTL. Na poniższej ilustracji jest zaprezentowany wynik walidacji html strony głównej systemu.

Rysunek - Wynik walidacji kodu html



System został również poddany testom bezpieczeństwa za pomocą skanera bezpieczeństwa. Skan został przeprowadzony programem **Skipfish** w wersji **2.10b** na systemie operacyjnym **Linux** w dystrybucji **Ubuntu** w wersji **15.10.**  W celu wykonania skanu bezpieczeństwa program został uruchomiony na lokalnym serwerze **Jetty** w wersji **9.3.2.v20150730**  , na tej instancji został przeprowadzony skan. Skaner Skipfish porządkuje podatności według hierarchii:

* High risk
* Medium risk
* Low risk
* Warnings
* Notes (dane informacyjne)

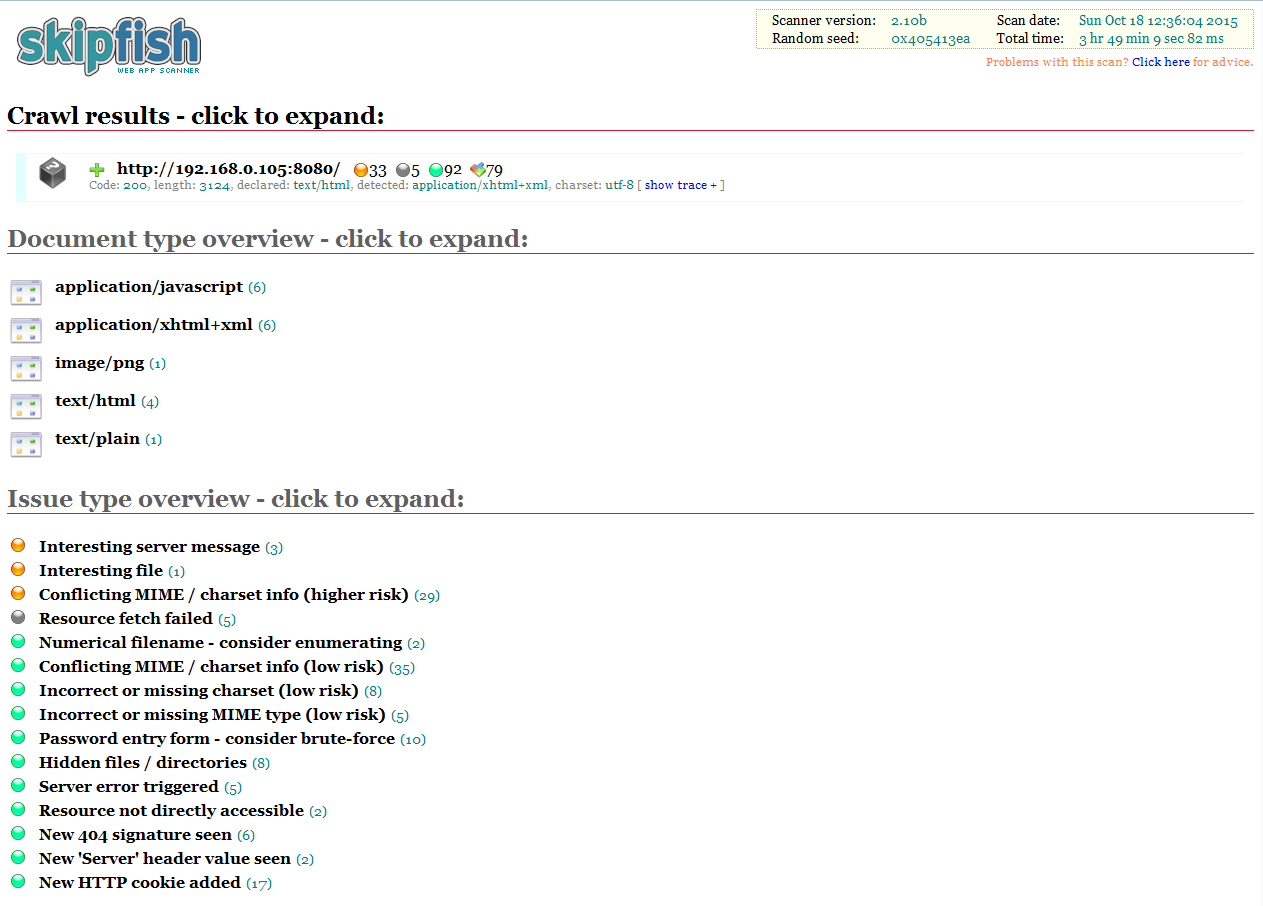
Skipfish po wykonaniu skanu generuje raport w postaci pliku html.

**Skan systemu przeprowadzony18 października 2015 wykazał:**

* Brak podatności wysokiego ryzyka
* Trzy typy podatności średniego ryzyka występujące w systemie
* Pięć ostrzeżeń
* Noty informacyjne (notes) w skład których wchodziły także podatności niskiego ryzyka

Raport skanu przedstawia poniższa ilustracja.

Rysunek - Raport skanu bezpieczeństwa



Raport wygenerowany przez Skipfish znajduje się w załącznikach.

# Prezentacja aplikacji

Aplikacja jest dostępna na hostingu unicloud.pl pod adresem

<http://iboard-games.unicloud.pl/>

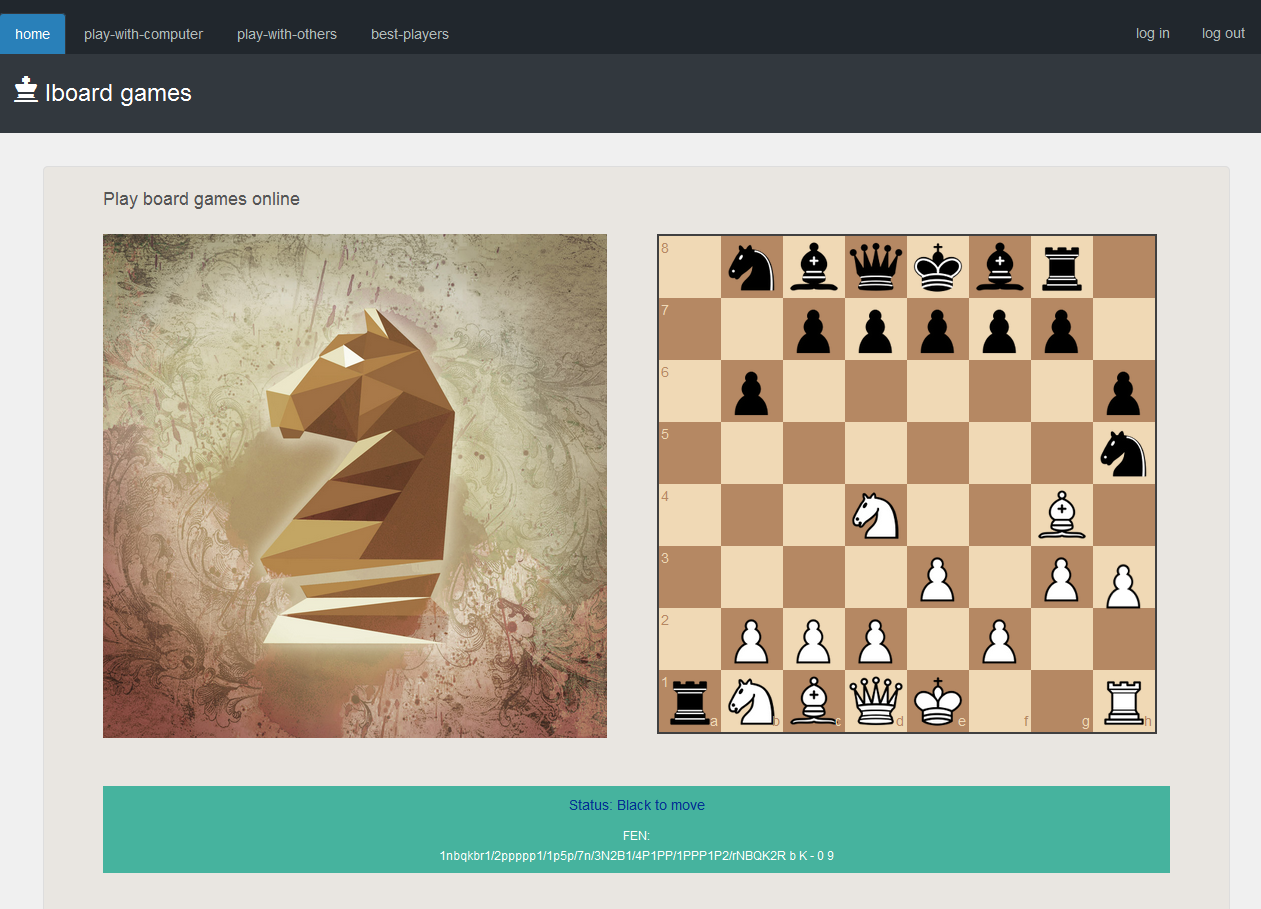
Posiada założone konta fikcyjnych użytkowników oraz konto administratora

Login: **unreal**

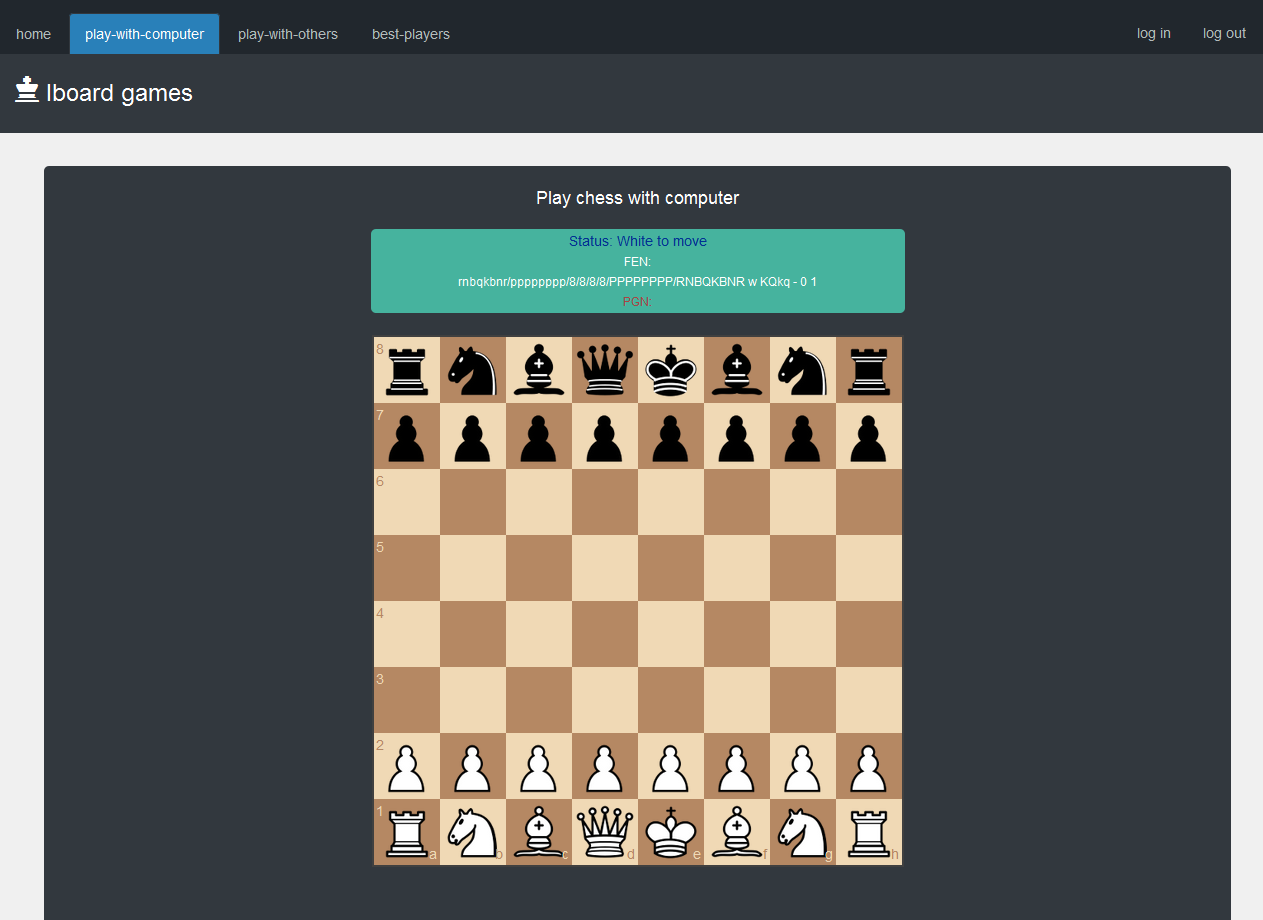
Hasło: **AdminNaSzachownicyNa100%**

Na poniższych ilustracjach zostaną zaprezentowane slajdy prezentujące wybrane funkcjonalności systemu, w kolejności przykładowego użycia programy, od anonimowego użytkownika poczynając po administratora kończąc.

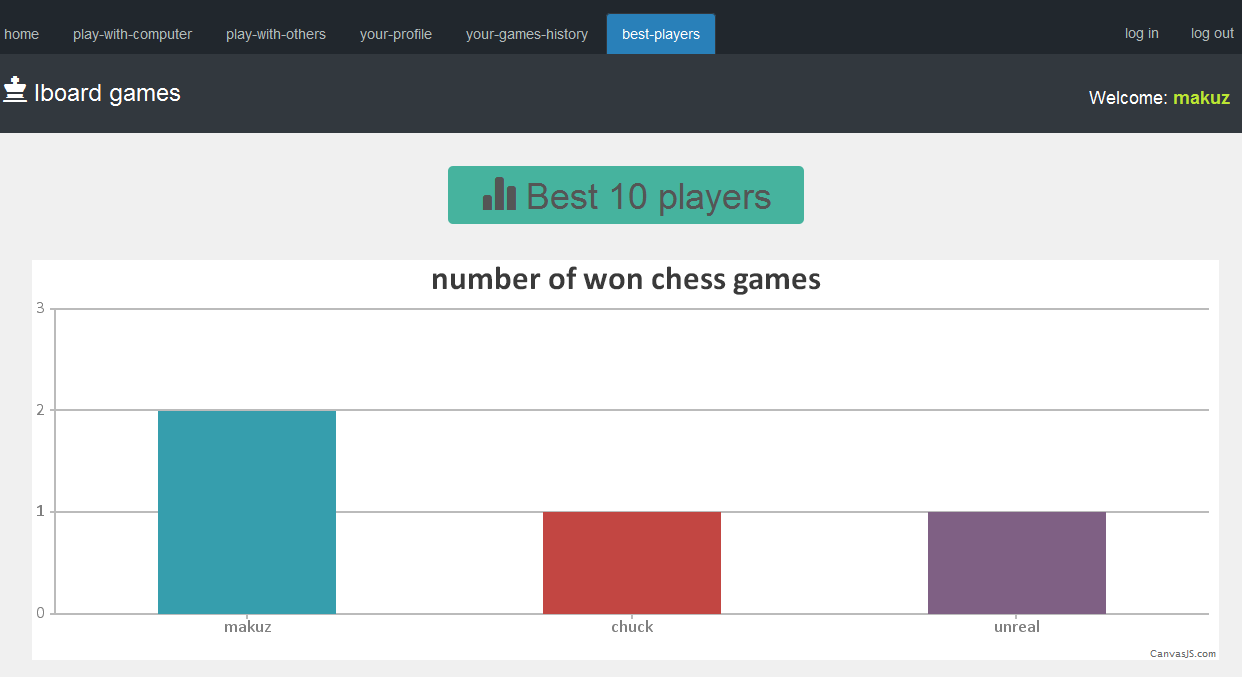
Strona domowa aplikacji jest dostępna dla wszystkich użytkowników. Zawiera grafikę oraz widok gry w szachy uruchomiony w trybie komputer kontra komputer jako demonstrację programu.



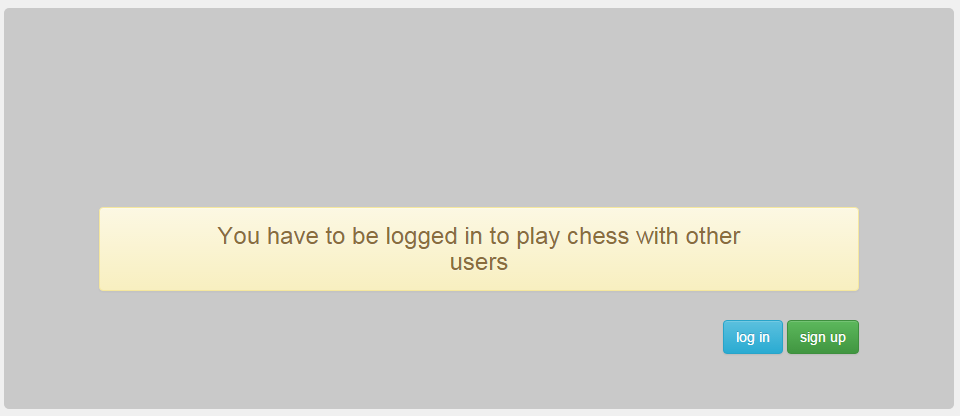
Użytkownik anonimowy może wejść na podstrone **play-with-computer** i zagrać w szachy z komputerem. Na poniższej ilustracji jest zaprezentowany interfejs do gry z komputerem.



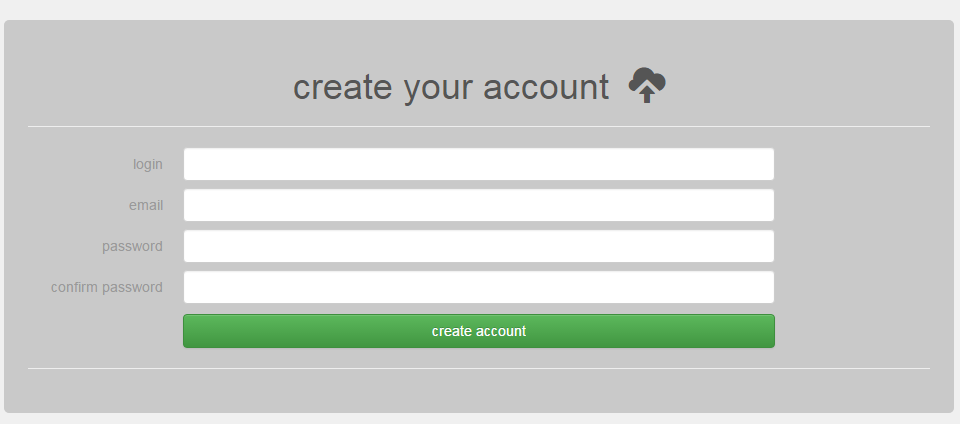
Anonimowy użytkownik ma również dostęp do podstrony **best-players** z rankingiem dziesięciu najlepszych graczy w formie wykresy słupkowego. Na poniższej ilustracji zaprezentowano stronę z rankingiem.



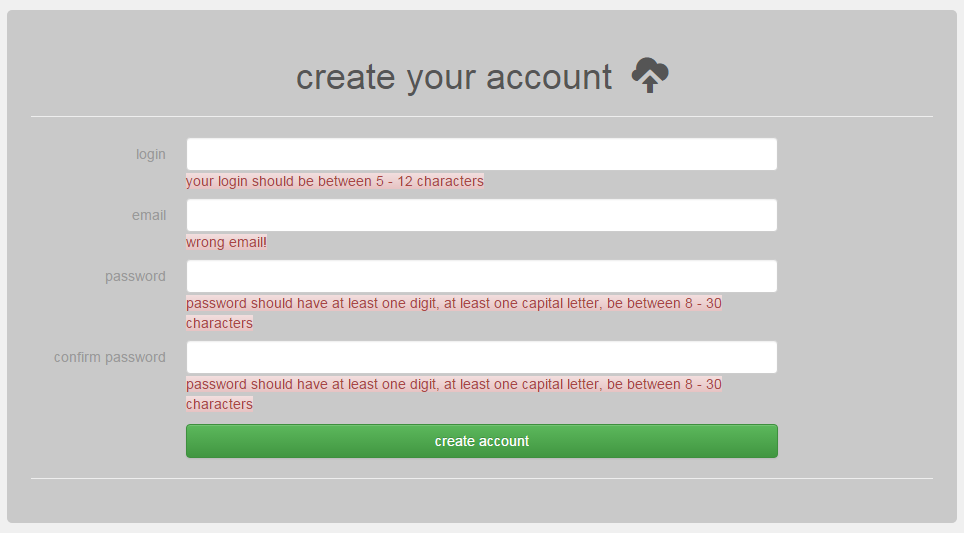
Anonimowy użytkownik nie ma dostępu do możliwości gry onlinę z innymi użytkownikami, gdy wchodzi na podstroję **play-with-others**, wyświetla się poniższa strona z komunikatem.



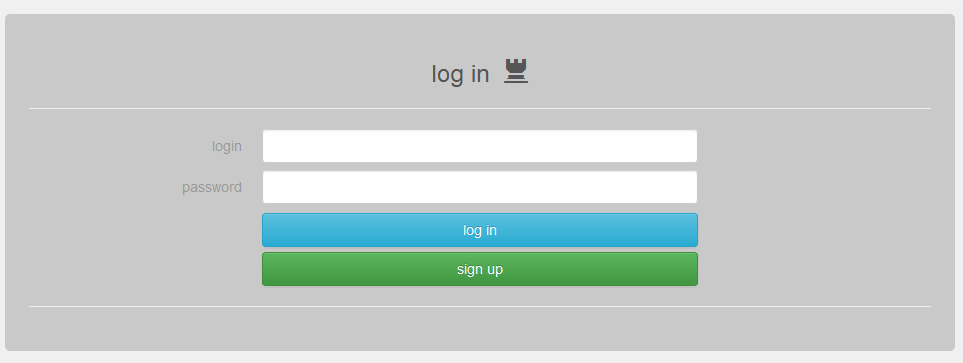
Użytkownik anonimowy może stworzyć konto za pomocą poniższego formularza rejestracji.



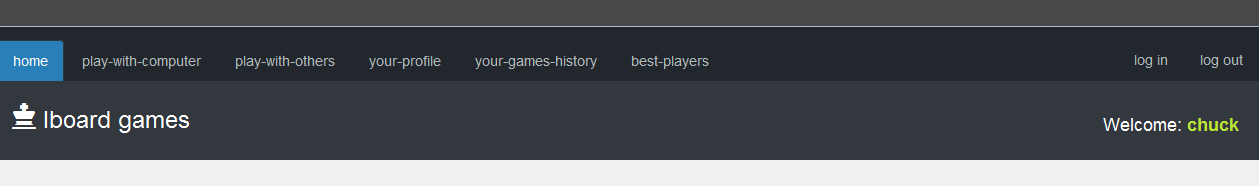
Formularz jest zabezpieczony walidacją danych. Poniższy rysunek prezentuje formularz z wyświetlonymi komunikatami walidacyjnymi.



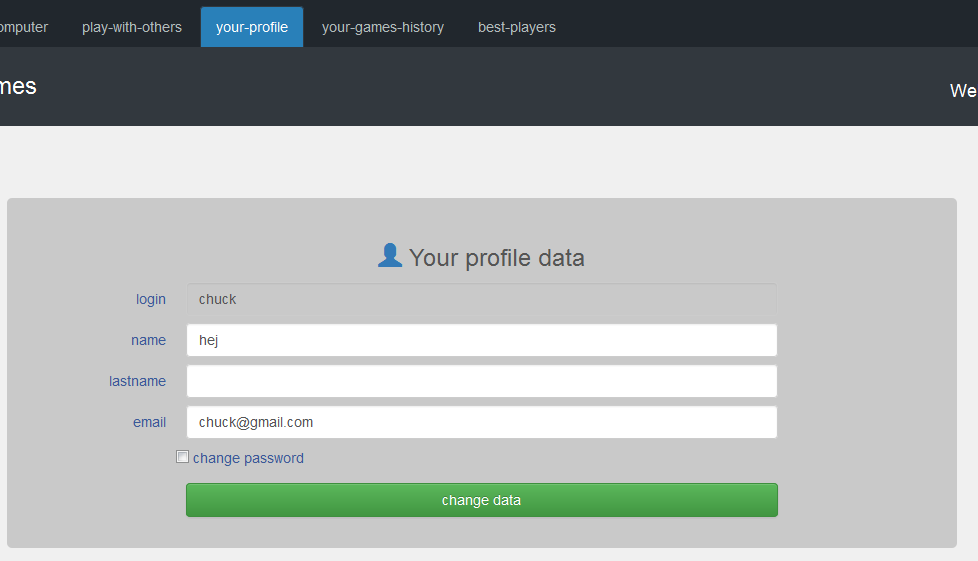
Zarejestrowany użytkownik może się zalogować za pomocą formularza logowania widocznego poniżej.



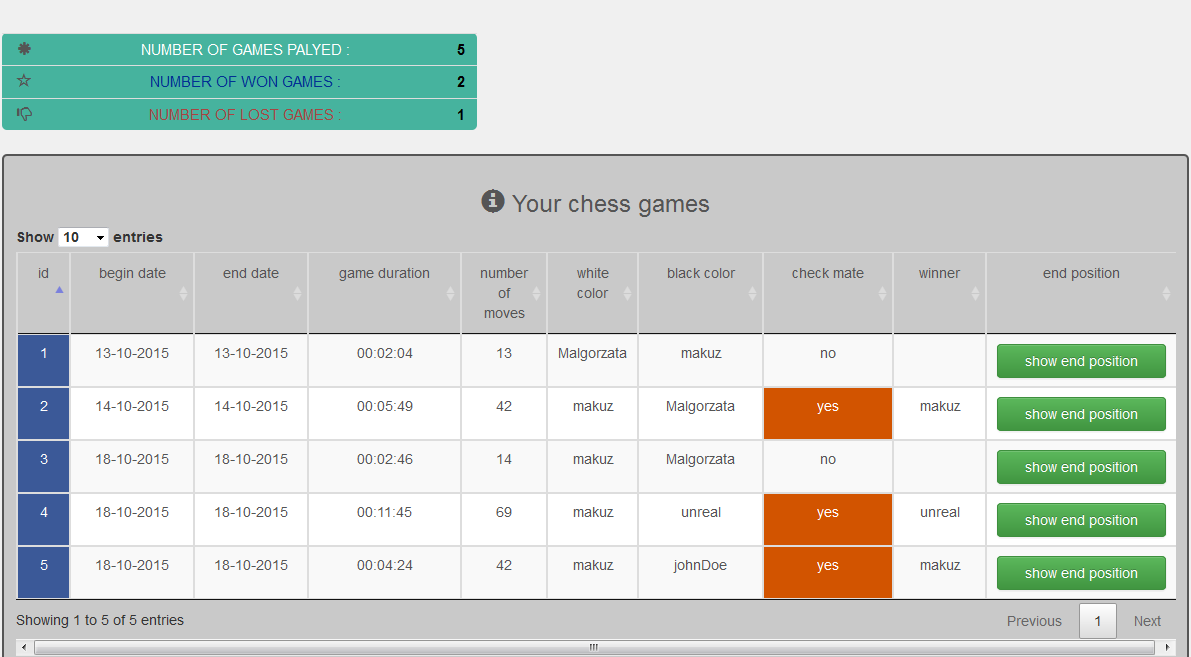
Po zalogowaniu w górnym prawym rogu wyświetla się nazwa aktualnie zalogowanego użytkownika.



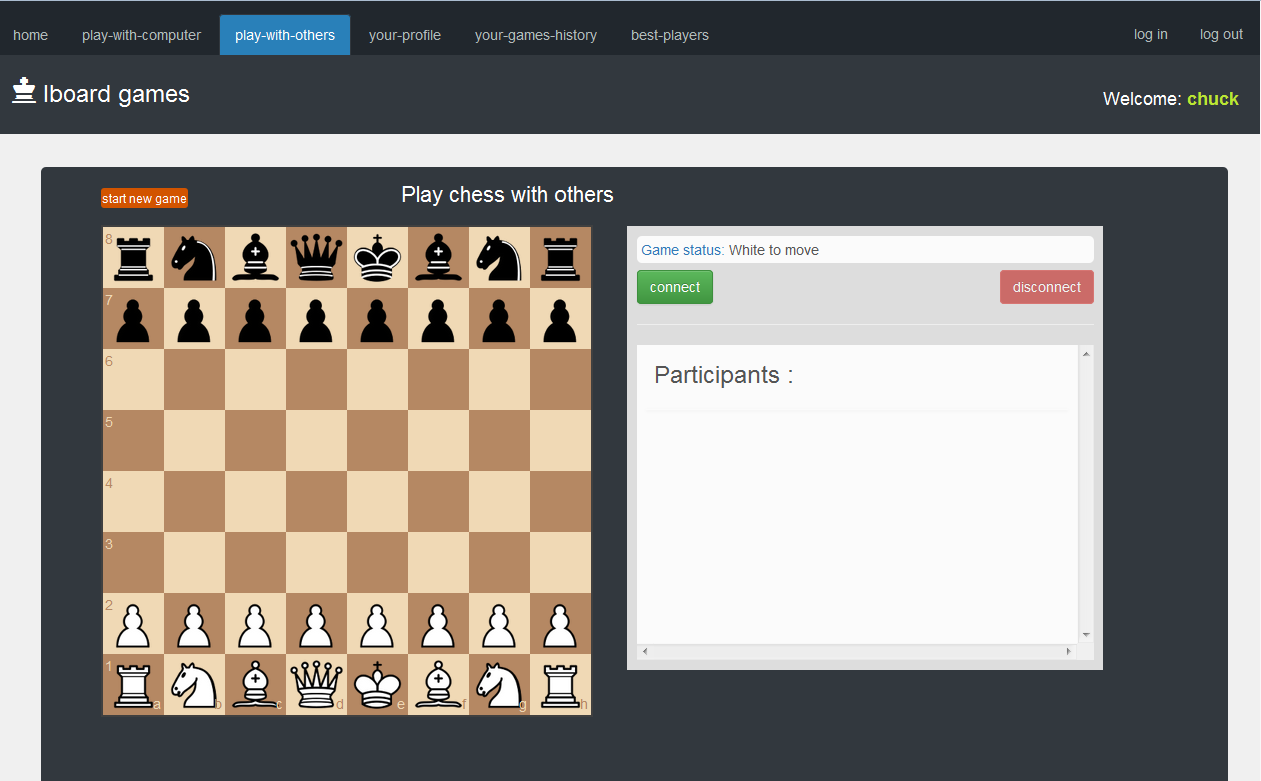
Po zalogowaniu użytkownik ma dostęp do dwóch dodatkowych podstron: **your-profile** oraz **your-games-history**. Your-profile to strona z formularzem danych użytkownika z możliwością ich edycji z wyłączeniem możliwości edycji loginu.



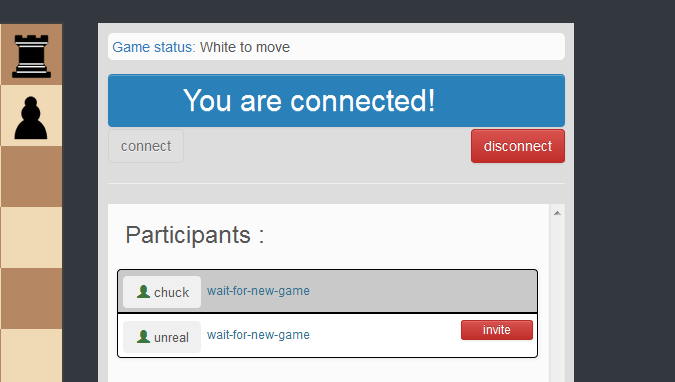
Podstrona **your-games-history** to tabelka z historią odbytych pojedynków szachowych , wraz z tabelką zestawiająca dane statystyczne: ilość odbytych pojedynków, ilość wygranych, ilość przegranych.



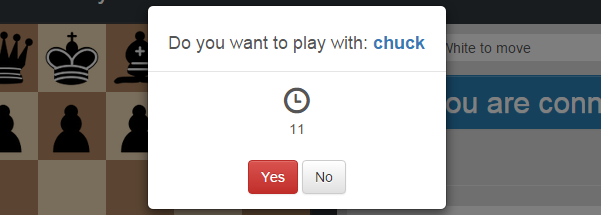
Zalogowany użytkownik ma dostep do interfejsu do gry online z innymi użytkownikami, poprzez korty może się podłączyć do repozytorium gry za pomocą websocket.



Po kliknięciu *connect*, wyświetla się lista dostępnych użytkowników z których można zaprosić do gry.



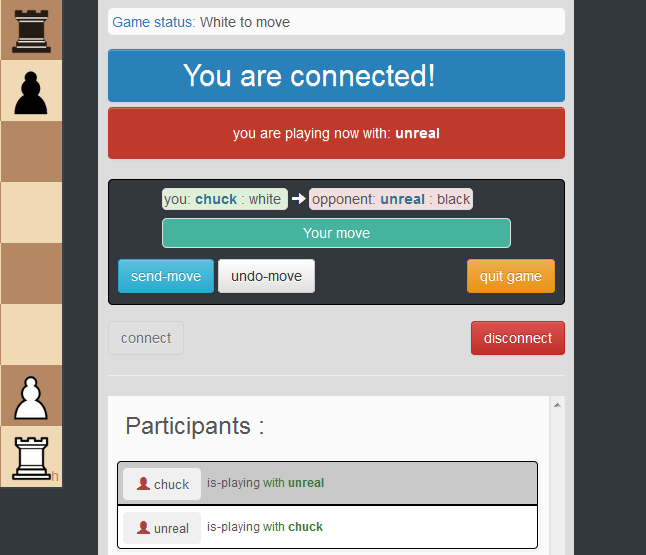
Po kliknięciu *invite* zaproszonemu użytkownikowi wyświetla się okno z zaproszeniem i timerem, dający mu czas na podjęcie decyzji.



Zaproszony użytkownik ma do wyboru zgodę na grę lub odmowę. Jeżeli odmówi to drugiemu użytkownikowi wyświetla się okno z komunikatem, że zaproszona osoba odmówiła. Jeżeli zaproszony użytkownik się zgodzi to drugiemu użytkownikowi wyświetla się okno z komunikatem, że zaproszona osoba się zgadza. Komunikaty powyższych sytuacji zaprezentowano poniżej.

|  |  |
| --- | --- |
| zgodanawspolnagre.PNG | odmowa.PNG |

Gdy zaproszona osobą się zgodzi gra się rozpoczyna. Interfejs użytkownika połączonego zmienia się w wzbogacony interfejs do prowadzenia gry.

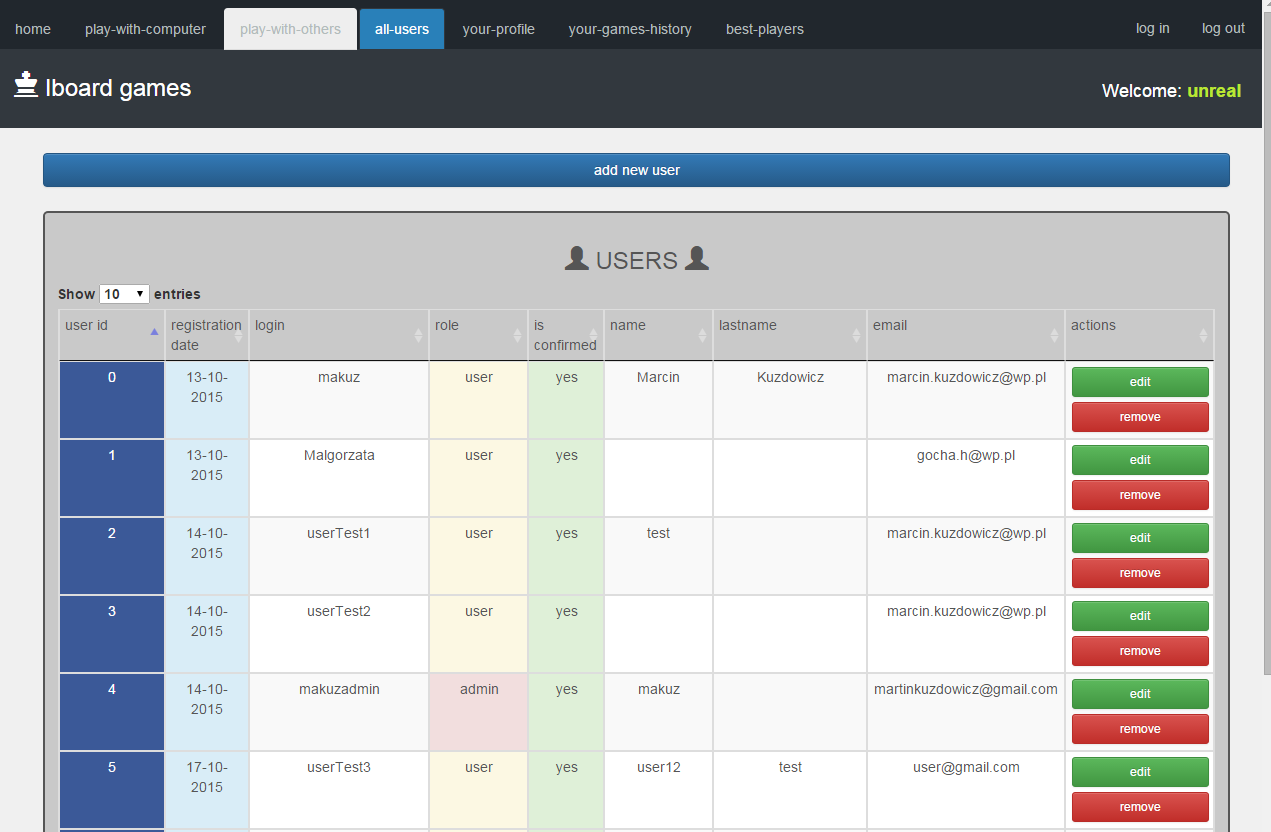


Teraz użytkownicy mogą wysyłać pomiędzy sobą ruchy szachowe. Gdy zajdzie zdarzenie końca gry w postaci „szach mat”, to dla przeciwników są wyświetlane odpowiednie komunikaty.

|  |  |
| --- | --- |
| komunikatowygranej.PNG | komunikatoprzegranej.PNG |

Po zajściu tych zdarzeń gra się kończy i dane na temat gry oraz wyniku są zapisywane do bazy danych.

Jeżeli zalogowany użytkownik jest administratorem to ma dostęp do dodatkowej podstrony **all-users**, która zawiera tabelkę z wszystkimi użytkownikami. Z pozycji tej podstrony administrator może edytować i usuwać istniejących użytkowników oraz dodawać nowych.



# Podsumowanie

**Aktualny stan aplikacji i jej wady**

Ze względu na fakt, iż opracowanie kompletnego systemu społecznościowego jest problemem dosyć obszernym, zdecydowano się na przyjęcie pewnych założeń upraszczających. Wybrany użytkownik może prowadzić tylko jedną grę z jednym użytkownikiem w danym czasie. Użytkownik może się połączyć do repozytorium gry tylko na podstronie *play-with-users* i tylko na tej podstronie jest utrzymywany jego kontekst jako użytkownika gry, w momencie zamknięcia tej podstrony lub przejścia na inną połączenie zostaje zrywane. Ranking najlepszych graczy jest prowadzony na podstawie prostych przeliczeń, czyli zliczania wygranych gier kończących się jako szach mat gdy zachodzi zdarzenie szach mat to pole ilość wygranych gier u zwycięscy jest inkrementowane. Takie przeliczenia statystyczne są dosyć proste. Nie jest prowadzone zliczanie pojedynczych punktów ani czasu trwania pomiędzy ruchami szachowymi.

**Zalety aplikacji**

Zaletą aplikacji jest fakt jej obsługi przez Internet. Użytkownik nie musi instalować na swojej maszynie żadnych dodatkowych zasobów aby móc z niej skorzystać. Możliwość zagrania w szachy zarówno z komputerem jak i wybranym użytkownikiem. Zaletą jest wykorzystanie technologii websocket, co umożliwia grę w czasie rzeczywistym, każde zdarzenie jakie zachodzi podczas gry między dwoma użytkownikami jak i w repozytorium podłączonych graczy jest automatycznie aktualizowane, bez konieczności odświeżania strony. Zaletą aplikacji jest również prosta budowa interfejsu użytkownika, co sprawia że jej obsługa jest bardzo intuicyjna i nie wymaga przygotowywania do tego użytkowników.

**Możliwości rozbudowy**

Zaprezentowany w niniejszej pracy system ma niewątpliwie możliwości rozbudowy. W przyszłości zaplanowanie jest aby rozbudować sposób prowadzenia rankingu najlepszych graczy i przeliczenia statystyczne. Przewidziane jest aby zaimplementować do tego wskaźnik Bergera. Wprowadzenie funkcjonalności zmiany hasła gdy użytkownik zapomniał swoje hasło przy pomocy jego adresu email, jaki został użyty do rejestracji bez konieczności angażowania do tego administratora. Możliwość prowadzenia gier z kilkoma graczami jednocześnie, przy ustaleniu maksymalnej liczby. Będzie do tego potrzebne rozbudowanie Interfejsu użytkownika i otwieranie każdej gry w osobnym oknie. Możliwość przerwania gry z danym użytkownikiem i dokończenia tej samej gry od miejsca zakończenia w innym terminie. Przetrzymywanie kontekstu repozytorium gry i gier prowadzonych przez użytkownika niezależnie od tego na jakiej podstronie użytkownik się znajduje, gra będzie otwarta w osobnym podoknie i będzie możliwość prowadzenia jej z każdej podstrony aplikacji. Rozbudowanie prezentacji graficznej aplikacji, rozwinięcie jej interfejsu pod względem użycia na komputerach PC jak i urządzeniach mobilnych. Możliwość zapraszania znajomych do serwisu z pozycji aplikacji przez podanie maila znajomego w formularzu i kliknięciu przycisku zaproś znajomego do gry w serwisie.

# Literatura

**Pozycje książkowe**

1. Amuthan G, Spring MVC. Przewodnik dla początkujących, Helion, Gliwice 2015.
2. Cay S. Horstmann, Gary Cornell, *Java. Podstawy. Wydanie IX*,wyd. Helion, Gliwice 2013.
3. Cay S. Horstmann, Gary Cornell, *Java. Techniki zaawansowane. Wydanie IX*,wyd. Helion, Gliwice 2013.
4. Craig Walls, *Spring w akcji. Wydanie IV*, wyd. Helion, Gliwice 2015.
5. Felipe Gutierrez, *Wprowadzenie do Spring Framework dla programistów Java*, wyd. Helion, Gliwice 2015.
6. Jacob Seidelin, *HTML5. Tworzenie gier*, wyd. Helion, Gliwice 2012.
7. Łukasz Pasternak, CSS3. Tworzenie nowoczesnych stron WWW, Helion, Gliwice 2012.
8. Mario Andres Pagella, *Tworzenie izometrycznych* gier społecznościowych w HTML5, CSS3 i JavaScript, wyd. Helion, Gliwice 2012.
9. Peter Lubbers, Brian Albers, Frank Salim, *HTML5. Zaawansowane programowanie*,wyd. Helion, Gliwice 2013.
10. Pramod J. Sadalage, Martin Fowler, *NoSQL. Kompendium wiedzy*, wyd. Helion, Gliwice 2015.
11. Syed Fazle Rahman, *Bootstrap. Tworzenie interfejsów stron WWW. Technologia na start!*, wyd. Helion, Gliwice 2015.
12. Thomas H. Cormen, *Algorytmy bez tajemnic*, wyd. Helion, Gliwice 2013.
13. Tom Negrino, DoriSmith, *Po prostu JavaScript Wydanie VIII,* wyd. Helion, Gliwice 2012.
14. Willie Wheeler, Joshua White, *Spring w praktyce*, wyd. Helion, Gliwice 2014.

**Źródła internetowe:**

1. Bootstrap, <http://getbootstrap.com/> [10.10.2015]
2. CanvasJs, <http://canvasjs.com/> [10.10.2015]
3. Chess.js, <https://github.com/jhlywa/chess.js/> [10.10.2015]
4. Chessboardjs, <http://chessboardjs.com/> [10.10.2015]
5. CSS3, <http://www.css3.info/> [10.10.2015]
6. DataTebels, <https://www.datatables.net/> [10.10.2015]
7. GIT, <https://git-scm.com/> [10.10.2015]
8. HTML5, <http://www.w3.org/TR/html5/> [10.10.2015]
9. Java web development tutorials, <http://www.mkyong.com/> [10.10.2015]
10. Java, <https://www.java.com/> [10.10.2015]
11. JavaScript, <https://www.javascript.com/> [10.10.2015]
12. JavaStart - Kurs programowania Java i Android, <http://javastart.pl/> [10.10.2015]
13. Jenkov.com, <http://jenkov.com/> [10.10.2015]
14. Jetty, <http://www.eclipse.org/jetty/> [10.10.2015]
15. JQuery, <https://jquery.com/> [10.10.2015]
16. JUnit, <http://junit.org/> [18.10.2015]
17. Mave, <http://maven.apache.org/> [10.10.2015]
18. MongoDB, <https://www.mongodb.org/> [10.10.2015]
19. Skipfish web application security scanner, <https://code.google.com/p/skipfish/> [18.10.2015]
20. Spring, <https://spring.io/> [10.10.2015]
21. Stackoverflow, <https://stackoverflow.com> [10.10.2015]
22. Walidator kodu CSS, <https://jigsaw.w3.org/css-validator/> [18.10.2015]
23. Walidator kodu HTML5, <https://validator.w3.org/nu/> [18.10.2015]

# Spis ilustracji

[Rysunek 1 - Diagram przyopadków użycia 12](#_Toc432970117)

[Rysunek 2 - model MVC 16](#_Toc432970118)

[Rysunek 3 - Diagram klas 22](#_Toc432970119)

[Rysunek 4 - - Model bazy danych 24](#_Toc432970120)

[Rysunek 5 - - Schemat blokowy komunikacji graczy 48](#_Toc432970121)

[Rysunek 6 – Wynik walidacja kodu CSS 50](#_Toc432970122)

[Rysunek 7 Waliodacja kodu CSS 50](#_Toc432970123)

[Rysunek 8 - Wynik walidacji kodu html 51](#_Toc432970124)

[Rysunek 9 - Raport skanu bezpieczeństwa 52](#_Toc432970125)

**Spis tabel**

[Tabela 1 - Słownik pojęć biznesowych 7](#_Toc432970157)

# Załączniki

1. Uruchomienie\_programu\_w\_środowisku\_developerskim.docx
2. ChessAppDiagramKlas.png
3. ChessAppDiagramPrzypadkowUzycia.png
4. SchematBlokowyKomunikatora.png
5. ChessAppModelBazyDanych.png
6. RaportSkipfish