|  |
| --- |
| **Politechnika Wrocławska** |
| **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:

1. Cel i zakres pracy
2. Opis wycinka rzeczywistości i słownik pojęć biznesowych
   1. Wstęp
   2. Słownik pojęć biznesowych
   3. Opis wycinka rzeczywistości
3. Specyfikacja wymagań funkcjonalnych i niefunkcjonalnych
   1. Wymagania funkcjonalne
      1. Ogólne wymagania systemowe
      2. Perspektywa administratora
      3. Perspektywa Użytkownika
      4. Wymagania niefunkcjonalne
   2. Diagram przypadków użycia
4. Wybór technologii, ogólna architektura systemu i opracowanie modelu danych oraz bazy danych
   1. Wybór technologii
   2. Architektura aplikacji
      1. Ogólna architektura
      2. Diagram klas
      3. Model bazy danych
      4. Hierarchia plików
5. Realizacja systemu
   1. Tworzenie kontsa użytkownika
   2. Logowanie
   3. Implementacja gry w szachy.
   4. Gra w szachy użytkownik versus użytkownik implementacja komunikatora do gry.
6. Testy
7. Podsumowanie

Litesratura

Spis rysunkó

Spis tabel

Załączniki

1. **Cel pracy:**

Celem pracy jest zaprojektowanie i implementacja systemu informatycznego, przy użyciu poznanych w toku studiów technologii internetowych, służącej do gry w szachy w sieci internetowej. Użytkownik aplikacji może grać z komputerem lub po utworzeniu konta i zalogowaniu z innym użytkownikiem w zależności od tego czy inni użytkownicy są w tym momencie zalogowani do systemy i obecni w repozytorium gry online. Aplikacja również ma gromadzić i wyświetlać dane statystyczne na temat wyników pojedynków szachowych użytkowników. Ma wyświetlać dla wszystkich widocznych dziesięciu najlepszych graczy, czyli takich którzy wygrali najwięcej pojedynków szachowych, pogrupowanych w według zdobytych punktów w postaci wykresu słupkowego. Program ma również gromadzić dane na temat odbytych gier poszczególnych użytkowników oraz prezentować aktualnie zalogowanemu użytkownikowi w jego panelu gry odbyte przez niego.

1. **Opis wycinka rzeczywistości.**
   1. **Wstęp**

Jednym z najciekawszych funkcji współczesnego Internetu jest możliwość komunikacji w czasie rzeczywistym miedzy dwoma osobami znajdującymi się fizycznie w innych miejscach. Funkcjonalność ta może służyć do wymiany słów między ludźmi w formie tekstowej, jak również wymianą informacji nie tekstowej, co jest nową sferą komunikacji udostępnionej przez nowe technologie.

Interesującym tematem są tak zwane gry komputerowe online. Sferą zainteresowań tej pracy jest stworzenie takiego właśnie programu, który będzie służył do gry online miedzy użytkownikami Internetu.

Przyczyny podjęcia tematu

Sam temat jest interesujący pod wzglądem programistycznym, gdyż będzie wymagał użycia szerokiej palety technologii, jak i zastosowanie złożonej logiki mającej na celu obsłużyć komunikację w czasie rzeczywistym.

Głównie z tego względu został podjęty.

Aplikacja ma na celu umożliwić zalogowanym użytkownikom prowadzenie pojedynków szachowych między sobą online.

* 1. **Słownik pojęć biznesowych**

|  |  |
| --- | --- |
| Użytkownik | Osoba korzystająca z oprogramowania |
| Administrator | Osoba korzystająca z oprogramowania i mająca uprawnienia do tego aby zarządzać użytkownikami |
| Szachy | Rodzaj gry planszowej będący głównym tematem niniejszego oprogramowania |
| Szachownica | Obiekt stanowiący, w której odbywa się gra |
| Pojedynek szachowy online | Gra w szachy pomiędzy dwoma użytkownikami w czasie rzeczywistym za pomocą oprogramowania |
| Lista najlepszych graczy | Posortowana Lista dziesięciu użytkowników, którzy wygrali największą liczbę pojedynków szachowych |
| Figura szachowa |  |
| Kolor figury szachowej |  |
| Status gry | Aktualna sytuacja jaka występuje w grze: ruch dla figur białych, szach, szach mat. |
| Fen | Notacja Forsytha-Edwardsa. Ciąg znaków reprezentujący, aktualny status gry, położenie wszystkich figur na szachownicy; |
| Ruch szachowy | Ruch szachowy |
| Komunikator do gry online | Część oprogramowania służąca do wysyłania ruchów szachowych między dwoma użytkownikami przez sieć internetową |
| Oczekiwanie na nową grę | Stan w jakim znajduje Się użytkownik, który w danym momencie z nikim nie gra a jest online |
| Zaproszenie do gry | Stan w jakim znajduje Się użytkownik, został zaproszony do gry lub sam zaprosił do gry |
| Stan w trakcie gry | Stan w jakim znajduje Się użytkownik, który aktualnie prowadzi grę z innym użytkownikiem |
| Pojedynek szachowy | Dwóch użytkowników grających przeciwko sobie |
| Lista najlepszych graczy | 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 |

* 1. **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 użytkownik Internetu może odwiedzić stronę z aplikacją, ale 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:

1. Administrator
2. Użytkownik

Rola Użytkownika powstaje poprzez rejestrację w systemie, za pomocą formularza do rejestracji obecnego w aplikacji. Użytkownik musi podać login, który ma być unikalny dla systemu, nie mogą wystąpić takie same nazwy użytkownika. Jeżeli login podany podczas rejestracji już istnieje, zostanie wyświetlony komunikat walidacyjny, aby stworzyć inny login. Oprócz tego użytkownik podaje hasło, potwierdzenie hasła oraz adres email. Następnie konto zostaje utworzone, ale jeszcze należy je aktywować. Na podany adres email zostanie wysłany link aktywacyjny. Jeżeli konto nie zostanie potwierdzone w przeciągu tygodnia to zostanie usunięte.

1. **Specyfikacja wymagań funkcjonalnych i niefunkcjonalnych**

**3.1 Wymagania funkcjonalne**

**3.1.1 Ogólne wymagania systemowe**

Na podstawie przeprowadzonych badań sformułowano następujące wymagania funkcjonalne:

WF/01 dodawanie/usuwanie użytkownika przez administratora

WF/02 tworzenie kont użytkowników przez użytkowników internetu poprzez formularz tworzenia konta użytkownika, weryfikowanych za pomocą linka aktywacyjnego wysyłanego, na wcześniej podany adres email

WF/03 możliwość gry w szachy z komputerem

WF/04 możliwość gry w szachy z innym użytkownikiem w czasie rzeczywistym

WF/05 przeglądanie/danych na temat odbytych wcześniej pojedynków szachowych z innymi użytkownikami

WF/06 przeglądanie aktualnego rankingu najlepszych dziesięciu graczy w postaci wykresu słupkowego

WF/07 pojedynek szachowy wstępuje w postaci użytkownik kontra użytkownik

WF/08 użytkownik może prowadzić tylko jeden pojedynek w czasie rzeczywistym z innym użytkownikiem

WF/09 użytkownik przed rozpoczęciem pojedynku musi zaprosić drugiego użytkownika do gry. Zaproszony użytkownik ma 15 sekund na decyzje, w momencie trwania zaproszenia, dana para użytkowników już jest zablokowana i inni użytkownicy nie mogą do nich wysłać wiadomości.

WF/10 użytkownik po rejestracji nie może zmienić swojego login, gdyż login jest identyfikatorem w grze online dal innych użytkowników

WF/11 użytkownik może edytować po rejestracji swoje dane takie jak: adres email, imię, nazwisko. Może również zmienić hasło

WF/12 administrator może przeglądać listę użytkowników

* + 1. **Perspektywa administratora**
* Administrator ma te same uprawnienia co zalogowany użytkownik
* Administrator ma dostęp do tych samych zasobów co zalogowany użytkownik
* Administrator systemu może dodawać nowych użytkowników
* Usuwać istniejących.
* Tworzyć konta z uprawnieniami administratora
* Nadawać uprawnienia administratora lub je odbierać.
  + 1. **Perspektywa Użytkownika**

Niezalogowany użytkownik ma dostęp do:

* Strony domowej
* możliwości zagrania w szachy z komputerem, aby zapoznać się z systemem. Wyniki pojedynków z komputerem nie są zapisywane do bazy, pełnią rolę warstwy demonstracyjnej.
* wykresu słupkowego z maksymalnie 10 najlepszymi graczami (przy założeniu że w bazie mamy dziesięciu graczy, którzy odbyli zwycięskie pojedynki w 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, plus:

* Możliwość grania w szachy z innymi użytkownikami online.
* Danych na temat swojego konta i możliwością ich edycji z wyłączeniem zmiany loginu.
* Danych na temat odbytych pojedynków szachowych
  1. **Wymagania niefunkcjonalne**

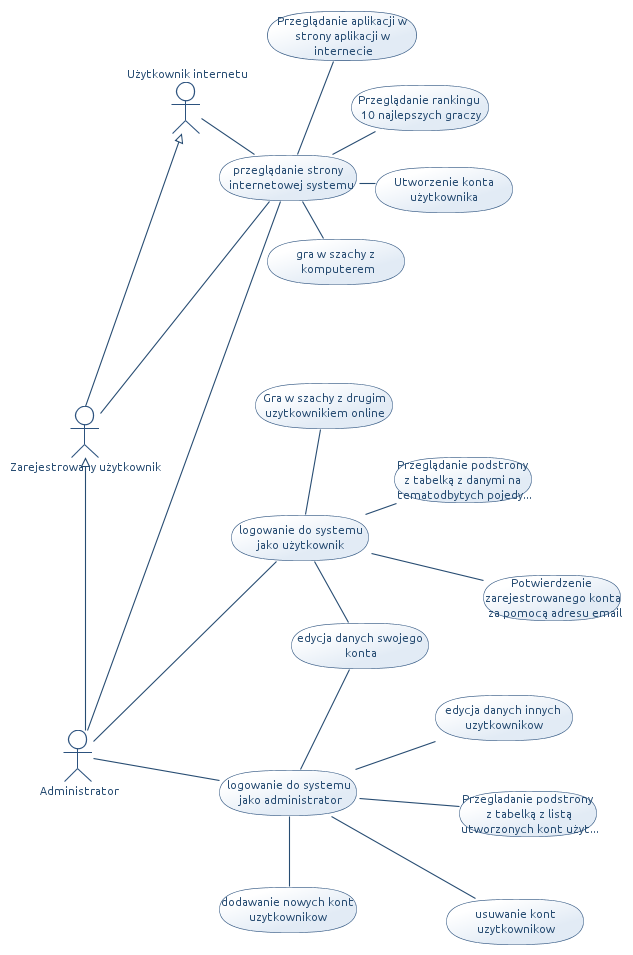
Program jest aplikacją webową

Program ma czytelny interfejs dla użytkownika

Program ma służyć rozrywce

Program propaguję grę w szachy w sieci

Program ma rozwijać umiejętności logicznego myślenia u użytkownika

* 1. **Diagram przypadków użycia**

1. **Wybór technologii, ogólna architektura systemu i opracowanie modelu danych oraz bazy danych**

**4.1 Wybór technologii**

* Java – obiektowy język programowania, główna technologia użyta w aplikacji do stworzenia kodu działającego po stronie serwera.

Więcej informacji: <https://www.java.com/pl/>

* Spring MVC – Framework języku java wspierający architekturę Model-View-Controller i gotowe komponenty, które mogą być wykorzystane do opracowania aplikacji internetowych.

Więcej informacji: <http://spring.io/>

* Spring security - Framework języku java wspierający zagadnienie bezpieczeństwa takie jak uwierzytelnianie, autoryzację.

Więcej informacji: <http://spring.io/>

* 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 htm jak i Java. Skłądnia jsp składa się na elementy: html, elementy skryptowe skryplety (kod w języku Java kontrolujący proces generowania dokumentu) oraz elementy składniowe tzw. Expression Language oraz tagi XML wywołujące określone metody.
* JSTL (Java Standard Tag Library) – kolekcja tagów z przypisanymi funkcjonalnościami. Jstl wspiera typowe zadania takie jak iteracja czy konstrukcje warunkowe, wykonywane na widoku.
* Jetty – serwer www oraz kontener serwletów Javy. Więcej informacji: <http://www.eclipse.org/jetty/>
* 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 zewnętrzne biblioteki importowane do projektu w postaci plików .jar

Więcej informacji: <http://maven.apache.org/>

* JSON – (Javascript Object Notation) format wymiany danych komputerowych, przypominający swoją strukturą tzw. Hashmapę, organizujący dane w postaci (klucz=wartość). Jest to format transportowy, którym posłużyłem się do wysyłania ruchów szachowych pomiędzy użytkownikami, aktualnego stanu gry, zaproszenia do gry czy zrezygnowania z gry.
* XML (Extensible Markup Language) czyli rozszerzalny język znaczników. W projektach javowoych bardzo często używany do plików konfiguracyjnych, oraz do tagów z przypisanymi 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 jaki listę aktualnie połączonych użytkowników. Dzięki 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 interfejsu użytkownika programowania zdarzeniowego. Interfejs do gry w szachy jak i logika gry w szachy jest obsługiwana zdarzeniowo za pomocą javascript drag and drop (m.in. zdarzenia drag start i dragstop).

Więcej informacji: <https://www.javascript.com/>

* Jquery – biblioteka jasvacritpu dostarczająca dodatkowe narzędzia i umożliwiająca zaprogramowanie tych samych funkcjonalności za pomocą mniejszej ilości kodu.

Więcej informacji: <https://jquery.com/>

* Bootstrap 3 – biblioteka css i javascript do tworzenia interfejsów użytkownika stworzona przez programistów Twittera. Główną cechą bootsrapa są dostarczane przez niego gotowe klasy css, gwarantujące dobry wygląd interfejsu i jego responsywność.

Więcej informacji: <https://jquery.com/>

* MongoDB – baza danych typu NoSql.   Charakteryzuje się dużą skalowalnością, wydajnością oraz brakiem ściśle zdefiniowanej struktury obsługiwanych baz danych. Zamiast tego dane składowane są jako dokumenty w stylu JSON, co umożliwia aplikacjom bardziej naturalne ich przetwarzanie, przy zachowaniu możliwości tworzenia hierarchii oraz indeksowania. Do aplikacji typu live-game baza danych tego typu jest dobrym wyborem ze względu na szybkość działania i dynamiczne dostosowywanie się do zapisywanych do niej obiektów. Cechą charakterystyczna jest to że gdy chcemy do tzw. Kolekcji (odpowiednik tabelki w bazach typu sql) dodać dodatkowe pole, to nie musimy najpierw go tworzyć, ono samo się utworzy przy pierwszym zapisie, co jest bardzo efektywne.

Więcej informacji: <https://www.mongodb.com/>

* Html 5 - jest najnowszą wersją standardu opisującego język HTML. Zalety tej technologii zostały użyte np. przy walidacji formularzy po stronie klienta np. za pomocą atrybutu required czy type=”email” oraz w przekazywaniu danych do javascriptu za pomocą data attributes.

Więcej informacji: <http://www.w3.org/TR/html5/>

* CSS 3 – kaskadowe arkusze stylów, język odpowiedzialny za wygląd aplikacji.

Więcej informacji: <http://www.css3.info/>

* DataTables – plugin do jquery umożliwający szybkie oprogramowanie tabelki wraz z paginacją zaimplementowaną po stronie klienta w javascript.

Więcej informacji: <https://www.datatables.net/>

* Canvasjs – biblioteka javascript umożliwiająca wyświetlenie responsywnych histogramów, słupków, wykresów prezentujących dane statystyczne w przyjazny dla użytkownika formacie. W programie użyta do wyświetlenia rankingu 10 najlepszych graczy w postaci wykresu słupkowego.

Więcej informacji: <https://www.datatables.net/>

* Chessboardjs - biblioteka javascript dostarczająca interfejs graficzny do gry w szachy.

Więcej informacji: <http://chessboardjs.com/>

* Chess.js - biblioteka javascript implementująca logikę gry w szachy umożliwiająca integrację z Chessboardjs i obsługę oprogramowanie gry w szachy zdarzeniowo przez programistę.

Więcej informacji: <https://github.com/jhlywa/chess.js/>

* GIT – System kontroli wersji umożliwiający zarządzanie poszczególnymi wersjami aplikacji i przechowywanie wszystkich wersji na lokalnym i zdalnym repozytorium.

Więcej informacji: <https://git-scm.com/>

**4.2 Architektura aplikacji.**

**4.2.1 Ogólna architektura aplikacji.**

Program jest oparty na wzorcu architektonicznym MVC (model view controller) zaimplementowanej przy pomocy komponentów dostarczanych przez framework Spring MVC.



Wszystkie klasy Javowe znajdują się w pakiecie **src/main/java**

Wszystkie requesty HTTP obsługują klasy w pakiecie **com.chessApp.controllers .**

Te klasy pełnią rolę **kontrolerów**. Są opatrzone anotacji @Controller co informuje Springa o tym że ta klasa jest kontrolerem.

**Widok** jest realizowany za pomocą technologii: Jsp (java server pages) , Jtsl (JavaServer Pages Standard Tag Library), HTML 5, CSS 3, JavaScript i Jquery oraz Bootstrap . Pliki widoków znajdują się w głównym folderze: **/src/main/webapp**

w folderze **assets** są pliki css i javascript

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

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

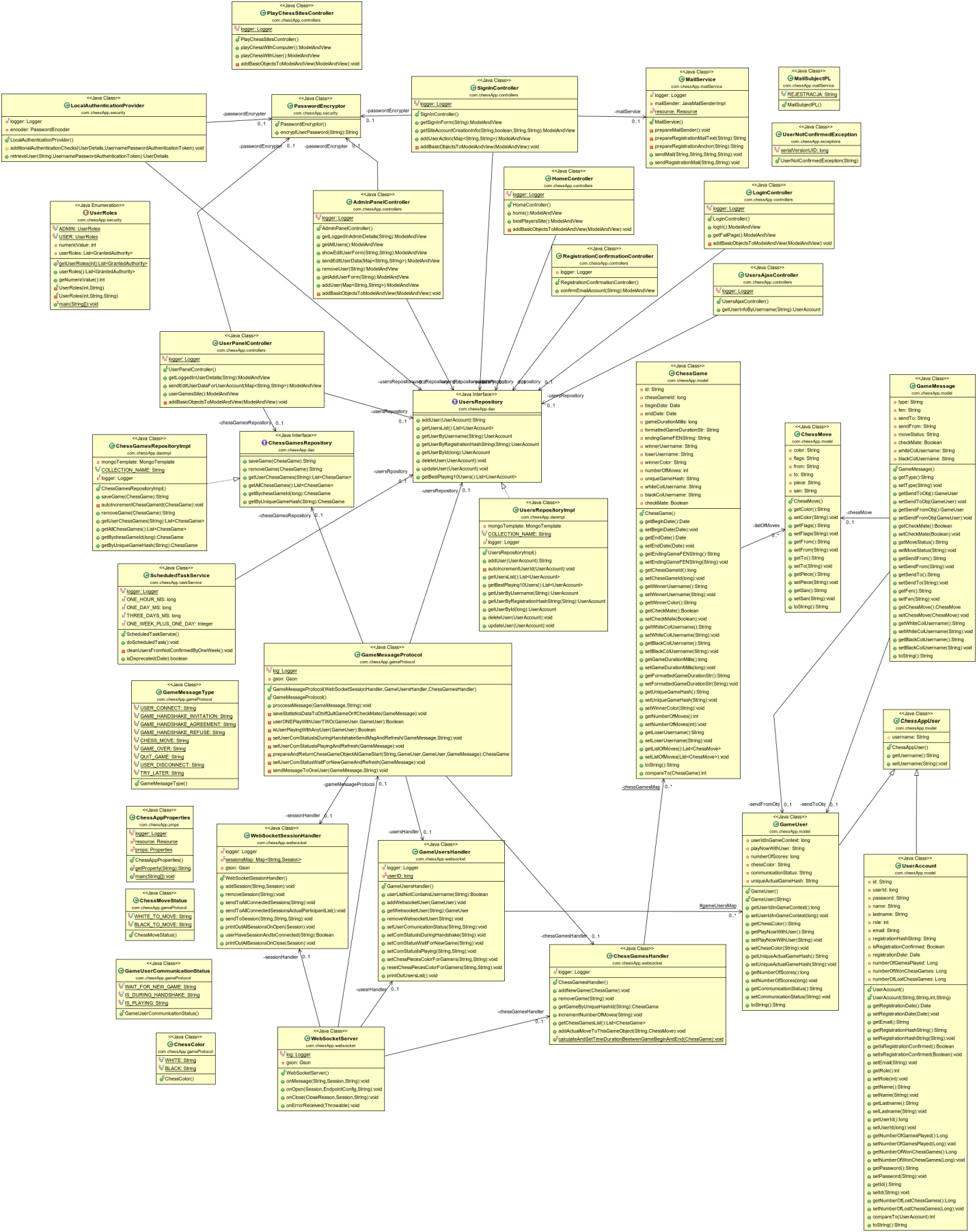
w folderze **WEB-INF/** **views** są pliki .jsp

**Model** jest zmapowany do klas javovwych w pakiecie **com.chessApp.model.**

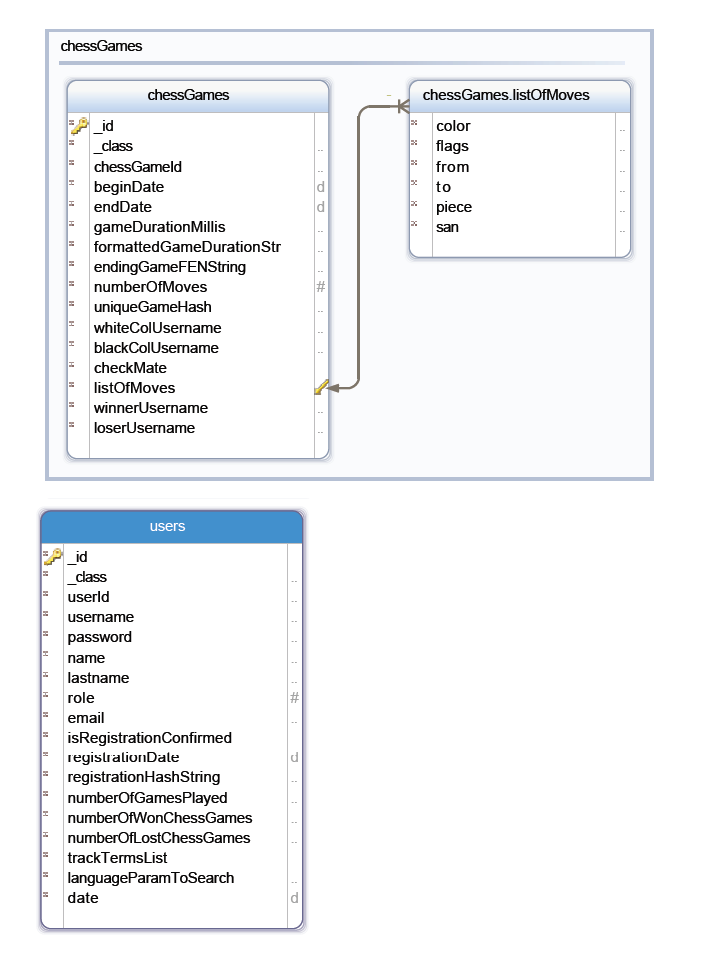
Aplikacja wykorzystuje bazę danych MonogDB, która jest bazą NoSQL. W programie jest zastosowana koncepcja **ORM** (object relational mapping), za pomocą technologii Spring Data. ORM mapuje obiekty i kolekcje z bazy danych MongoDB do klas Javovych i dzięki zaletmo jakie daje MongoDB, ten mechanizm zachodzi też odwrotnie.

Oprócz tego w folderze **WEB-INF** znajdują się pliki konfiguracyjne .xml

**4.2.2 Diagram klas**

****

**4.2.3 Model bazy danych**

Nazwa bazy danych**: chessapp\_db**

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

**MongoDb** nie jest bazą relacyjną, nie występuje tu koncepcja kluczy obcych, natomiast jest możliwość przechowywania zagnieżdżonych obiektów i list typów prostych i stringów jak również list obiektów. Ta właściwość została wykorzystana w implementacji bazy: kolekcja chessGames przechowuje listę obiektów Move. Obiekt Move eprezentuje ruch szachowy.

Model bazy danych został zmapowany do obiektów javowych: UserAccount, ChessGame, Move obecnych w pakiecie **com.chessApp.model**

Między obiektami ChessGame i Move występuje relacja Has-a. W postaci agregacji: klasa **ChessGames** zawiera listę obiektów Move

1. **Realizacja systemu**

Przybliżone zostaną wybrane zagadnienia implementacji systemu

**przybliżenie implementacji poszczególnych elementów aplikacji**

**5.1 Tworzenie konta użytkownika**

**5.2 logowanie**

**5.3** **Implementacja**  **gry w szachy.**

Interfejs do gry w szachy oraz logika do gry jest napisana po stronie klienta w języku javascript. Jest to realizowane przy pomocy biblioteki javascript: **chessboard.js** (interfejs)

Link: <http://chessboardjs.com/>

**chess.js** (silnik gry)

link: <https://github.com/jhlywa/chess.js/blob/master/README.md>

Pliki źródłowe z tymi bibliotekami są obecne w pakiecie w katalogu:

**/src/main/webapp/assets/js/lib/**

W katalogu **/src/main/webapp/assets/js/** umieszczone są pliki źródłowe z implementacji zdarzeniowego api udostępnionego przez powyższe biblioteki.

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:**

/\*\*

\* html objects on home page:

\*

\* div#board, div#status, div#fen, div#pgn

\*/

***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' // NOTE: always promote to a queen for example

// simplicity

**});**

// illegal move

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

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

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

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

updateStatus**();**

**};**

// update the board position after the piece snap

// for castling, en passant, pawn promotion

***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 sprawdzamy czy możemy wykonać ruch, czy ruch należy do na s, 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:

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

Z kolei funkcja fen() to aktualny stan gry według notacji fen w postaci stringa.

UpdateStatus wywolany w zdarzeniu onDrop aktualizuje obiekty przechowywujace aktualny status gry w postaci ciągu znaków fen i jeżeli to szach mat to wysyłana jest o tym wiadomość do serwera w postaci obiektu 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 Gra w szachy użytkownik versus użytkownik implementacja komunikatora do gry.**

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.

Do tego została użyta technologia technologii Websocket. Klasy implementujące komunikator znajdują się w pakiectach: **com.chessApp.websocket** i **com.chessApp.** **gameProtocol**.Implementacja komunikatora opiera się na obsłudze zdarzeń po stronie serwera w postaci **Server endpoint (zaprogramowanego w Javie) i** po stronie klienta w postaci Client endpoint (**zaprogramowanego w JavaScript**), oraz przechowywaniu wybranych danych w pamięci serwera w kontekście połączenia wybranego użytkownika do Server endpoint.

Server endpoint jest 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:**

**Server endpoint:**

package com**.**chessApp**.**websocket**;**

**import** java**.**io**.**IOException**;**

**import** javax**.**websocket**.**CloseReason**;**

**import** javax**.**websocket**.**EndpointConfig**;**

**import** javax**.**websocket**.**OnClose**;**

**import** javax**.**websocket**.**OnError**;**

**import** javax**.**websocket**.**OnMessage**;**

**import** javax**.**websocket**.**OnOpen**;**

**import** javax**.**websocket**.**Session**;**

**import** javax**.**websocket**.**server**.**PathParam**;**

**import** javax**.**websocket**.**server**.**ServerEndpoint**;**

**import** org**.**apache**.**log4j**.**Logger**;**

**import** org**.**springframework**.**stereotype**.**Service**;**

**import** com**.**chessApp**.**gameProtocol**.**GameMessageProtocol**;**

**import** com**.**chessApp**.**gameProtocol**.**GameMessageType**;**

**import** com**.**chessApp**.**gameProtocol**.**GameUserCommunicationStatus**;**

**import** com**.**chessApp**.**model**.**GameMessage**;**

**import** com**.**chessApp**.**model**.**GameUser**;**

**import** com**.**google**.**gson**.**Gson**;**

@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 obsługująca zdarzenia: **onOpen**, **onMessage**, **onClose** i **onError**. Na podstawie tych zdarzeń wywoływanych przez klienta aplikacji wywoływane są określone działania na serwerze.

Serwer w swojej pamięci 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 została stworzona hashmapa webSocketSessions<String username, Session session>. Nazwa użytkownika jest kluczem, sesja polaczeniem do którego można wysłać wiadomość przy zdarzeniu onMessage. Szczególy na temat użytkowników są przechowywanie w hashmapie gameUsers<String username, GameUser gameUser> Ta kolekcja przechowuje użytkowników gry w postaci obiektów ktroe przechowuja informacje potrzebne do tego aby rpowadzic gre z innymi użytkownikami, kluczem do tej kolekcji jest także nazwa użytkownika. Przy tym rozwiązaniu zarówno sesja danego uzytkonika jak i jego reprezentacja w postaci obiektu maja ten sam klucz dzieki temu pozostają w relacji one to one. Jeden użytkownik ma swoje konto zarejestrowane w aplikacji co reprezentuje obiekt UserAccount. W kontekście gry jest reprezentowany obiektem GameUser i obiektem Session polaczenia webSockte i wszystkie te obiekty są powiązane tą samą nazwą użytkownika.

Serwer także przechowuje w swojej pamieci operacyjnej informacje na temat aktualnie toczących się gier w kolekcji Map<String, ChessGame> chessGamesMap Kluczem jest tutaj randomowo generowany unikatowy hash, wartością obiekt gry. Uzytkownicy uczestniczący w pojedynku przechowuja w swoim obiekcie GameUser hash aktualnie toczonej przez siebie gry, za pomocą właściwości uniqueGameHash tworzą w ten sposób unikatową parę, zamkniętą w konteście prowadzonej przez siebie gry.

Akcje związane z obsługą tych kolekcji zostały oddelegowane do osobnych klas: GameUsersHandler, ChessGamesHandler, WebSocketSessionsHandler. Zastosowanie tych kolekcji w postaci map oraz obiektów wykonujących na nich operacje dało możliwość wysyłania wiadomości do wsyztskich uizytkownikw w formie broadcastu, co jest zastosowane w momencie gdy ktos nowy dołącza 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 wysylania wiadomości w komunikacji jeden do jeden, kiedy jest porwadzona gra w parach.

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 obecny w katalogu **/src/main/webapp/assets/js/** w pliku **websocketClientEndpoint.js**

Kod z pliku:

/\*\*

\* WEBSOCKET CLIENT ENDPOINT

\*

\* websocket client variable name = WEBSOCKET\_CLIENT\_NAME

\*

\* GAME PIECE COLOR STATUGLOBAL VAR = SENDED\_CHESS\_MOVE\_STATUS

\*

\*/

***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 obsługiwane tu zdarzenia te co na serwerze: **onOpen**, **onMessage**, **onClose** i **onError**.

Stan gry i ruchy poszczególnych użytkowników są przekazywane między nimi za pomocą notacji fen. **Jest to tak zwana notacja Forsytha-Edwardsa.** Polega ona na zapisaniu położenia wszystkich figur I do kogo należy następny ruch w postaci ciągu znaków. Przykład notacji fen:

**„r1bqkbnr/pppp1ppp/2n5/1B2p3/4P3/5N2/PPPP1PPP/RNBQK2R”.** Jedną z większych trudności przy implementacji komunikatora do gry było zapewnienie parom użytkowników zamkniętej konwersacji. Zostało to umożliwone za pomocą implementacji protokołu komunikacji przy zdarzeniu onmessage, który to w zalezności od rodzaju wiadomości na podstawie instrukcji warunkowych wykonuje takie czynności bądź inne. Onjekt GameMessage ma właściwość private String type, dzięki której możliwe było zidentyfikowanie jakiego rodzaju wiadmosc jest wysylana aktualnie na serwer i co przy taim zachowaniu zrobic. Opróćz tego obiekt wiadmosc przechowuje informacje o tym kot wysyla i do kogo. Typy wiadomości zostaly zdefiniowane w kalsie GameMessageType przechowującej pola statyczne w postaci stringow 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 Mozę się zgodzić GAME\_HANDSHAKE\_AGREEMENT lub nie GAME\_HANDSHAKE\_REFUSE. Jeżeli jest zgoda drugiego użytkownika to obiekty tych użytkowników przyjmuja status is-playing. Statusy użytkowników SA reprezentowane w klasie GameUserCommunicationStatus równiez przez pola statyczne:

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 uzytkownik ma status WAIT\_FOR\_NEW\_GAME każdy może się z nim skontaktowac i on może z każdym. Gdy ma status IS\_DURING\_HANDSHAKE moze skontaktowa sie tylko z użytkownikiem, ktrego zaprosil do gry bądź zostal przez niego zaproszony, gdy ma status IS\_PLAYING, wysyłać wiadmosci mzoe tylko do uzytkonika, z którym gra.

Ruch szachowy to wiadomość typu CHESS\_MOVE, koniec gry to GAME\_OVER lub QUIT\_GAME, wtedy automatycznie użytkownicy zmieniaja status na WAIT\_FOR\_NEW\_GAME. USER\_CONNECT : gdy nowy uzytkownik przylacza sie do gry, wtedy do wszystkich wysylana jest aktualna lista użytkowników. Tak samo jest z USER\_DISCONNECT: uzytkownik zakonczyl polaczenie, do wszystkich wysylana jest aktualna lista użytkowników. TRY\_LATER gdy dany uzytkownik ma status IS\_DURING\_HANDSHAKE albo IS\_PLAYING, przy czymgdy uzytkownik jest w jednym ze stanow IS\_DURING\_HANDSHAKE lub IS\_PLAYING to dla pozostałych użytkowników poza swoją para do gry SA blokowane akcje umożliwiające kontakt z tym użytkownikiem.

Opis przebiegu gry:

Nowo polaczony użytkownik ma status WAIT\_FOR\_NEW\_GAME

Uzytkownik zaprasza drugiego użytkownika,

zmieniaja Satsua na IS\_DURING\_HANDSHAKE

jezeli się zgadza rozpoczynaja gre i zmieniaja status na IS\_PLAYING

graja

jeden z użytkowników wygrywa,

gra się konczy

użytkownicy znowu maja status WAIT\_FOR\_NEW\_GAME

jeżeli użytkownik się nie zgadza

użytkownicy znowu maja status WAIT\_FOR\_NEW\_GAME

1. Testy
2. Podsumowanie

Listeratura

Spis rysunków

Spis tabel

Załączniki