

Antti-Pekka Palomäki

Web Browser Based Online Chess, Human versus Human Games with Multiple End Point Devices

Helsinki Metropolia University of Applied Sciences

Master's Degree

Information Technology

Master's Thesis

17 April 2017

Author Title	Antti -Pekka Palomäki Web Browser Based Online Chess, Human versus Human Games with Multiple End Point Devices
Number of Pages Date	60 pages + 13 appendices 17 April 2017
Degree	Master of Engineering
Degree Programme	Information Technology
Instructor	Harri Airaksinen, Principle Lecturer
<p>The purpose of this study was to examine how it is possible to create multiuser web browser based real-time online games. In these games it is critical that all players can see the ongoing game situation at all times. In turn based games it is essential that actions take place in the correct order and in time critical games it is important that the latency of the actions is minimal in order to have a fluent game experience.</p> <p>Traditional HTML documents downloaded from an HTTP server do not directly provide communication methods with other server's clients even though all the files would be loaded from a single source. Communicating with an HTTP server using traditional HTTP methods do not produce information about the invoked actions for all participants.</p> <p>This study evaluates a few possible techniques to use in order to setup an online gaming environment. A lightweight proof of concept was made using the most promising techniques. Based on the received results AntsaChess system was built providing a real-time multiuser online platform for chess games.</p> <p>The paper describes what already available components and techniques were used, what components were created and how the communication took place between all components. A way of modelling a chess game is depicted from the object-oriented programming perspective.</p> <p>AntsaChess can be used by browsers supporting WebSocket and HTML 5 protocols. Based on manual and programmatic testing it can be said that the implemented concept seems good enough to be used on other web browser based games also. The actual robustness of the system will be found out only after the system receives tens or hundreds concurrent real users.</p>	
Keywords	Real-time online game, chess, WebSocket, Java, AngularJS

<p>Tekijä Työn nimi</p> <p>Sivumäärä Päivämäärä</p>	<p>Antti -Pekka Palomäki</p> <p>Web Browser Based Online Chess, Human Versus Human Games with Multiple End Point Devices</p> <p>60 sivua + 13 liitettä 17. huhtikuuta 2017</p>
Tutkinto	Master of Engineering
Koulutusohjelma	Information Technology
Työn ohjaaja	Harri Airaksinen, Yliopettaja
<p>Työn tarkoituksena oli tutkia kuinka voidaan tehdä reaaliaikaisia verkkoselainpohjaisia usean samanaikaisen käyttäjän pelejä. Näissä peleissä on kriittistä, että kaikki osapuolet näkevät aina vallitsevan pelitilanteen. Vuoropohjaisissa peleissä on tärkeää, että toimintoja voi suorittaa vain omalla vuorolla ja aikakriittisissä peleissä oleellista on toimintojen lyhyet vasteajat, jotta peli olisi jouhevaa.</p> <p>Perinteiset HTTP -palvelimelta ladatut HTML -dokumentit eivät suoraan anna mahdollisuutta kommunikoida toisten käyttäjien kanssa, vaikka kaikki ladatut tiedostot olisivatkin peräisin samasta yksittäisestä lähteestä. Perinteinen HTTP -metodien kautta kommunikointi HTTP – palvelimen kanssa ei suoraan tarjoa tarvittavaa tiedonvälitystä kaikkien osapuolten kanssa.</p> <p>Työssä on arvioitu muutamien mahdollisten tekniikoiden käyttämistä selainpohjaisten pelien toteuttamisvaihtoehdoiksi. Potentiaalisimmasta konseptista on tehty kevyt soveltuvuustesti, jonka pohjalta on toteutettu verkkoselaimille tarkoitettu shakkijärjestelmä. Järjestelmä mahdollistaa useiden samanaikaisten pelien pelaamisen ja katsomisen.</p> <p>Työssä on kuvattu mitä ja miten eri tekniikoita järjestelmässä on käytetty, minkälaisia komponentteja se sisältää sekä miten komponenttien välinen interaktio on toteutettu. Lisäksi on kuvattu eräs malli hahmottaa shakkipeli olio -ohjelmoinnin kautta.</p> <p>Lopputuloksena syntyi AntsaChess sivusto. Sen kautta ihmiset voivat pelata reaaliaikasta shakkia toisiaan vastaan verkkoselaimella, joka tukee WebSocket ja HTML 5 protokollia. Manuaalisen ja ohjelmallisen testaamisen perusteella voidaan sanoa, että konsepti vaikuttaa toimivan riittävällä tasolla ja sitä voitaisiin käyttää muissakin vastaavissa peleissä. Järjestelmän todellinen toimivuus selviää kuitenkin vasta siinä vaiheessa kun järjestelmää käyttää kymmenet tai sadat yhtäaikaista käyttäjä verkkoselaimella.</p>	
Avainsanat	Reaaliaikainen online peli, shakki, WebSocket, Java, AngularJS

Contents

Abstract

Table of Contents

1	Introduction	4
2	Research Question and Defining Requirements for System	6
3	Evaluating Suitable Technologies and Components	9
3.1	Possible Solutions to Implementation Technologies	10
3.2	Creating Proof of Concept with WebSockets	11
3.2.1	POC Architecture	12
3.2.2	Sending First Move	15
3.2.3	Conclusions on POC	18
4	Creating Client User Interface	19
4.1	Initial Thoughts and Remarks	19
4.2	Selecting Client Technologies	20
4.3	UI Areas and Functionalities	20
4.4	Login Functionality Turned into Practise	24
4.5	From Lobby Area to Table Area – Starting game	28
4.6	Under the Hood – Moving Piece	31
4.7	Security – Client Cannot Be Trusted	32
5	Communication between Domain Components	34
5.1	Web server as Intermediary Component	34
5.2	AntsaChessServer as Backend Component	37
6	AntsaChess Game with Object Oriented Style	40
6.1	Converting Physical Chess to Digital One	40
6.2	Processing Games	41
6.3	Table and Game Concept	43
7	Testing and Analysing AntsaChess	47
7.1	Testing Chess Moves	47
7.2	Message Delivery Times between Domain Systems	49
7.3	Performance Testing	50
8	Summary	59

References

61

Appendices

Appendix 1. Websocket.js

Appendix 2. First version of Lobby UI

Appendix 3. Start game with white pieces command peek

Appendix 4. Piece image URLs

Appendix 5. Chessboard.js JavaScript peek

Appendix 6. Example of Move -command sent from backend

Appendix 7. ChessRequest.java

Appendix 8. ChessMessage.java

Appendix 9. ChessMessageHeader.java

Appendix 10. ChessMessageToken.java

Appendix 11. Tomcat.java

Appendix 12. ChessServer.java code peek

Appendix 13. WebClientMessageHandler.java code peek

1 Introduction

One of the author's current dreams is to build his own web based system capable of providing different kinds of single and multiplayer online games. The games would be interesting and enjoyable enough to get the people playing frequently. Gradually this system would be used by enough people so that it would cover at least the maintenance costs for example by showing customized advertisement to the people.

Chess happens to be one of the author's favourite games, therefore, it was chosen to be the first game to get the dream started with. What also affects is that the author has some earlier experience of developing a plain chess in a smaller scope. Chess is a two player board game where the goal is to checkmate opponent. Checkmate is a situation in the board where opponent's king has no free squares to move and opponent's piece is threatening this king.

The modern chessboard consists of 64 squares (8x8) in two dissenting colours. In many cases the colours are white and black but they can vary. In the beginning of the game both players have 16 pieces in total. There are 6 different types of piece and they all move differently. In the beginning each player has one king, one queen, two rooks, two bishops, two knights and eight pawns.

Players start to move their pieces from opposite sides but during gameplay the situation changes a lot. The moves go in pairs, white (i.e. lighter colour) starts, following black's (i.e. darker colour) move. The game is finished immediately when a player is either checkmated, a draw has occurred or player voluntarily resigns. In timed games a player wins if an opponent time has run out regardless of the material situation. Usually the gameplay goes so that players try to eliminate each other's pieces and maybe promote a pawn to a queen at the other end of the starting point and make a checkmate. However, a game can end without eliminating any piece on the board. Timed games can enable a player victory without moving any piece.

Since the present study is the author's own project there are no business side specifications. It gives the freedom and responsibility to design and implement the system quite freely. However, that does not mean that plans could be changed significantly whenever problems occur. The long term objective guiding the designing of

the whole system is that a similar kind of concept could be used also in other web browser based games as well.

First, the paper discusses the evaluation of the possible technologies for system architecture and moves on to testing a promising concept. Second, the client user interface is introduced, followed by the description of the domain components. Third, the object oriented AntsaChess game is illustrated. Then, testing and analyzing the system is reported. In the end, a summary is provided.

2 Research Question and Defining Requirements for System

The purpose of this study is to find an answer to a question how it is possible to create web browser based multiuser online games. Chess is used as a case study. Requirements for the system are divided into two categories. The first category requirements are related to finding out appropriate technical requirements and the second are user experience related requirements. It must be noticed that if a feature works in a local environment that does not necessarily mean that it will work in a production environment since the system load can be significantly greater. The latter category is harder to measure since there are as many thoughts and ideas as there would be end users.

The user experience walks hand in hand with technical aspect. The initial thought about the user interface is that there must be minimal latency between sending the move to backend and updating the user interface.

End user multi-device paradigm

Before stating any requirements it is good to have an understanding of what is possible and what is not. Supporting all potential combinations of end user devices and operating systems is not reasonable. Building native software for laptops, desktops, mobile phones, PDAs and tablets would take an excessive amount of time and superior expertise resulting in maintenance problems while adding new features and fixing up bugs. Besides it cannot be guaranteed that even exactly similar devices behave similarly since there might be other running processes that take the processor's precedence and network latency varies between clients and server.

Setting the system requirements

The requirements must be set so that most users get a fluent gaming experience. It involves supported devices and servers being able to meet the requirements. An example of too detailed a technical requirement is that a user interface needs to be updated within 500 ms after the server has sent an update command. This kind of requirement cannot be guaranteed to be met at all times.

The first and most critical requirement is to find platform components that the system can be built on. Too detailed requirements towards technologies should be avoided because technologies only enable game play and can be replaced with better ones.

The second critical requirement is to find the techniques that can be used to ensure dynamic gameplay. After finding solutions to the first and second requirements comes the requirements towards the game itself. Those can be divided to must have and nice to have features. The must have features are being able to simultaneously play chess games against other humans with different kinds of devices. Minimum is one simultaneous game for each user utilizing the system. The system must also indicate when the game has ended and enable rematches against the same opponent. Another must have requirement is to be able to watch ongoing games played by others and communicate textually at the same time.

Table 1 summarizes the requirements for the system that should result in a good user experience.

Table 1. The initial guideline for the system.

Requirement	Priority	Results to	If requirement not met
Finding platform components.	Critical	Enables to start building the system.	System components cannot communicate. No further steps can be taken.
Finding / developing architecture for dynamic system.	Critical	Enables to start developing the system.	Without proper architecture the system will not be usable.
Being able to play human vs. human games with different end devices.	Critical	Functional chess games.	Study has not provided a platform.
Watching ongoing games and communicating textually.	High	Enables building a community.	Difficult to form a community.
Creating a nickname.	Medium	People are able to know their opponent. Helps building a community.	Enables reusing same nickname (without a password).
Measuring user experience.	Medium	Getting improvement ideas and feelings.	Rough errors can be left undetected.
User statistics	Low	Information about played games.	Users can't view ranking nor history of own games.
Localisation	Low	Getting the service with a custom language.	English is the main language.

Table 1. Requirements, priorities and results.

Requirements set in Table 1 is the initial guideline for the system. The consequences for requirements that are not met are also shown in Table 1.

3 Evaluating Suitable Technologies and Components

Some technical ideas about the system at large must exist before starting. Standards and utility frameworks help software developers to choose suitable technologies. This chapter describes initial ideas towards technologies.

Because network connection is anyhow required to play and many end device has a web browser natively it is natural to start exploring options from HTML / XHTML over HTTP. A possible solution is depicted in Figure 1.

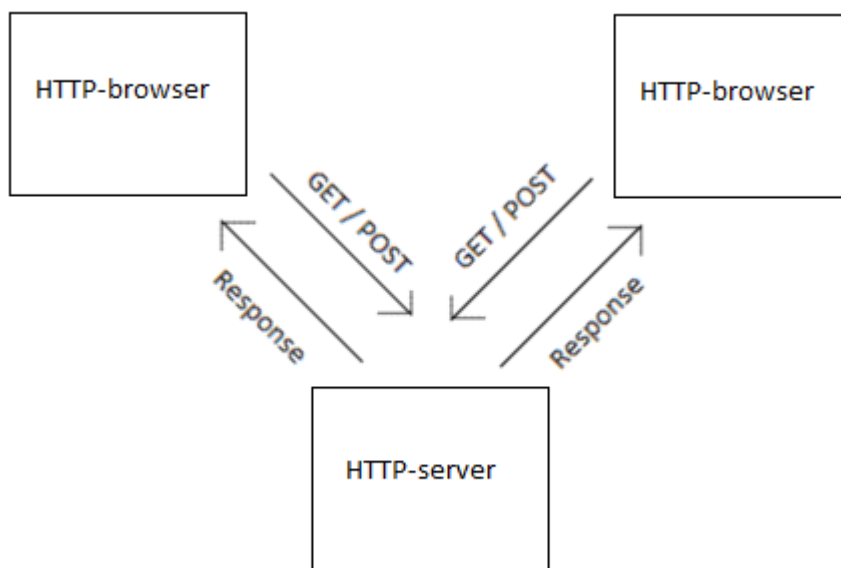


Figure 1. A typical HTTP session (Mozilla Developer Network, A Typical HTTP session: 2017).

Usually browser issues an HTTP GET request to receive data from server and sends user data to the server with POST request. In both cases browser makes the initiative and server responds accordingly. HTTP 1.1 is stateless by nature (Network Working Group: RFC 2616 HTTP 1.1) which nowadays is often used as a transfer protocol. Stateless means that a web server does not keep track which client sends the message although there are means to track the clients if needed.

This being the nature of HTTP the problem is that how several users get a chess game played when the browsers do not know about each other. A crucial part of the game is that people in a current chess game see the same representation of the board all the time.

3.1 Possible Solutions to Implementation Technologies

Three different possible solutions as to the implementation technologies are evaluated in this chapter: peer-to-peer communication, HTTP-polling and WebSockets. They are introduced below.

Peer -to -peer communication

It could be possible that browsers directly communicate with each other using for example WebRTC data channel (Mozilla Developer network: WebRTC data channels). This technology is still experimental and is suspicious security wise. In addition, this technology would not allow other users to watch the games. Collecting reliable statistics would be difficult in a situation where communication happens straight between end devices browsers. That is why this option is ruled out.

HTTP -polling

Polling in this scenario means that while player A is in turn, the player B periodically sends a request to web server and asks if there is any new data available for the current game session. If player A has made a move then as an answer player B would get a response for instance “update E2 -E4 and make your own move” with the next polling request.

A polling kind of system could be done with JavaScript and the system would work also in cases where either of the players closes the browser because that can be detected. The weaknesses of this kind of architecture are the playability of the game and the overhead of unnecessary HTTP traffic to a web server. Polling time would be very difficult to optimize for every game. Polling could create unnecessary load to the web server or the players might not get an immediate update even though the data would be already available.

Considering a game where player A is in turn and thinks for five minutes for the next move. If the polling time is set to one second then player B and possible spectators send 300 request each ending up in 600 unnecessary requests to web server. Alternatively, if

the polling time was 30 seconds then only 10 redundant request would be sent by each user around the table. HTTP polling can be ruled out because of the nature of chess.

WebSockets

“The WebSocket Protocol enables two-way communication between a client running untrusted code in a controlled environment to a remote host that has opted-in to communications from that code.” (Internet Engineering Task Force: The WebSocket protocol). In other words, the client and a server can interact with each other by sending and receiving data at the same time.

Java Specification Request 356 (JSR 356) is provided to support WebSocket protocol for Java Developers (Johan Vos: Java API for WebSocket). Idea is to upgrade an HTTP connection to a full duplex WebSocket. After successful upgrade the communication is bi-directional by nature meaning that also web server can push messages directly to connected browsers.

3.2 Creating Proof of Concept with WebSockets

This chapter delves into initial thoughts about the system architecture by creating a proof of concept. In general, a proof of concept means to test an idea under the surface not going too deep into details (Proof of Concept (POC): 2017). The idea of this proof of concept is to test the initial system architecture. Considering a game session where player A and player B have started a chess game and a few spectators have joined to watch the game. Player A with light colour pieces tries to move from square E2 to square E4. Before updating the chessboards the suggested move must be validated. Commands that come from the user interface cannot be automatically taken as valid move since the UI can be tampered with different tools. If the move is valid according to chess rules all participants including the original sender must be notified to update the UI. If the suggested move is invalid then only the original sender needs to receive information about what to do next.

3.2.1 POC Architecture

The idea of the POC is to test a very light system to see if a browser can send a chess move to another browser via web server and chess server. Some configuration and coding is required in order to enable the communication between browsers.

Figure 2 shows system components in the proof of concept.

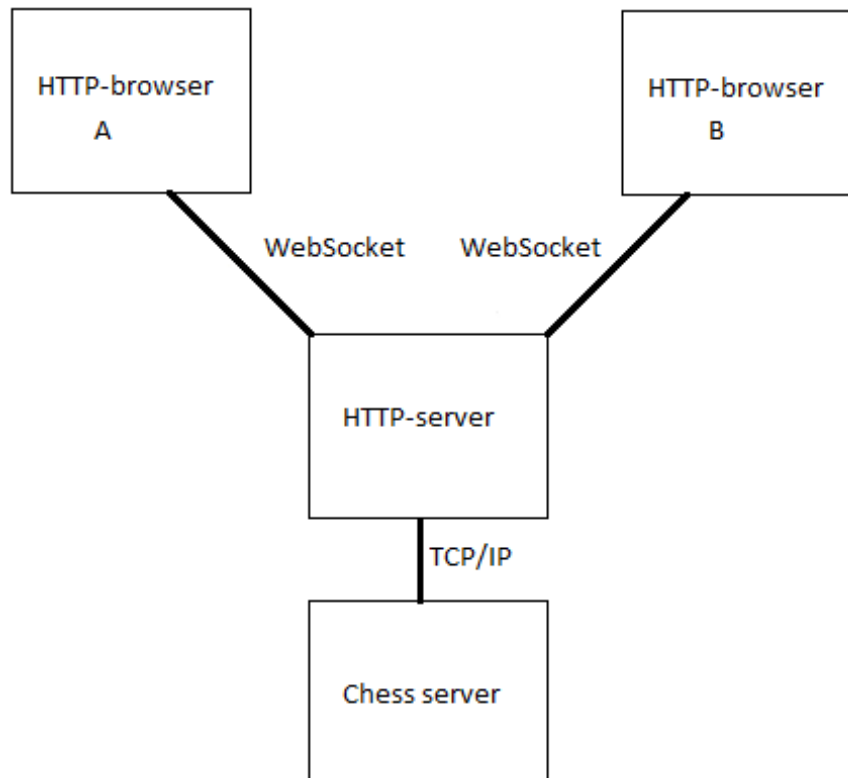


Figure 2. Architecture used in POC.

All components in Figure 2 reside in a single laptop computer so there is no actual network between components.

Coming from Java programming language background the first thing needed is to find an available web server which supports JSR 356. Reviewing Apache Tomcat specifications it was found that from version 8 WebSockets are supported (The Apache Software Foundation: Apache Tomcat 8). Also Jetty, which is a web server and javax.servlet container, supports WebSockets from version 9 (Eclipse: Jetty). Having a little earlier experience of Tomcat it is chosen to be the web server. Tomcat's function in this POC is to handle incoming and outgoing web traffic from the browsers and

communicate with very lightweight Java ChessServer using socket technology. A socket is an endpoint for communication.

Java provides options to use a blocking and non -blocking server. A blocking server, which in practice means Java -ServerSocket, seems like the appropriate choice. The web server and ChessServer communicate via TCP/IP and exchange information with direct Java classes. During Tomcat's startup a connection to a port that ChessServer listens to is made. The POC environment information is shown in Table 2.

Table 2. POC environment.

Test Equipment	Intel® Core™ i5-3230M CPU @ 2.60GHz, 6GB RAM, Windows 8.1 (64 bit)
Web server	Tomcat 8.0
Tester page	http://localhost:8080/AntsaChessWeb/faces/pages/poc/poc.xhtml Partly generated with JSF 2.2 (MyFaces)
ChessServer	AntsaChessServer 0.0.1 pass-through phase
Development platform (IDE)	Eclipse Luna

The minimal tester page represents the client UI from which is possible to simulate a situation where user sends a chess move to the Tomcat web server. Figure 3 shows the tester page layout and network traffic involved viewed from Firefox browser UI developer tools perspective.

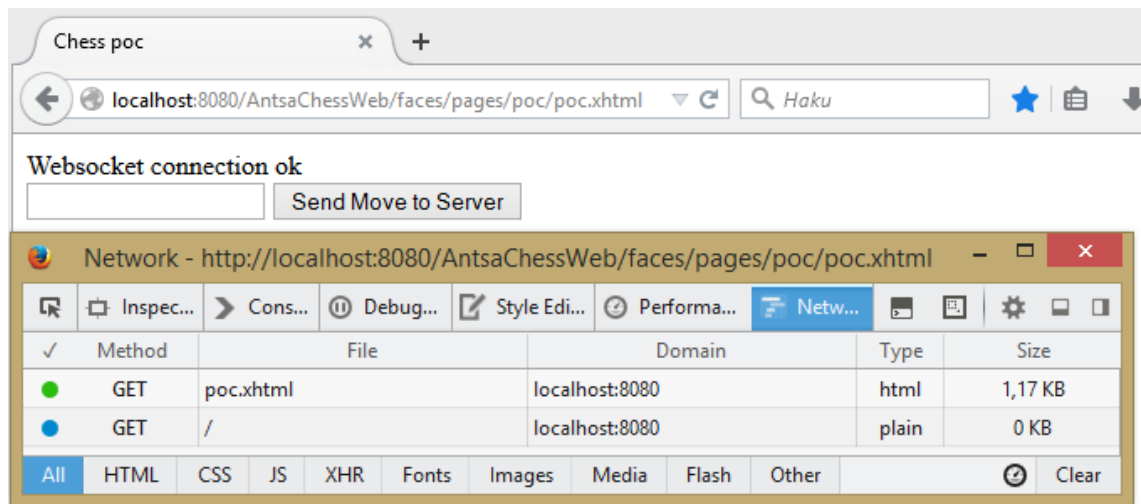


Figure 3. View from Firefox browser.

After examining the request and response headers it can be noticed that there is an extra GET initiated by JavaScript from tester page. Header section from developer tools in Figure 4 verifies that the connection is indeed upgraded to a full duplex WebSocket.

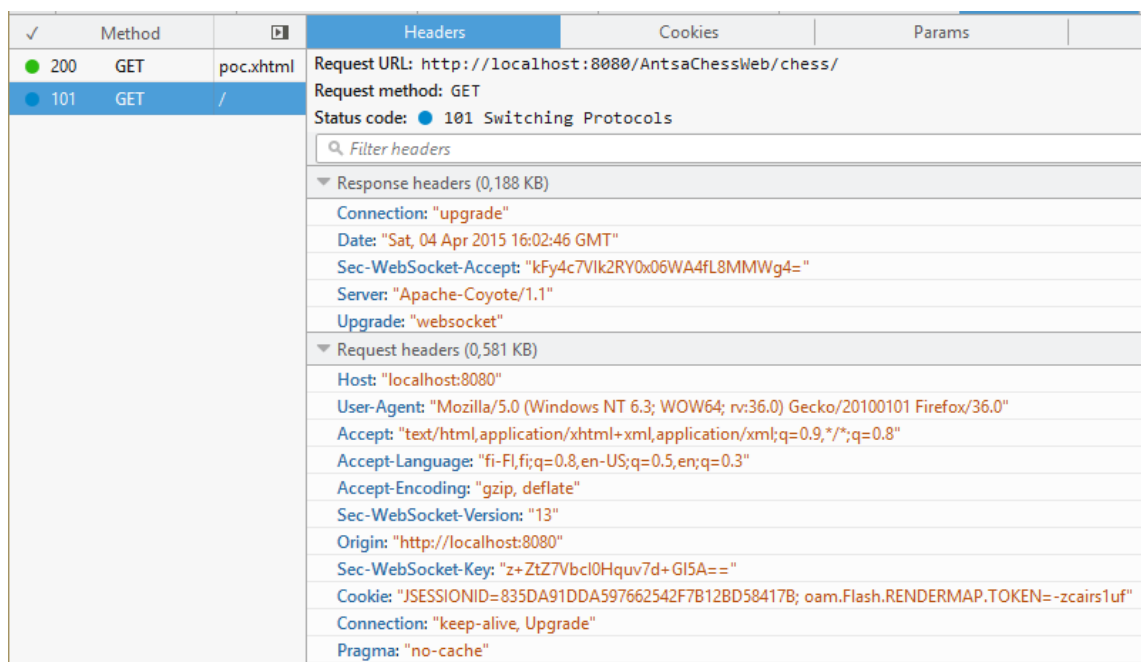


Figure 4. Browser request and web server response headers.

Tester page's HTML structure and JavaScript can be viewed in Figure 5.


```

<h:body>
  <div id="messages">

  </div>
  <div>
    <input type="text" id="move" />
    <input type="button" id="sendButton" name="sendButton"
      value="Send Move to Server"
      onclick="sendToWs()"
    />
  </div>
  <script type="text/javascript">
    var websocket = new WebSocket('ws://89.27.2.232:8080/AntsaChessWeb/chess/');
    function onOpen(event) {
      document.getElementById('messages').innerHTML = 'Websocket connection ok'
    }
    function onMessage(event) {
      document.getElementById('messages').innerHTML += "<br />" + event.data;
    }
    function sendToWs() {
      var text = document.getElementById("move").value;
      websocket.send(text);
      return false;
    }
  </script>

```

Figure 5. Contents of poc.xhtml.

The <h:body> tag comes from JSF -namespace

<html xmlns:h="http://java.sun.com/jsf/html">. JSF means JavaServer Faces which is utility technology for generating server side web pages. Browser tries to make a WebSocket connection to hardcoded IP -address which points to localhost. Browsers that do not have JavaScript enabled are not supported. This is a definition of policy for the whole AntsaChess concept made during POC.

On Tomcat side /AntsaChessWeb -context root has been configured to detect WebSocket connections for URL pattern /chess/. Web server's Java -class has been annotated with @javax.websocket.server.ServerEndpoint(value = "/chess/") similarly to Oracle's WebSocket programming snippets (Johan Vos: Annotation-Driven Approach).

3.2.2 Sending First Move

Moving a piece from E7 to E5 could be represented in AntsaChess as "4143" as shown in Figure 6.

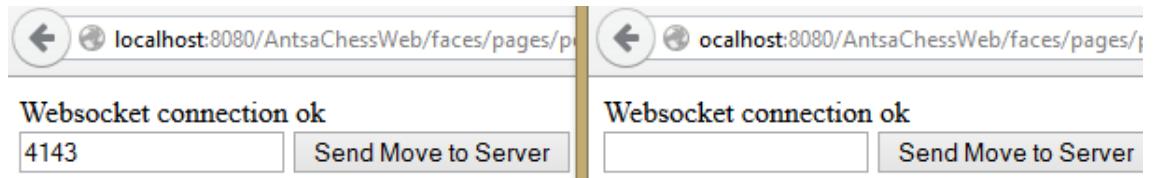


Figure 6. Two browser side by side where leftmost browser is about to send a move.

In the POC phase it is enough to use loggers at the servers' side to verify that the message is really going all the way through to ChessServer. The browser sends a text "4143" to web server by pressing the "Send Move to Server" -button. After receiving the "4143" text the web server creates a new Java object called Move which internally contains two Square objects for storing from and to positions. The square object contains row and column information where the information is put. The web server sends the constructed Move object to ChessServer using a TCP/IP connection created in web server startup.

At this point ChessServer only returns the same Move object back to web server which creates a textual representation from the Move object and sends it to both browsers. The tester page's JavaScript function "onMessage" is called. Outcome is shown in Figure 7 after both browsers have received the message.

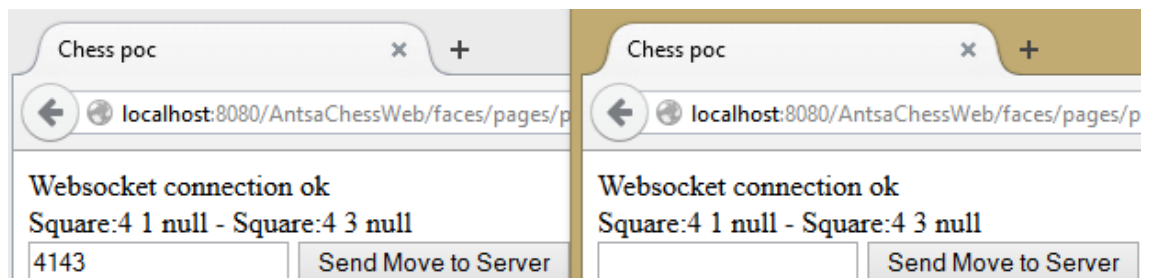


Figure 7. Web server response has been received.

The null word comes from the Move object's textual representation which could also contain information about a piece in a square.

Figure 8 shows that the rightmost Firefox instance is also able to send move.

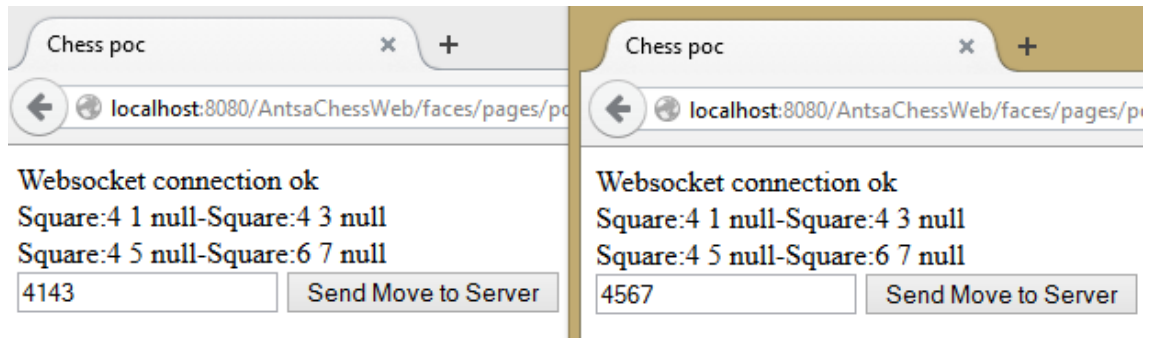


Figure 8. Rightmost browser has sent a message.

A final test for the concept is made by using 5 different clients at the same time. The devices and outcome is presented in Table 3.

Table 3. Browser test results.

On device	Browser	Result
localhost	Firefox 36.0.4	Could receive and send messages.
localhost	Chrome 41.0.2272.118	Could receive and send messages.
localhost	Internet Explorer 11.0.9600.17690	Could receive and send messages.
iPhone 5 - Mobile	Safari (inbuilt)	Could receive and send messages.
Samsung Galaxy Mini (2011) - mobile	Inbuilt - custom	Fail. Test page opened but WebSocket connection was not established.

The concept is not working with year 2011 Samsung Galaxy Mini which is a minor drawback. The web server never got WebSocket contact from Galaxy Mini. However, several mobile browsers support WebSockets and this gives a reason to continue with the concept (caniuse: 2015)

3.2.3 Conclusions on POC

The messages were able to traverse from frontend to backend and back. First a browser sent a textual representation of a move to a Tomcat web server which converted the message into Java classes. The message was sent to ChessServer and the same information was sent back to web server. Both browsers received a textual representation of the move and updated the user interface.

All browsers will not work with the WebSocket concept. JSF did not give any added value for generating the UI layout. Plain HTML and vanilla JavaScript seems to be enough. Visually UI updates happened in a blink of an eye like there would not be any delay.

The Tomcat web server functioned fine as an intermediary component between end users and AntsaChessServer. TCP/IP with Java sockets used by `java.io.ObjectOutputStream` and `java.io.ObjectInputStream` worked fine thus there was no need for higher level protocols in between web server and ChessServer. POC solved the first two critical requirements by finding platform components and architecture to use later on.

4 Creating Client User Interface

Client is anything that can communicate with the underlying backend system. In AntsaChess the client UI is built for web browsers but it would be possible to replace them. Creating a custom native client could be an alternative for browsers. For example, a client made with Java could bypass the web server part and communicate directly with ChessServer using sockets if they would be supported.

The last unsolved critical requirement for the system in Table 1 is “Being able to play human vs. human games with multiple devices”. That is a massive requirement which can be split into subcategories within UI, web server and ChessServer. This chapter sheds light on the UI and its communication with the web server. The backend ChessServer can be thought of as a black box from UI perspective.

4.1 Initial Thoughts and Remarks

Mobile first is a good way to start planning when multiple browser needs to be supported. Basically it means that it is good to start thinking about the content for the smallest supported display. However, that does not mean that UI testing should be done with a real mobile phone all the time since it's quite slow.

Google Chrome provides emulators for devices that can be effectively used even though the result might look a little different in a real device. It should be decided first if the layout is allowed to look distinct in different devices. If the same layout must be used with all the devices then scaling needs to be done. Eventually the most important thing is that UI is functional.

The challenge to start a chess game can be overcome by creating a screen where all online users are seen and where game preparing actions can happen. Derived from the mobile first approach the game part should be another screen to get sufficient space for the chessboard. With big displays there is enough space to place everything neatly but with mobile phones the display size is very limited and in order for the UI to be practical it should be possible for the chess pieces to be moved by a mouse or a finger. Chessboard consist of 8 x 8 squares and the whole board should be seen at one glance. For example the Apple iPhone 5 screen dimensions are 123.8 mm x 58.6 mm (Apple

iPhone5: 2017). Showing the whole board in a square shape and moving pieces is quite an equation. Luckily many mobiles have zoom capabilities.

4.2 Selecting Client Technologies

Within WebSockets it is possible to send binary and textual data. There is no mandatory reason to choose over another but for browser debugging purposes textual data in JSON format is chosen. JSON is text in JavaScript Object Notation format (W3Schools, JSON - Introduction). Many browsers support natively JSON to form objects and converting them back to strings of text. Moreover it is possible to form hierarchical and readable data. Messages to web server is sent with partly hardcoded JSON with dynamic population of objects. Web server in turn converts JSON -text into `org.json.JSONObject` objects.

A noticeable problem raised quite early at development phase from the UI side. WebSocket connection variable is created in global namespace but transferring control to another HTML page causes a page load which results to lost connection. In order to use a single WebSocket connection and maintain prevalent states all the actions should occur from a single page.

Nonetheless, for the sake of clarity and maintainability it is better to divide the application into logical HTML sections. AngularJS 1 helps developers to create single page applications (SPA) but still being able to divide application into logical html parts. AngularJS 1 is dependent on jQuery and works with several browsers. Furthermore, Angular team run their test suite against Safari, Safari (IOS), Firefox, Chrome and IE ensuring the main browsers to work (AngularJS: FAQ). JQuery itself is a JavaScript library for manipulating HTML documents (jQuery: What is jQuery?). The technologies mentioned here got a green light to continue.

4.3 UI Areas and Functionalities

Some of the functionalities can be directly derived from the requirement specifications. For instance watching the game turns into a Watch table function and creating a nickname turns into Login function. The UI has been divided into three logical areas called USER, LOBBY and TABLE. Each area is handled accordingly by its own

AngularJS controller. The files named `UserController.js`, `lobbyController.js` and `tableController.js` take care of the states of the sections regarding visibility, actions and bi-directional network traffic in Angular scope. Controller files are able to handle messages from web server even though the area they control might be hidden in UI. Later on this section it is shown how the areas look like.

Altogether there are 12 different possible functionalities available to invoke from UI. Web server always sets the area into responses where the message is intended but it is good to underline that not all messages UI receives are responses. Some of the messages are caused by other users and some by the system. Table 4 shows the functionalities in the USER area.

Table 4. Functionalities that can be invoked from the USER area.

Functionality	Area	Parameters needed	Web server primary response(s)	Result in UI
Login	USER	Nickname	Info about created tables and who is online. Info about who logged in.	Newly logged in user gets a private key, sees tables and who are online. Others add new nickname.
Logout	USER	-	Info about who logged out.	Who logged out reloads the page causing disconnection from server. Others remove the nickname.

Pressing a “logout” -button in the USER area, closing a browser or navigating to another page invokes the Logout functionality.

Table 5 shows the functionalities in the LOBBY area.

Table 5. Functionalities that can be invoked from the LOBBY area.

Functionality	Area	Parameters needed	Web server primary response(s)	Result in UI
Create new table	LOBBY	Private token, nickname, selected game time item from the list	Command to create new table with given parameters.	Everybody adds created table to list. Different controls are seen amongst users.
Remove table	LOBBY	Private token	Command to remove table by id.	Everybody removes table from list.
Join table	LOBBY	Private token, table id	Start game and update table status.	Table creator and joiner opens table with chessboard. Others update status of the joined table.
Watch table	LOBBY	Private token, table id	Parameters to create chessboard from current situation.	Selected table is opened.

Functionalities in the LOBBY area provide means for handling the tables. The LOBBY area also shows the currently logged in players.

Table 6 shows the functionalities in the TABLE area.

Table 6. Functionalities that can be invoked from the TABLE area.

Functionality	Area	Parameters needed	Web server primary response(s)	Result in UI
Move piece	TABLE	Private token, from square, to square	Command to move accordingly.	Everybody around the table updates board. Turn changes.
New game	TABLE	Private token, tableId	Command suggesting a new game.	Possible new game.
Resign	TABLE	Private token, tableId	Resigned nickname	Game ends. Users around table sees who won.
Offer draw	TABLE	Private token, tableId	Info about draw if both players invoke functionality.	Game ends. Everybody around table sees info about draw by agreement.
Send message	TABLE	Private token, tableId, text message	Info about chat message	Everybody around table adds new to chat part.
Return to lobby	TABLE	Private token	Command to remove table if player returned.	Returner sees lobby. Others around table sees update in chat if returned user was player.

With TABLE area functionalities it is possible to manage the game and table.

4.4 Login Functionality Turned into Practise

There must be an agreement regarding message exchange between the UI and the web server. This agreement makes them understand each other during message exchange. All UI files are loaded when browser makes HTTP -GET to web server's URL /AntsaChessWeb/antsaChess.html. At the same time WebSocket connection is negotiated.

Figure 9 visualizes the USER area which either shows login or logout possibilities.

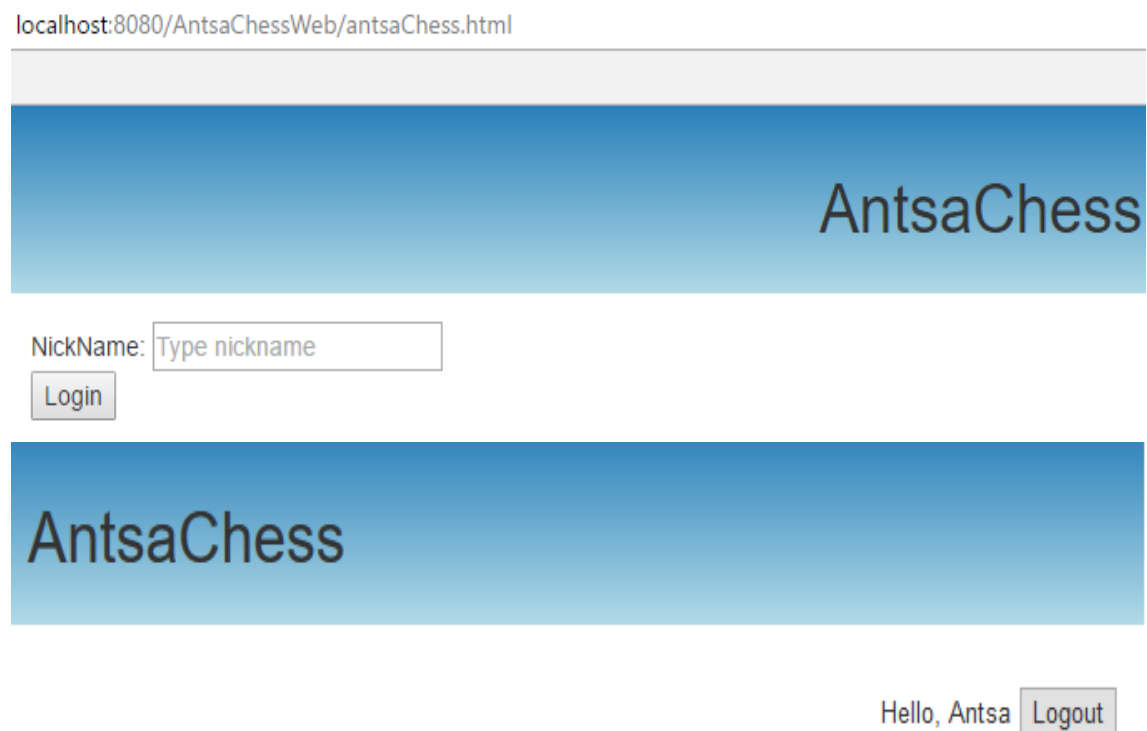


Figure 9. The USER area.

The upper part in Figure 9 is a view for a client which has not logged in yet. The lower part is seen after successful login.

Figure 10 shows all the files loaded and needed to construct the UI when user enters AntsaChess for the first time.




















Name	Method	Status	Type	Initiator	Size
 chess/	GET	101	websocket	websocket.js:2	0 B
 antsaChess.html	GET	200	document	Other	2.8 KB
 websocket.js	GET	200	script	antsaChess.html	1.5 KB
 chessboard.js	GET	200	script	antsaChess.html	7.4 KB
 jquery-3.1.1.min.js	GET	200	script	antsaChess.html	84.9 KB
 jquery-ui.js	GET	200	script	antsaChess.html	509 KB
 angular.min.js	GET	200	script	antsaChess.html	159 KB
 app.js	GET	200	script	antsaChess.html	278 B
 chessFactory.js	GET	200	script	antsaChess.html	1.7 KB
 tableFactory.js	GET	200	script	antsaChess.html	1.6 KB
 mainController.js	GET	200	script	antsaChess.html	784 B
 lobbyController.js	GET	200	script	antsaChess.html	4.3 KB
 userController.js	GET	200	script	antsaChess.html	1.3 KB
 tableController.js	GET	200	script	antsaChess.html	21.6 KB
 bootstrap.min.js	GET	200	script	antsaChess.html	36.2 KB
 bootstrap.min.css	GET	200	stylesheet	antsaChess.html	119 KB
 antsaChess.css	GET	200	stylesheet	antsaChess.html	6.2 KB
 table.html	GET	200	xhr	angular.min.js:104	4.5 KB
 lobby.html	GET	200	xhr	angular.min.js:104	2.6 KB
19 requests 964 KB transferred Finish: 518 ms DOMContentLoaded: 404 ms Load: 517 ms					

Figure 10. UI is created from shown files.

The UI consist of JavaScript, HTML and CSS files. Bootstrap files are used for making the site responsive in cooperation with jQuery files. In addition, jQuery is used for dynamic manipulation of HTML Document Object Model. Specific AngularJS 1.5.9 version is used for helping to handle the single page application actions and UI rendering. File app.js is used for defining the whole antsaChessApp AngularJS module.

After pressing Login -button (see Figure 11) the control flows to websocket.js which handles all traffic in WebSocket and runs outside of AngularJS scope. Appendix 1 shows the whole JavaScript that a browser uses with WebSocket.

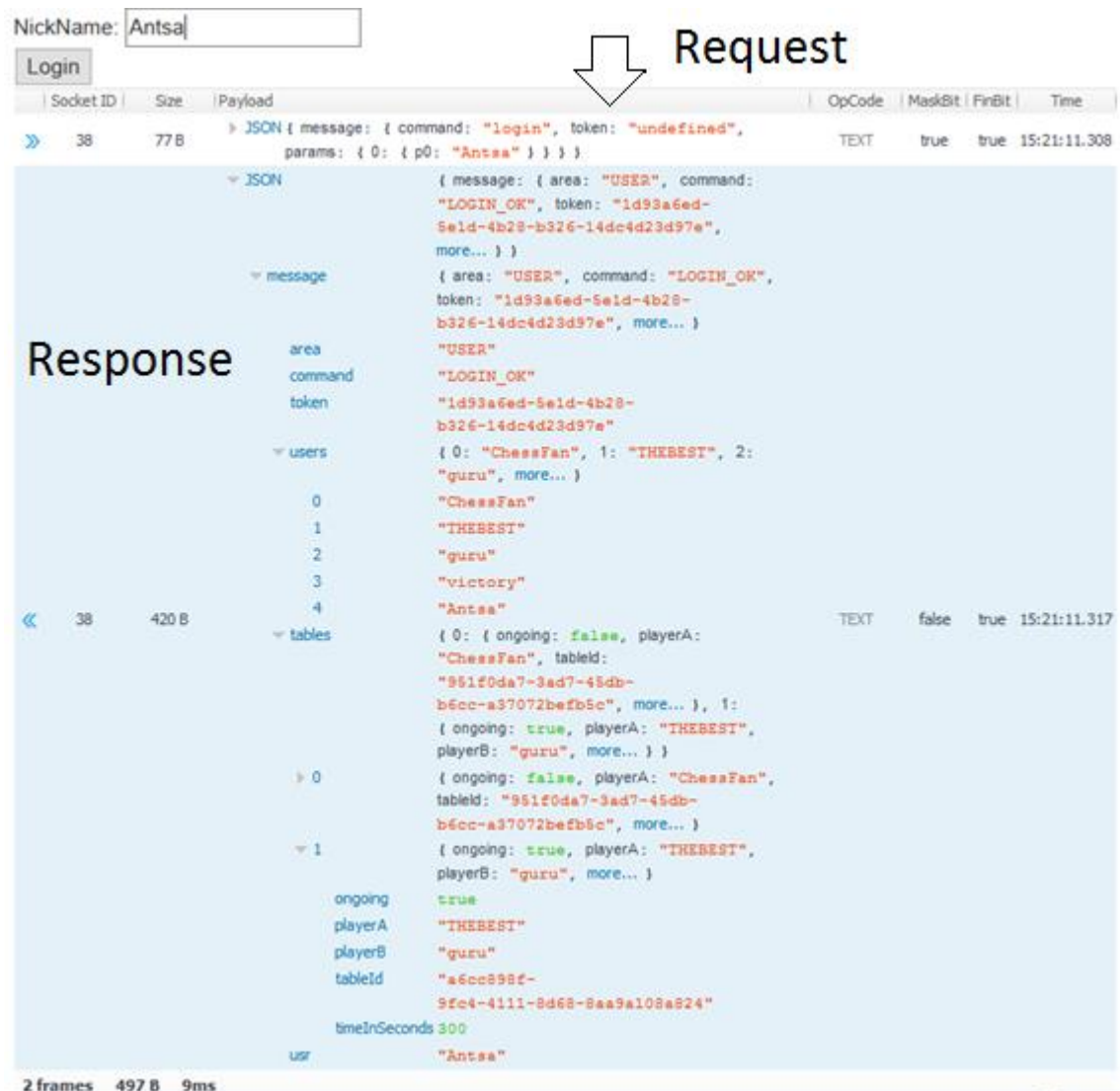


Figure 11. A login functionality's request and response in JSON format.

When looking at Figure 11 it can be detected that behind the scenes browser sends first a 77 bytes frame which contains data

```
{message: {command: "login", token: "undefined", params: {0: {p0: "Antsa"}}}}
```

in a JSON -format. The web server interprets from the command part that a user wants to login and reads the nickname from params part. Private token is not known at this point and "undefined" is sent because of common data handling. The response frame in Figure 11 is 420 bytes and tells that the message is intended for the USER area, the

login was ok with nick “Antsa” and the private token for session is 1d93a6ed-5e1d-4b28-b326-14dc4d23d97e.

The USER area is handled by `UserController.js` which populates `chessFactory.js` and emits the information forward. The `chessFactory.js` holds the private token for the whole session. The token is universally unique identifier representing 128 -bit value. In Java Platform, Standard Edition 8 the development kit provides this identifier in form of `java.util.UUID` class (Oracle: Class UUID). The web server generates an UUID per WebSocket session and passes it to ChessServer and UI.

Figure 11 also shows the users array which tells all online nicknames and the tables array contains information about the existing tables. Lastly the `usr` field tells the user’s own accepted nickname which is limited to 9 characters.

Figure 12 shows an example of how a browser renders LOBBY area after populating pertinent JavaScript files from web server’s response.

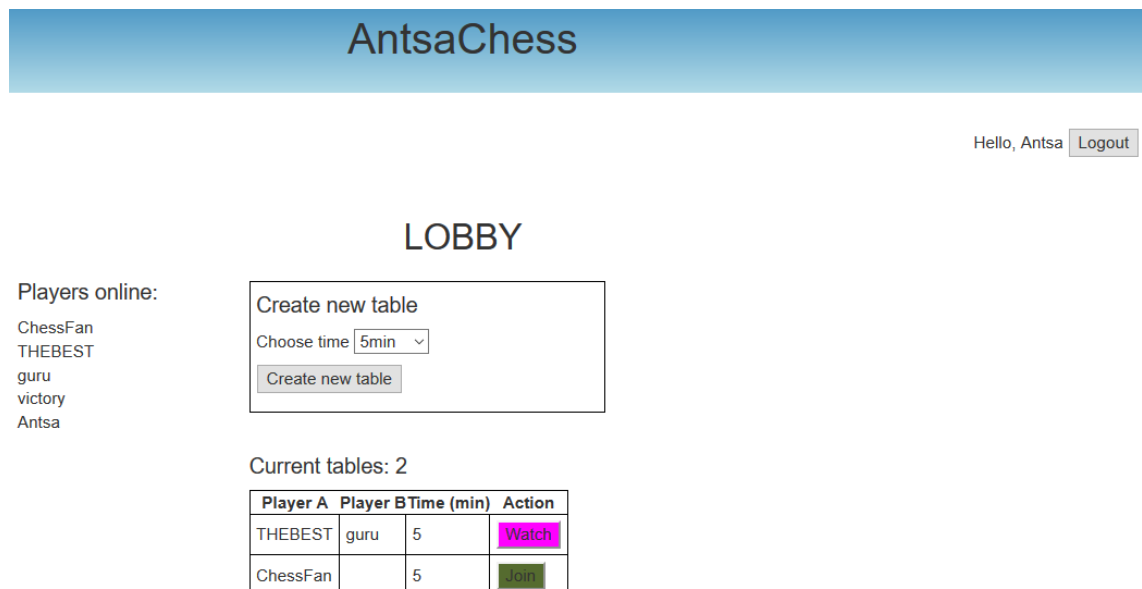


Figure 12. Populated LOBBY area after login.

An early version of the LOBBY can be seen in Appendix 2.

4.5 From Lobby Area to Table Area – Starting game

The user can create his/her own table or join in an existing one. The system gives white pieces for the user who created the table and for joiner is given black pieces. The following kind of JSON is sent to web server when Join table functionality is invoked.

```
{message: {command: "join_table", token: "8cfa4c8a-1d2d-4d42-a706-43072ff223ce",
params: {0: {p0: "4bce2941-e24b-4ffb-86e9-df990d0056c5"}}}}
```

Parameter p0 is the table id and token is the joiner's private key.

It might be possible that several players would try to join the table at exactly the same time causing a race condition. ChessServer synchronizes the joining by ending up in a situation where only the first one is attached to table and the rest are ignored.

With little UI manipulation it is possible to send the "join_table" command even when the "Join-button" is not visible.

After a successful join the web server sends information for both players about the created table regarding both players' nickname, game time and table id. The board position is also sent covering every square with piece information and available moves. The starting command is different amongst players. Appendix 3 shows the JSON which contains the information needed to start the game with white pieces (all available moves are not included). The chessboard is dynamically populated from initial parameters received from the web server. The chessboard is boosted with CSS and built with HTML `<table>` element along with `<thead>` `<tr>` and `<td>` elements. Event handling is done in chessboard.js where for example the clicks are verified against available moves received from the web server. Appendix 5 shows details of JavaScript used for handling chessboard events. Figure 13 shows the view from the white pieces' perspective after getting the command `START_GAME_AS_WHITE`. A few chat messages has been also exchanged.

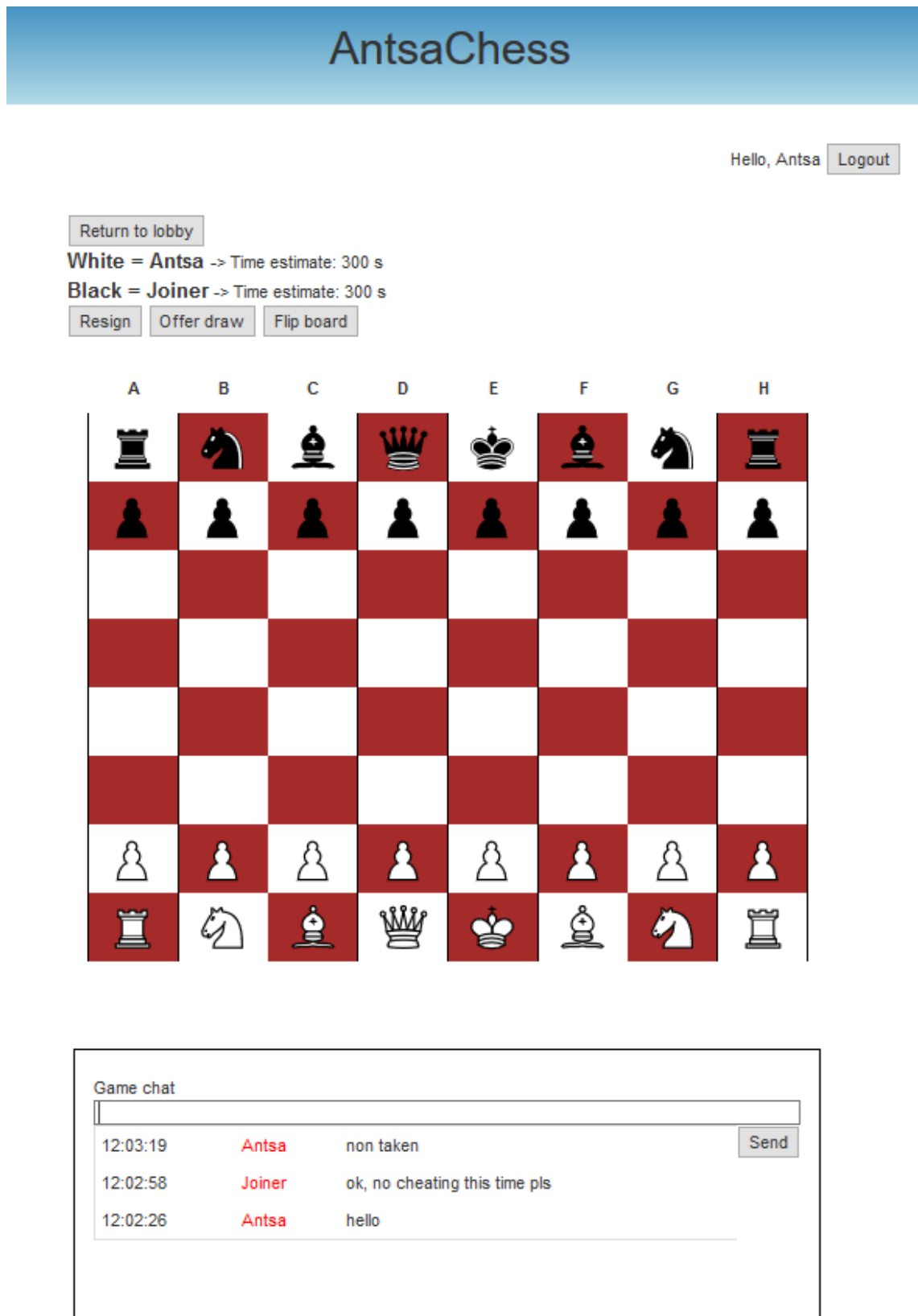


Figure 13. Start view from white pieces player perspective.

The HTML structure is the same for both players but on black pieces side the board and pieces have been rotated 180 degrees using CSS.

Figure 14 reveals two hidden bishops. Each row's last column contains these images.

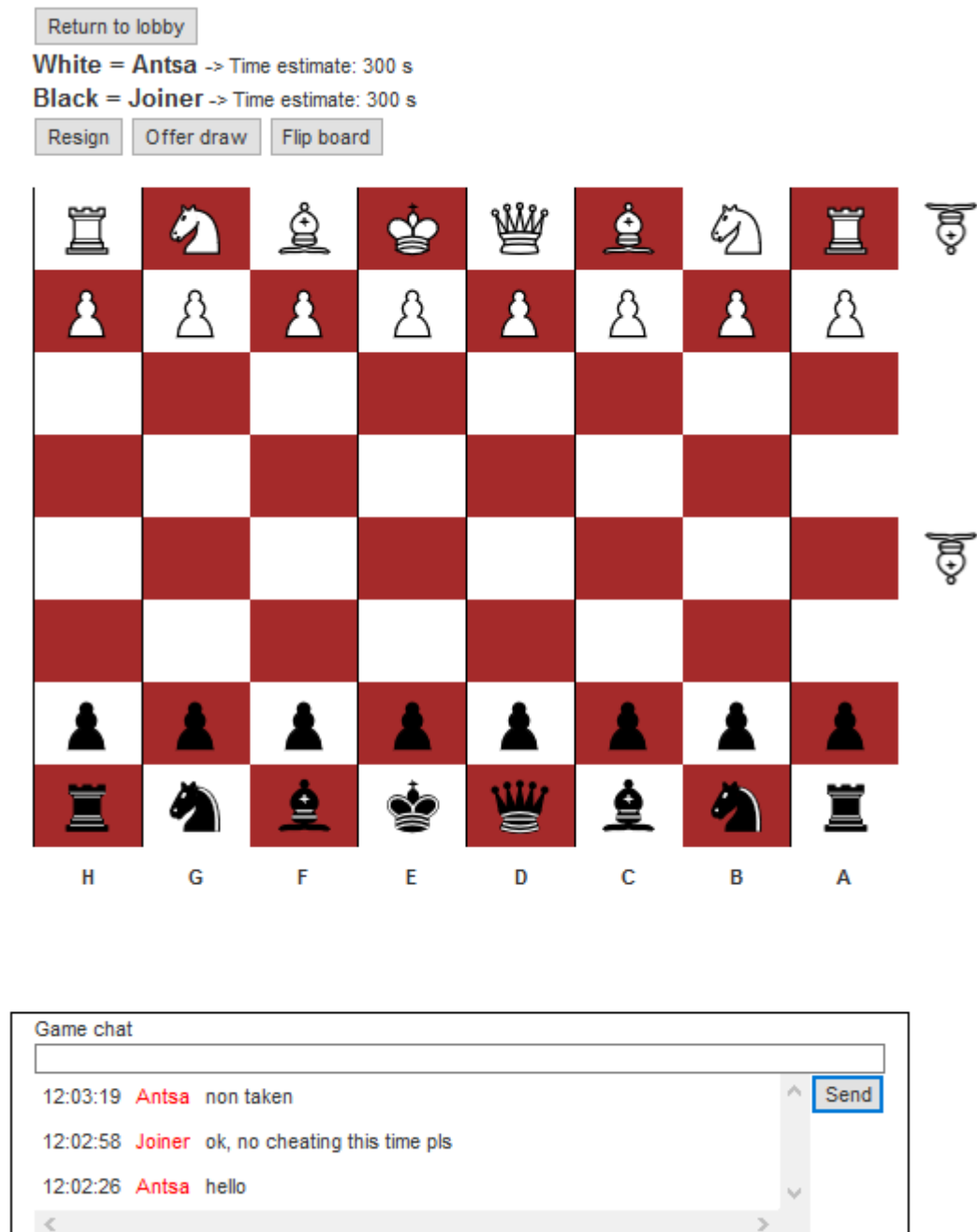


Figure 14. Layout from black pieces perspective with 2 hidden bishops revealed.

Figure 14 reveals two out of eight hidden bishops in the HTML structure. Adding a hidden image to every <tr> keeps the rows at equal height when there is no piece on the row.

All piece images are retrieved directly from Wikipedia and they are free to use according to Wikipedia (Wikipedia: Image use policy). The images are scaled with Bootstrap -library to fit in <td>. All used piece images URLs can be found in Appendix 4.

4.6 Under the Hood – Moving Piece

In all its simplicity a move is done by sending info to backend of from which square a piece moves to which square. Backend validates the move and sends the move to players and spectators where the UI is responsible for visualizing the action. The raw HTML of a <td> -square can be seen looking at Figure 15.

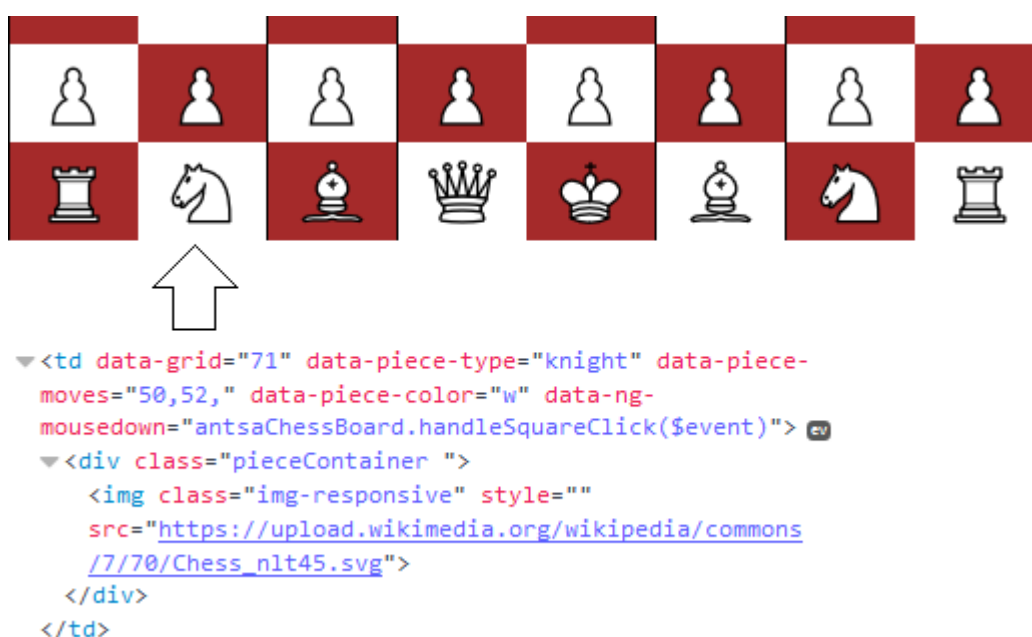


Figure 15. Raw HTML behind a knight image.

Examining more closely the HTML part in Figure 15 it can be detected that <td> contains several data-attributes. Attribute data-grid tells position in the board, attribute data-piece-moves tells available squares from this square, attribute data-piece-color is “w” for white, “b” for black and attribute data-ng-mousedown is AngularJS directive for mouse down event.

Behind the data-ng-mousedown function is the UI’s custom tableController.js which handles the action when a mouse button is pressed. The “ng-mousedown” differs from “ng-click” so that “ng-mousedown” action is fired immediately before the mouse button has been released. TableController holds an instance of JavaScript closure that encloses the handling of chessboard related functions.

The UI verifies from the first click to <td> containing a knight, that the click is made in the player's own turn and that the game time has not run out. After these conditions are met UI visualizes a square selection which can be seen in Figure 16.



Figure 16. Knight piece is selected.

The selection is changed if another <td> containing the player's own piece is clicked. An error message is shown if the knight tries to move into a <td> other than which contains data-grid "50" or "52". Otherwise the move is considered valid from the UI perspective. Instead of immediately updating the UI the move is sent to backend for validation. The message sent by the UI consists of the following text

```
{message: {command: "move", token: "8bec746b-0a43-4a51-887f-c48d4385fb52",
params: {0: {p0: "71"}, 1: {p1: "50"}}}}
```

where the token is the private key, p0 is from square and p1 is to square.

If the backend validation returns that the move is valid then the players and spectators receive a move command containing the actual move to be made. The UI moves the image from the source square to destination square by manipulating HTML DOM. Appendix 6 shows the move command from backend. All pieces of the player in turn need to be updated upon move command since the last move made has changed the state of the board and pieces.

4.7 Security – Client Cannot Be Trusted

HTML DOM is modifiable on the fly ending up into a situation where the sent parameters can be tampered. Security vulnerability is exposed if functionalities were implemented using public information. For example if the implementation needed messages such as

```
{message: {command: "remove_table", tableCreator: "Antsa"}}
```

or

```
{message:{command:"remove_table",tableId:"77f70955-821a-440f-aa33-
eb5af98ef653"}}
```

then others could make nuisance by removing tables that they did not create or send chat messages on behalf of others. These kinds of forgeries can be avoided using the private key as a determining element:

{message:{command:"remove_table",token:"1d93a6ed-5e1d-4b28-b326-14dc4d23d97e "}}. The back end system must anyway know the states of the users and tables and act accordingly.

However, it is noticeable that if in AntsaChess someone gets access to another user's private key for example using social engineering via chat then it is possible to represent another user. The aforementioned situation could be avoided if the current implementation checked that the incoming token belongs to the sender. Or even better yet, by not exposing the private token even to the user itself but keeping it only in between web server and ChessServer.

5 Communication between Domain Components

In this chapter it is examined how the messages move between web server and ChessServer and back. At first the message traversal is looked on from web server perspective and after that is explored how ChessServer handles the messages.

5.1 Web server as Intermediary Component

The role of the Tomcat web server is crucial. In this chapter the role as intermediary component is examined more closely. All communication between each party in AntsaChess goes through the web server and moreover it provides all the ingredients for web browsers to build the UI in the form of files.

A problem that this component solves can be related into a real world example where people from different countries talking different languages do not understand each other. Acting as an interpreter web server talks to browsers through the WebSocket and the files it has provided. With ChessServer the communication is done with Java objects.

Before diving into details of how this sophisticated interpreter has been technically implemented it is good to look at shared classes used in communication. AntsaChess code parts are divided in three projects which are AntsaChessInterface, AntsaChessServer and AntsaChessWeb. Additionally there is Servers2 project for Tomcat settings.

The AntsaChessInterface project contains all Java classes used in communication between ChessServer and the web server. Figure 17 shows the classes and how they are named.

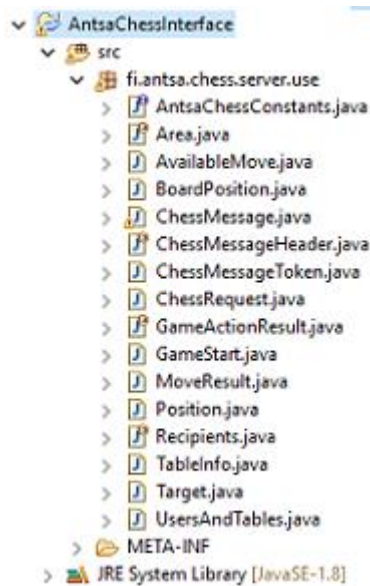


Figure 17. Classes used in communication between the servers.

Every message sent between servers is wrapped into `ChessRequest` which code can be viewed in Appendix 7. `ChessRequest` always contains a `ChessMessage` (Appendix 8). `ChessMessage`, on the other hand, always contains a `ChessMessageHeader` information (Appendix 9) and parameters related to the message. Mostly `ChessMessage` also contains the identifier token which is called `ChessMessageToken` (Appendix 10). Figure 18 shows the relations between core communication classes.

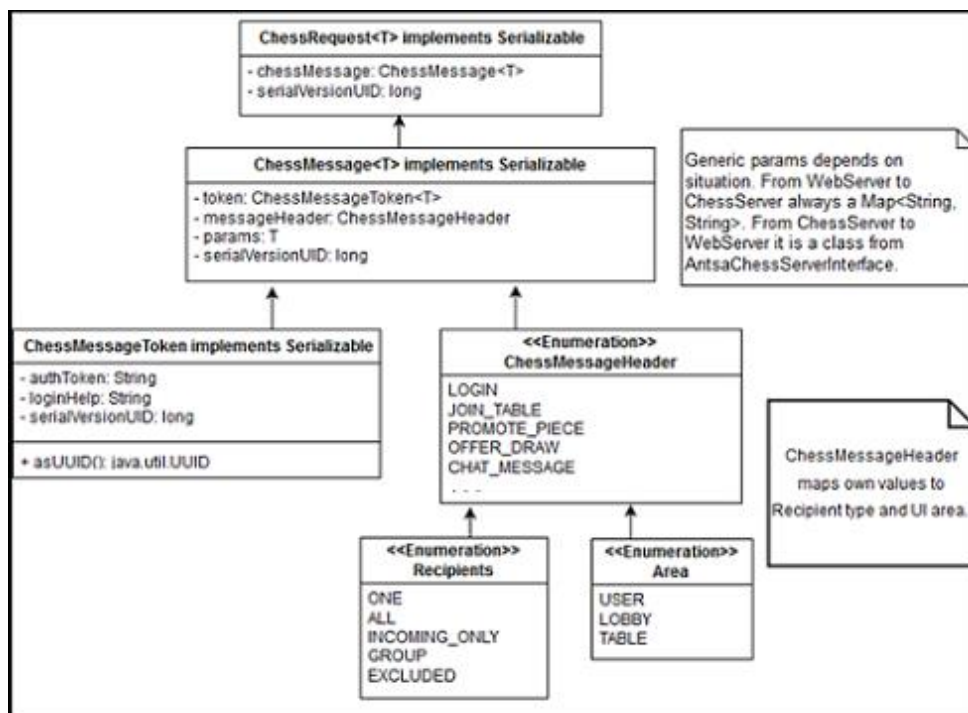


Figure 18. Class diagram of core classes used between servers.

The ChessMessage class contains a generic field “params” which content varies depending on the current flow.

Creating a ChessRequest from web server’s perspective

The web server creates a ChessRequest based on the JSON received from the UI. The whole web server domain is created using four custom classes which are shown in Figure 19.

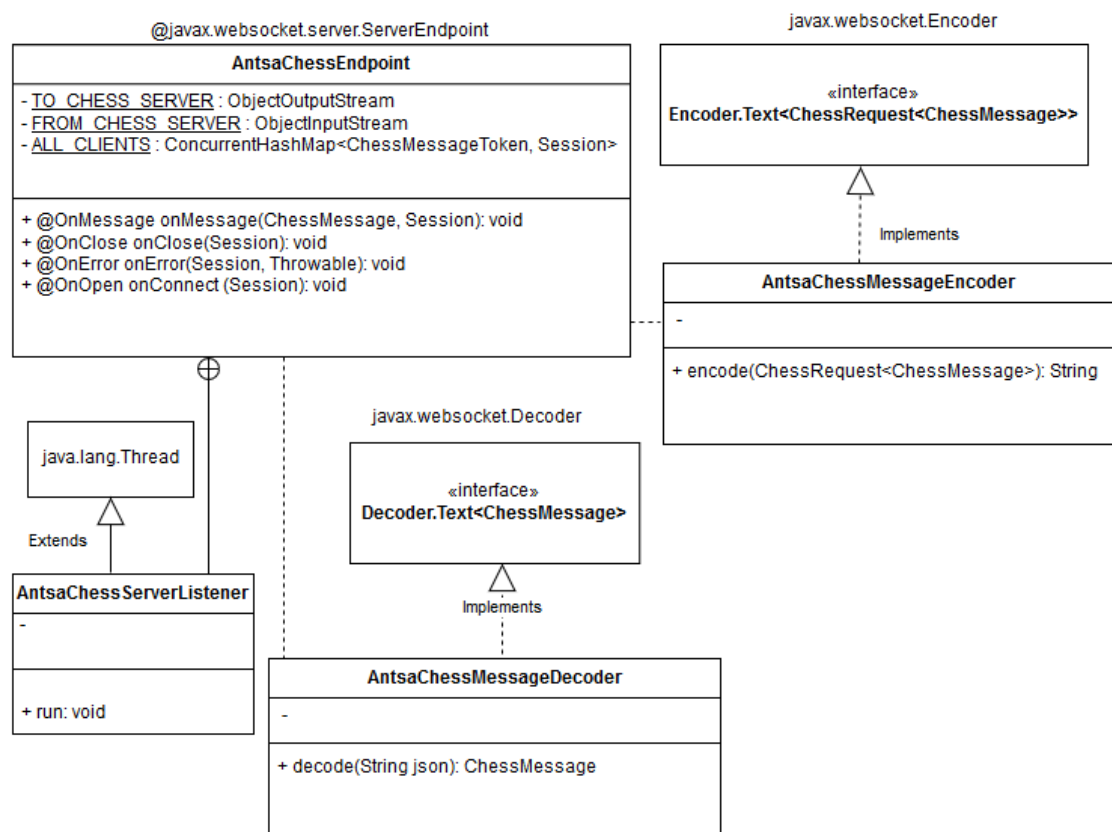


Figure 19. Web server domain classes.

The focal point is **AntsaChessEndpoint** which has the responsibility of converting all communication messages in a form which every endpoint can understand. The web server creates an instance of **AntsaChessEndpoint** for each connected browser and shares a single communication media to ChessServer between all clients.

In Figure 19 it can be observed that **AntsaChessEndpoint** is connected with **AntsaChessMessageDecoder**. **AntsaChessEndpoint** uses this decoder to convert JSON from client into **ChessMessage** class if possible. It does this just before the

@onMessage-method is called. The @onMessage in turn locks the shared java.io.ObjectOutputStream by synchronizing. Locking the ObjectOutputStream means that other users need to wait until the message is sent. This means potential bottleneck if the number of users is high enough. Annotated methods (@) are automatically invoked by the web server.

Receiving messages from ChessServer is done in a private static AntsaChessServerListener class which runs in a separate thread started from static initializer. This listener class observes the java.io.ObjectInputStream which is used to receive messages from ChessServer. The sole purpose of this class is to get rid of the message as fast as possible and continue listening new messages. Before doing so the class checks what kind of an object is received and for whom it is targeted and sends the message to its target(s). The message can be meant for a single client, for a group of clients or for all clients. The message must be encoded from Java object to JSON before sending it to a client. AntsaChessMessageEncoder is attached in this encoding operation.

5.2 AntsaChessServer as Backend Component

ChessServer domain is divided in two logical sections. The first section - ChessServer handles the internal logic of users, tables and games. The second section is WebClientMessageHandler which processes incoming and outgoing messages. Figure 19 shows the relations between the core logic components.

The startup of ChessServer is explored first and how messages sent from the web server are handled on ChessServer side. Figure 20 shows a sequence diagram of the startup process and message handling.

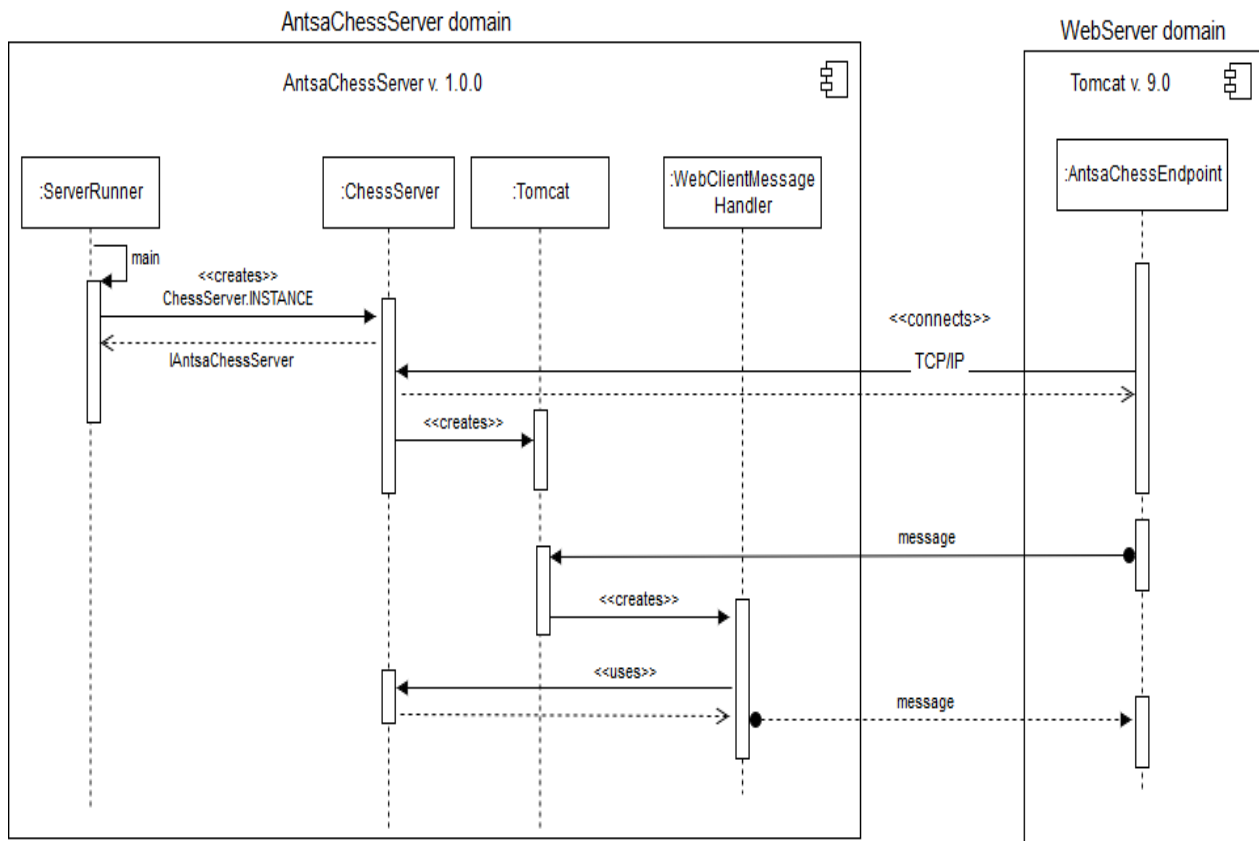


Figure 20. Sequence diagram from startup to message handling.

The **ServerRunner** class contains the **main** method from which the singleton **ChessServer** class is initialized. All classes in the **AntsaChessServer** domain can refer to **ChessServer** calling **ChessServer.INSTANCE** that returns **IAntsaChessServer** interface for using **ChessServer**'s services. During the startup process **ChessServer** is put to run in its own thread which listens to a connection from **AntsaChessEndpoint**. **ChessServer** thread is alive the whole system uptime.

After the connection between the web server and **ChessServer** has been made **ChessServer** creates an instance of **Tomcat** class which is put to run in a separate thread. During the creation time of **Tomcat** class the **ObjectOutputStream** and **ObjectInputStream** handles are given for the **Tomcat** class to handle the messaging with **AntsaChessEndpoint**.

The communication happens using pure Java objects as it was shown in section 5.1. From **ChessServer**'s perspective the incoming messages are received in the **Tomcat** class from **ObjectInputStream** and outgoing messages are sent to **ObjectOutputStream**. Only one message is received from **ObjectInputStream** at a time since the web server synchronizes the messages to **ChessServer**. In a theoretical scenario **ChessServer**

might need to access an external database to write player data or statistics while processing messages. Anyhow, handling each message takes an unknown amount of time during which another message from the web server can already arrive. For these kind of situations Tomcat assigns a fixed thread pool of the size of ten threads using a `java.util.concurrent.ExecutorService` for handling incoming messages.

The `ExecutorService`'s `execute` method handles threads so that it might use the current thread, create a new thread or use a thread from the pool when executing a message task (Oracle: `ExecutorService`). When Tomcat class receives a message it always creates a new `WebClientMessageHandler` instance which implements the `java.lang.Runnable` interface. The new instance of `WebClientMessageHandler` takes care of the message and is executed by `ExecutorService`. After creating the handler class Tomcat returns to listen to new messages. See Appendix 11 for the code used in the Tomcat class.

`WebClientMessageHandler` is a class dedicated to using `ChessServer`'s methods and sending response back to the web server. The core classes and their relations are introduced in Figure 18.

`WebMessageClientHandler` receives a `java.io.ObjectOutputStream` handle as a parameter to communicate back to the web server in a synchronized manner. In most cases the `WebMessageClientHandler` immediately sends a response back to the web server. But for instance when chess game has ended and one player clicks the rematch button `ChessServer` only registers the first message but does not send anything back. When the other player also clicks rematch button then the players and spectators receive a message to start a new game. The main responsibility of the `WebMessageClientHandler` is to interpret the parameters from `ChessRequest<T>` and use `ChessServer`'s provided methods with the extracted parameters. Appendix 12 shows how some of the methods are implemented in `ChessServer`. Appendix 13 shows how the `WebClientMessageHandler` utilizes `ChessServer`'s services.

6 AntsaChess Game with Object Oriented Style

In this chapter it is shown how the digital AntsaChess game is developed from the parts that are used in live chess games.

6.1 Converting Physical Chess to Digital One

It is easier to start thinking about chess in objects after looking at how real physical world games are conducted. A chess game is played with special chess pieces on a chessboard. The board is usually located on a table and around the table there are two players. Furthermore, there might be occasional spectators watching the game. Certain rules define the game and there might be an optional chess clock. Often several games are played during one session and pieces are changed between players after a game has ended. In the description above the underlined words are potential core objects to be used in any chess application with any object oriented programming language. In the case of AntsaChess Java is used which is an object oriented programming language by nature (James Gosling, Henry McGilton: 1996).

After thinking more thoroughly about the objects it can be quickly discovered that the sole core objects are not enough. A real chessboard consists of squares that might have different type and coloured chess piece. The players move the pieces in turns according to the rules. The dilemma is how the various chess rules can be applied and verifying that actions happen in the correct order. To overcome this dilemma a first option could be using a RuleBase object which validates every input received, calculates every allowed move and gives responses accordingly. This kind of an approach might end up in Swiss army knife anti-pattern situation where one class has too many responsibilities.

A second option could be to split the rules totally between objects. The problem with this approach could be that the rule logic is too decentralized and information needed in one object instance is not easily available. An example of a situation like this could be the king's castling move. It can be done only in the player's own turn if the rook involved has not moved, the king itself has not moved, the king is not in check, no piece is blocking the move, an opponent piece is not able to move in castling squares and an opponent pawn is not reserving the squares involved diagonally. Outside of the moving rule logic a player can choose to resign on his/her own will and players can mutually agree to draw the game. For the reasons mentioned above AntsaChess uses a combination of the first

and second option where there is a master controller for the rules but not containing all the calculations.

6.2 Processing Games

Figure 21 is a snapshot from the Eclipse workspace of AntsaChess project's classes that are used for creating the games, maintaining their states and controlling the rules.

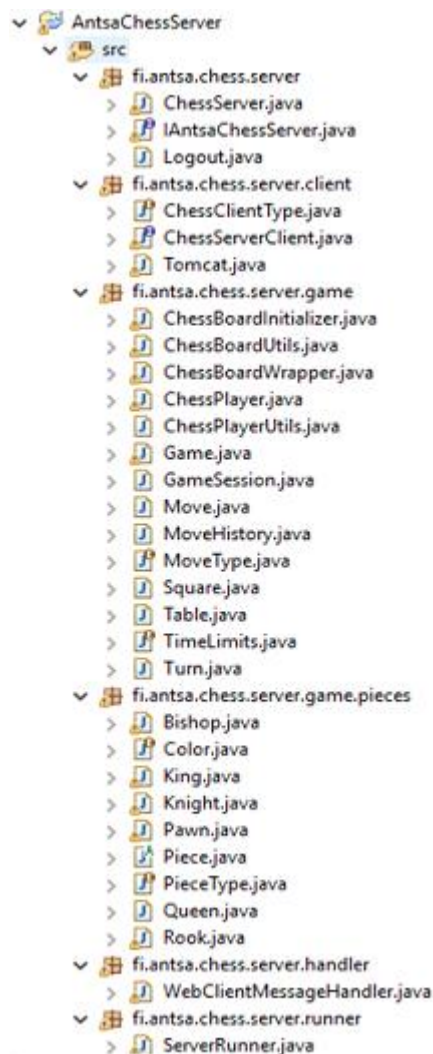


Figure 21. Java classes from AntsaChessServer -project.

Several consecutive games can be played against the same opponent and spectators are able to watch the games. It can be noticed that there is no Spectator nor Chat object. The reason for this is that spectators are also ChessPlayers and chat messages are not stored in the game thus they are conveyed outside the game concept. The main output

object that game returns is MoveResult which contains information about the chess pieces on board, where they can move and what the user can do and show next. ChessServer implements IAntsaChessServer which defines how ChessServer can be used inside AntsaChess domain. Detailed information is shown in Figure 22.

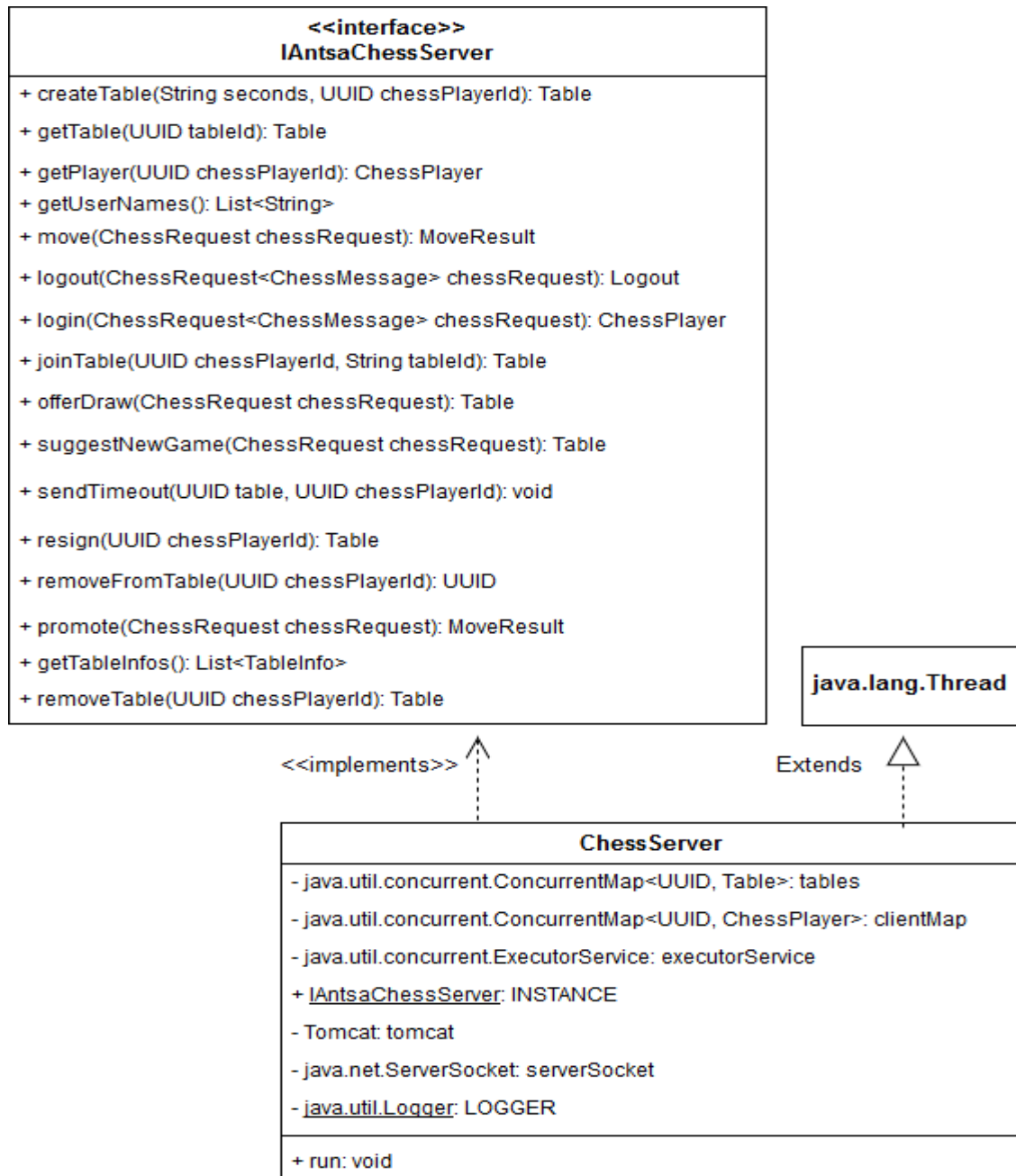


Figure 22. ChessServer and its interface.

ChessServer handles all simultaneous games and directs actions to the correct tables based on tableId. IAntsaChessServer interface shows all operations that can be invoked from outside of ChessServer class. The operations could also take

ChessRequest<ChessMessage> as the sole input parameter since this object contains all the information needed to handle the message (ChessRequest details are described in section 5.1). In some cases the method invoker WebClientMessageHandler, has already checked the information from ChessRequest and extracted it earlier.

The removeTable method takes playerId as a parameter and not tableId. This is because only the creator of the table can remove it. The downside of this decision is that only one table per player can be created at a time.

6.3 Table and Game Concept

Figure 23 shows AntsaChess class diagram of the table and game concept.

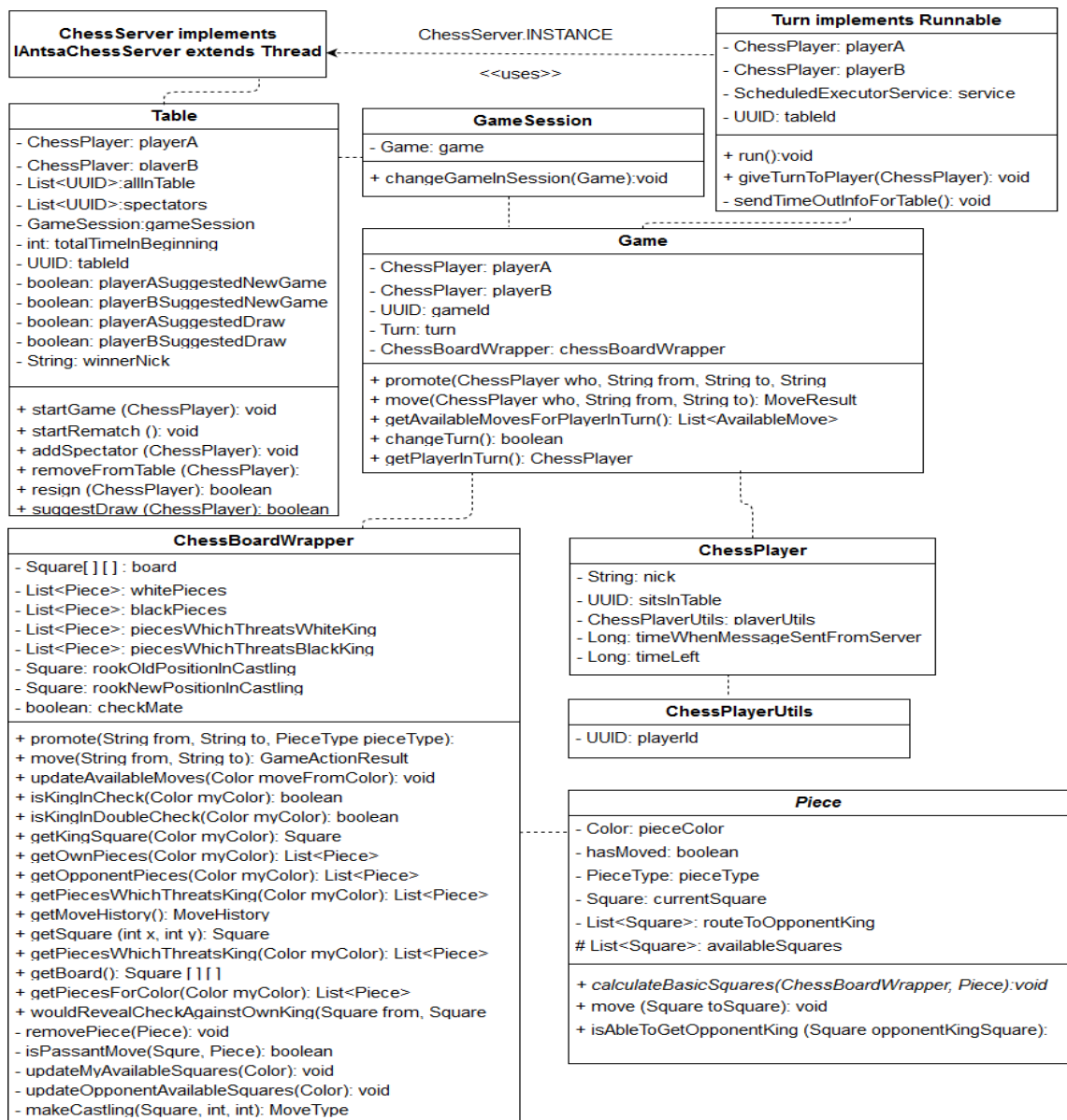


Figure 23. Class diagram used for playing games.

ChessServer as the master controller holds references to all game related objects.

The table class provides access to tamper the state of the game and the players in a game. The table contains GameSession which has handle to the actual game. The game knows the players which are playing, has a reference to ChessBoardWrapper and Turn objects. ChessBoardWrapper is the main rule base class but logic is divided also to other classes.

The turn class is responsible for maintaining information about which player is expected to make the next move and how much time the player has left. ChessBoardWrapper asks from the Turn object if the player who sent the move is actually in turn. If the player is not in turn ChessBoardWrapper sends back a MoveResult object which contains a Java enum titled GameActionResult.MOVE_NOT_ALLOWED. GameActionResult options are described later on this chapter.

Turn class is also responsible for handling timing controls of the players. Both of the players have the same selected time amount left in the beginning and after the first move time reducing starts. Turn executes `java.util.concurrent.ScheduledExecutorService` for every 500 ms which reduces 500 ms from the time the player in turn has left. The reason for timing starts only after the first move is that the UI takes a little time to load and the players can change words in the chat before starting. 500 ms was chosen because 1000 ms did not seem accurate enough.

The turn class does not end the game immediately when time left has gone to zero. This is because players can make a move faster than 500 milliseconds but the time penalty is always 500 ms. Also, the client UI updates take milliseconds and network latency varies between clients. When player's time left has gone more than 1500 milliseconds below zero the `ScheduledExecutorService` is shut down and the Turn class uses ChessServer to send timeout. ChessServer then uses the Tomcat class to push the message to players and spectators. The Game class shuts down the `ScheduledExecutorService` if the game ends before time runs out. It is necessary to check timings on the server side clock. If time left would be reduced only after the move comes in then there could be a situation that a player knows that he/she loses the game and might not make a move at all. This would mean that the game never ends on the server side and the opponent would end up waiting for the timeout based on the last move that never comes. One possibility still is that the move might come to ChessServer

immediately after time out has already been sent. Before making a move, promotion or castling the ChessBoardWrapper verifies from the Turn class that the player has not timed out.

The abstract Piece class knows the piece type and on what square in the chessboard it is. Implementing Piece classes e.g. King has their own moving rules which the pieces calculate by themselves getting the ChessBoard(Wrapper) as a calculation parameter. Pieces fill the availableSquare list based on their own rules. However, the Piece instances do not care about the ongoing situation on the board they always calculate the available squares based on their own moving rules.

It is the ChessBoardWrapper object's responsibility to remove the illegal squares from the pieces' availableSquare list based on the ongoing situation on the chessboard. An example of a situation when ChessBoardWrapper removes a square from a piece's available squares list is when the available Square would expose the player's own king to an opponent piece attack. Another example when an already calculated available Square is removed is when the player's own King is in a check situation and the piece cannot eat the threatening piece or the calculated Square is not blocking the threat.

ChessBoardWrapper internally stores the chessboard in a two dimensional Square object array. Table 7 shows the naming scheme used.

Table 7. Chessboard as two dimensional array.

8	00	01	02	03	04	05	06	07
7	10	11	12	13	14	15	16	17
6	20	21	22	23	24	25	26	27
5	30	31	32	33	34	35	36	37
4	40	41	42	43	44	45	46	47
3	50	51	52	53	54	55	56	57
2	60	61	62	63	64	65	66	67
1	70	71	72	73	74	75	76	77
	A	B	C	D	E	F	G	H

In a real chessboard squares are usually named with letters A -H and numbers from one to eight is used. In AntsaChess square H5 is called "37".

When a rematch is started a new Game object is created and set to GameSession replacing the old game. MoveResult is a class from the AntsaChessInterface project and it provides information for the UI about what to show and do in different situations. MoveResult contains GameActionResult which is a Java enum containing titles such as MOVE_NOT_ALLOWED, MOVE_DONE, CASTLING, PASSANT_DONE, TIMEOUT, WAITING_PROMOTION_SELECTION and CHECKMATE. GameActionResult is a title for MoveResult. Based on the title the utilisers know which fields they need to read from MoveResult. GameActionResult is also used internally in promotion situations.

The AvailableMove class contains a knowledge about the piece position in String format as in "34" and a list about the squares – in String format, where the piece can move. Each game has a single Turn object that measures time for both of the players. If the server side game time runs out the Turn object sends a push message using ChessServer's IAntsaChessServer interface which ChessServer implements.

7 Testing and Analysing AntsaChess

The AntsaChess system consist of the UI, the web server and ChessServer. This chapter focuses on testing some of the parts that might cause problems when the user amount increases. In addition, some ideas is given about how a similar kind of game system could be tested for ensuring the correct results.

7.1 Testing Chess Moves

Verifying that the pieces are able to make only correct moves is a complex task. The number of move combinations is enormous because every made move changes the state of the chessboard which possibly has effect on other pieces. Also, the opponent's potential moves affect what the player in turn can do. Manual testing through UI is the most realistic but very time consuming way to verify that a player can make only the moves allowed by the rules.

In the basic flow it is required to start ChessServer and the web server, open two browser windows or tabs, log in, create a table and join a game. After these initiative tasks the wanted board position needs to be played in order to see what are the valid moves for a piece in test. If some move accidentally goes wrong then a new game needs to be started and played to the same position again. One way to fasten the process on the server side is to setup a precondition where opening a starting page would automatically create a table for the first browser and to join from another browser. Still moving the pieces to erroneous positions is required.

Another way to test is without the UI at all. This can be done by creating only the pieces needed in the current test scenario and put them in correct positions on the board. Figure 24 shows a test code where it was programmatically tested that a player with white pieces could make a castling to the left side starting from the beginning of the game.

```

15 public class ChessBoardInitializer {
16     public static void main(String[] args) {
17         ChessBoardWrapper chessBoardWrapper = new ChessBoardWrapper();
18         castlingToLeftSucceeds(chessBoardWrapper);
19     }
20
21     private static void castlingToLeftSucceeds(ChessBoardWrapper
22         chessBoardWrapper) {
23         assert (chessBoardWrapper.move("64", "44")
24             == GameActionResult.MOVE_DONE) : "error, move did not go through";
25         assert (chessBoardWrapper.move("14", "34")
26             == GameActionResult.MOVE_DONE) : "error, move did not go through";
27         assert (chessBoardWrapper.move("73", "64")
28             == GameActionResult.MOVE_DONE) : "error, move did not go through";
29         assert (chessBoardWrapper.move("11", "21")
30             == GameActionResult.MOVE_DONE) : "error, move did not go through";
31         assert (chessBoardWrapper.move("63", "43")
32             == GameActionResult.MOVE_DONE) : "error, move did not go through";
33         assert (chessBoardWrapper.move("06", "25")
34             == GameActionResult.MOVE_DONE) : "error, move did not go through";
35         assert (chessBoardWrapper.move("72", "63")
36             == GameActionResult.MOVE_DONE) : "error, move did not go through";
37         assert (chessBoardWrapper.move("17", "37")
38             == GameActionResult.MOVE_DONE) : "error, move did not go through";
39         assert (chessBoardWrapper.move("71", "52")
40             == GameActionResult.MOVE_DONE) : "error, move did not go through";
41         assert (chessBoardWrapper.move("37", "47")
42             == GameActionResult.MOVE_DONE) : "error, move did not go through";
43         assert (chessBoardWrapper.move("74", "72")
44             == GameActionResult.CASTLING) : "error, castling did not happen";
45     }

```

Figure 24. Asserts can be used for testing moves and validating outputs.

In Figure 24 an instance of ChessBoardWrapper object is created on the main method. Internally this class setups the chessboard to the starting position using a two dimensional Square [8][8] object. There was no Table nor ChessServer objects related to the test since the intention was to test that a certain move sequence is allowed and castling occurred. Moving started from the line 23 with E2 to E4 square (64 - 44) followed by E7 to E5 (14 - 34) ending up in the white king's castling. If the ChessboardWrapper returned anything else than GameActionResult.MOVE_DONE on any of the moves before the castling move then the test case would fail.

Java asserts can be used to create a test set which at best verifies the wanted results throughout the lifecycle of a system. Java virtual machine needs a special -ea parameter for enabling asserts. Using the assert style for testing is convenient when one needs to verify that the program flow goes as expected (Oracle: Programming With Assertions). If all the moves go through before the castling move then putting a debugger breakpoint for example at line 43 in Figure 24 and then going step by step through the code saves time when finding out why the king is not able to make the castling. This kind of a testing mechanism and can be used with some of the trickier cases during development.

7.2 Message Delivery Times between Domain Systems

The purpose of this test was to find out how much time it takes for sending ChessRequest with different payloads from the web server to ChessServer. In a multithread environment which the AntsaChess concept is many threads might want to use the shared TCP/IP connection at the same time. Without synchronizing these writes on both ends the result would end up in a failure. The penalty of synchronizing is that only one thread can write at time and other threads need to wait for the previous one to complete.

The message delivery must be fast in order for the game and other actions to be fluent. Some magnitude of the connection locking times could be obtained by creating a manual test with a few browser tabs using the development computer as the test device and Eclipse as a platform. The test computer had Windows 10 and hardware which consisted of Intel Core i5 3230M @ 2.6 GHz and 6 GB DDR3 memory @ 1600MHz with 4 cores available to Java virtual machine. Table 8 shows the lock duration time when the web server sent a java object to ChessServer.

Table 8. Locking times for TCP/IP connection.

Message	Locktime (ms)
LOGIN	16
LOGIN	3
LOGIN	1
LOGIN	3
CREATE_TABLE	1
REMOVE_TABLE	1
CREATE_TABLE	2
JOIN_TABLE	2
MOVE	2
MOVE	2
PROMOTE	2
CHAT_MESSAGE	4
CHAT_MESSAGE	2

By looking at Table 8 it can be detected that sending an object to ChessServer for the first time takes 4 -16 times more than average. After restarting the servers and testing the locking times for a second time the results were similar. Most likely sending the first object through the pipe took more time because of the internal initialization of `java.io.ObjectOutputStream`. Tests were done comparing system timestamps just before an object was serialized and after the deserialization was completed and the object was ready to use.

The conclusion from the test was that the message payloads are not too big since the maximum locking time was 16 ms. At this point when there was not so many concurrent users no further actions were needed to improve delivery times. With better hardware the locking times could be a little lower. It is also good to notice that there was no other threads waiting during the test since the test was done using one UI at a time. In addition, the servers resided on a single computer thus there was real network between the web server and ChessServer.

7.3 Performance Testing

At this point the system internal robustness was unknown. There was no testing group to help verifying that how many concurrent games can be played. For finding out reasonable system limits a test code was generated. The following tests were run in a sequence so that only the code relevant to current test is shown and the tests demonstrated are continuation to the previous one.

`IAntsaChessServer` and its implementing class `ChessServer` were modified so that it was possible to reference the players and the tables outside `ChessServer`. Otherwise references to tables and players would have been needed to store also in test code. Tests were run using Oracle JRE 1.8.0_121.

Logging players in

In Figure 25 there is code for testing can `ChessServer` handle 30 000 logins created from the main thread.

```

public class Tester {
    @SuppressWarnings({ "unchecked", "rawtypes" })
    public static void main(String[] args) {
        ChessMessageHeader loginHeader = ChessMessageHeader.LOGIN;
        Map<String, String> params = new HashMap<String, String>();
        long startTime = System.nanoTime();
        for (int i = 0; i < 30000; i++) {
            params.remove("p0");
            String nick = Integer.toString(i);
            params.put("p0", nick);
            ChessMessage cm =
                new ChessMessage(loginHeader,
                    new ChessMessageToken(UUID.randomUUID().toString()), params);
            ChessRequest request = new ChessRequest<>(cm);
            ChessServer.INSTANCE.login(request);
        }
        long endTime = System.nanoTime();
        System.out.println("elapsedTime:" +
            TimeUnit.NANOSECONDS.toSeconds(endTime - startTime));
        System.out.println("players:" +
            ChessServer.INSTANCE.getPlayers().size());
        System.exit(0);
    }
}

elapsedTime:27
players:30000

```

Figure 25. Code and the output of the test.

ChessServer expected that a key “p0” has a nickname as a value. In Figure 25 the number of the loop iteration was used as the value for a single login. The result was that all 30 000 players were able to login approximately in half a minute. This number of successful logged in players was more than enough. Players were internally put to a `java.util.ConcurrentHashMap` which is thread safe meaning that using several concurrent threads to do the same work would be fine.

Available number of concurrent tables

The purpose of this test was to find out whether all the 30 000 logged in players can also create a table with one minute game time. The test was carried out with the additional piece of code shown in Figure 26.

```

        startTime = System.nanoTime();
        Map<UUID, ChessPlayer> playerMap = ChessServer.INSTANCE.getPlayers();
        playerMap.entrySet().forEach(entry -> {
            ChessServer.INSTANCE.createTable("1",
                entry.getValue().getPlayerUtils().getPlayerId());
        });
        endTime = System.nanoTime();
        System.out.println("tables created:" +
            ChessServer.INSTANCE.getTables().size());
        System.out.println("elapsedTableTime:" +
            TimeUnit.NANOSECONDS.toSeconds(endTime - startTime));
        System.exit(0);
    }
}
tables created:30000
elapsedTableTime:39

```

Figure 26. Code and output for creating tables.

The result was that all the 30 000 players could create their own table in approximately half of a minute.

Joining a game

With 30 000 players the maximum of concurrent games is 15 000 because a game needs two players. The purpose of this test was to verify whether 15 000 players can join into existing tables - one for each. Figure 27 show the additional test code and its output is.

```

    startTime = System.nanoTime();
    Map<UUID, ChessPlayer> playerMap = ChessServer.INSTANCE.getPlayers();
    Iterator i = playerMap.entrySet().iterator();
    int iteratedAmount = 0;
    Table table = null;
    while (i.hasNext()) {
        Map.Entry entry = (Map.Entry) i.next();
        ChessPlayer player = (ChessPlayer) entry.getValue();
        if (iteratedAmount % 2 == 0) {
            table = ChessServer.INSTANCE
                .createTable("1", player.getPlayerUtils()
                    .getPlayerId());
        } else {
            Table joinTable = ChessServer.INSTANCE
                .joinTable(player.getPlayerUtils().getPlayerId(),
                    table.getTableId().toString());
            //ChessServer returns table if join is ok
            if(joinTable == null){
                throw new RuntimeException("join not possible!?!");
            }
        }
        iteratedAmount++;
    }
    endTime = System.nanoTime();
    System.out.println("tables created:" +
        ChessServer.INSTANCE.getTables().size());
    System.out.println("elapsed time create/join:" +
        TimeUnit.NANOSECONDS.toSeconds(endTime - startTime));
    System.exit(0);
}
elapsedTime for logins:27
players added:30000
tables created:15000
elapsed time create/join:11

```

Figure 27. Code and output for creating tables and joining them.

The result of this test shows that all 15 000 players were able to join the table since no exception was thrown. All actions took time about 40 seconds.

Playing the game

The purpose of this test was to find out how much ChessServer needs time for validating a move and changing turn under a heavy load. A similar concept to the one AntsaChess uses in reality was used. ChessServer's move method was called by WebClientMessageHandler and the method's return value was serialized to a new test class. In this test 3000 players were logged in and 1500 tables were created. In the beginning of the games all 1500 players with white pieces tried to move a white piece pawn from E2 to E4 followed by a black piece move pawn from E7 to E5.

A new test class AntsaChessEndpointMock was created in ChessServer domain. This class represented the real AntsaChessEndpoint class in web server domain that handles messages between browsers and ChessServer. The connection between ChessServer and the mock class was created in static block as shown in Figure 28.

```
package fi.antsa.chess.server.runner;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.net.Socket;
import javax.net.SocketFactory;
import fi.antsa.chess.server.use.ChessMessage;
import fi.antsa.chess.server.use.ChessMessageHeader;
import fi.antsa.chess.server.use.ChessRequest;
import fi.antsa.chess.server.use.MoveResult;
//real AntsaChesspoint has also inner private class
public class AntsaChessEndpointMock {
    //not thread safe itself
    private static final ObjectInputStream FROM_CHESS_SERVER;
    static {
        ObjectInputStream i = null;
        Socket socket;
        try {
            socket = SocketFactory
                .getDefault().createSocket("127.0.0.1", 15000);
            i = new ObjectInputStream(socket.getInputStream());
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }
        FROM_CHESS_SERVER = i;
        new AntsaChessServerListener().start();
    }

    @SuppressWarnings("rawtypes")
    private static final class AntsaChessServerListener extends Thread {
        int counter=0;
        @Override
        public void run() {
            while (true) {
                try {
                    ChessRequest<ChessMessage> r =
                        (ChessRequest<ChessMessage>) FROM_CHESS_SERVER.readObject();
                    counter++;
                    if(r.getMessageHeader()==ChessMessageHeader.MOVE){
                        MoveResult m=r.getChessMessage().getParams();
                        System.out.println("In MockClass move from:"
                            +m.getFrom()+" to:"+m.getTo()+" totalMessages:"+counter);
                    }else if(r.getMessageHeader()==ChessMessageHeader.TIMEOUT){
                        System.out.println("In MockClass timeout:"+counter);
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

Figure 28. Code used in AntsaChessEndpointMock.

Real AntsaChessEndpoint uses a similar concept as the mock class. The part where the message is examined more closely in the real AntsaChessEndpoint and sent to browsers is omitted.

Figure 29 shows the code part how the connection between mock class and ChessServer was created.

```
public class Tester {
    @SuppressWarnings({ "unchecked", "rawtypes" })
    public static void main(String[] args) {
        // Starts the ChessServer
        IAntsaChessServer c = ChessServer.INSTANCE;
        // Mock connects to server and starts to listen
        new AntsaChessEndpointMock();
        // Modified ChessServer returns handle to AntsaChessEndpointMock
        ObjectOutputStream out = ChessServer.INSTANCE.getTomcat().getOut();
    }
}
```

Figure 29. Code for creating the connection between ChessServer and mock.

After running the code from Figure 29 all 3000 logins and 1500 tables was created using the same code from previous examples. Figure 30 shows the additional code used in this test where all tables are iterated and the two moves in each are sent to ChessServer using the WebClientMessageHandler class.

```

//logins and table creations are done before code below
Map<UUID, Table> tables = ChessServer.INSTANCE.getTables();
ChessMessageHeader moveHeader = ChessMessageHeader.MOVE;
ExecutorService executor = Executors.newFixedThreadPool(10);
try {
    startTime = System.nanoTime();
    //each table is iterated
    for (Map.Entry<UUID, Table> entry : tables.entrySet()) {
        Table t = entry.getValue();
        // Table internally checks from the Turn class who is in turn.
        UUID playerIdInTurn = t.getPlayerInTurn().
            getPlayerUtils().getPlayerId();
        Map<String, String> moveParams = new HashMap<String, String>();
        moveParams.put("p0", "64");
        moveParams.put("p1", "44");
        ChessMessage cm = new ChessMessage(moveHeader,
            new ChessMessageToken(playerIdInTurn.toString()),
            moveParams);
        ChessRequest request = new ChessRequest<>(cm);
        //WebClientMessageHandler synchronizes outputStream
        //and writes ChessServer response to Mock class.
        executor.execute(new WebClientMessageHandler(out, request));
        // 20ms sleeping in main thread seems enough
        Thread.sleep(20);
        playerIdInTurn = t.getPlayerInTurn().getPlayerUtils().getPlayerId();
        Map<String, String> moveParams2 = new HashMap<String, String>();
        moveParams2.put("p0", "11");
        moveParams2.put("p1", "21");
        ChessMessage cm2 = new ChessMessage(moveHeader,
            new ChessMessageToken(playerIdInTurn.toString()), moveParams2);
        ChessRequest request2 = new ChessRequest<>(cm2);
        executor.execute(new WebClientMessageHandler(out, request2));
    }
    endTime = System.nanoTime();
    System.out.println("TesterClass elapsed time for moves:"
        + TimeUnit.NANOSECONDS.toSeconds(endTime - startTime));
} catch (Exception e) {
    System.err.println("Error " + e);
    System.exit(0);
}
}

```

Figure 30. Additional code used in the test.

The code in Figure 30 puts the running thread to sleep for 20 ms after the first move is executed. If the sleep time was lower than 20 ms then in some of the tables the second move got a response stating that MOVE_NOT_ALLOWED. This was because ChessServer had not enough time for validating the previous move in the table and the turn had not changed yet. The lower the sleep time was the more often this response was received.

AntsaChessEndpointMock class counted all received messages tagged with the MOVE header.

The beginning and the end of the output from this test can be seen in Figure 31.

```

elapsedTime for logins:0
players added:3000
total amount of tables:1500
elapsed time for create/join:0
In MockClass move from:64 to:44 totalMessages:1
In MockClass move from:11 to:21 totalMessages:2
In MockClass move from:64 to:44 totalMessages:3
In MockClass move from:64 to:44 totalMessages:4
In MockClass move from:11 to:21 totalMessages:5
In MockClass move from:11 to:21 totalMessages:6
In MockClass move from:64 to:44 totalMessages:7
In MockClass move from:64 to:44 totalMessages:8
In MockClass move from:11 to:21 totalMessages:9

In MockClass move from:64 to:44 totalMessages:2997
In MockClass move from:11 to:21 totalMessages:2998
In MockClass move from:64 to:44 totalMessages:2999
TesterClass elapsed time for moves:30
In MockClass move from:11 to:21 totalMessages:3000
In MockClass timeout:3001
In MockClass timeout:3002
In MockClass timeout:3003
In MockClass timeout:3004
In MockClass timeout:3005

In MockClass timeout:4492
In MockClass timeout:4493
In MockClass timeout:4494
In MockClass timeout:4495
In MockClass timeout:4496
In MockClass timeout:4497
In MockClass timeout:4498
In MockClass timeout:4499
In MockClass timeout:4500

```

Figure 31. Output from the beginning and from the end of test.

Looking at the output lines in Figure 31 it was detected that all logins took less than a second likewise creating the tables and joining them. The Mock class received move messages in a different order as they were originally sent from the Tester class. ExecutorService's implementing class has its internal rules for handling tasks. A thread pool with 10 threads was chosen because in real application Tomcat class has the same number.

The Tester class finished sending 3000 moves in a little over 30 seconds but the elapsed time was rounded to 30 seconds. Most of the time went in waiting between moves where the thread was sleeping. One message still arrived in AntsaChessEndpointMock when the Tester class in main thread had already finished.

The games were still ongoing even though all wanted moves were sent meaning that 1500 tables still ran a counter which reduced the game time for the players in turn. After the game time ended in the tables ChessServer sent a timeout message for each table. The Mock class started to add the counter based on messages' TIMEOUT header.

The total number of logged messages was 4500 which contained 3000 MOVE messages and 1500 timeout messages. This test was successful. However, it cannot be guaranteed that 20 ms would always be enough for validating a move and changing the turn. Diminishing little the main thread sleep time between moves caused problems where for example the Turn class did not shut down the chess clock timer. Nevertheless, selecting a chess piece and sending the next move from a real browser takes a lot more time than 20 ms. As a conclusion of this test it can be said that validating a move and changing the turn takes about 20 ms.

8 Summary

It was found out that WebSockets between the web server and clients' browsers enable dynamic and fast online games. Messaging between the frontend and the backend seemed to be very fast using the architecture created in the proof of concept phase. However, it must be understood that messaging differs from HTTP GET/POST methods where requests and responses can be thought of as pair. In AntsaChess sending a message from the UI does not necessarily need to create any response since the messages are independent from each other.

There is no backup if the receiver fails to notice or react on a sent message. An example of this kind of a situation could be that a player with white pieces makes a first move immediately after his/her UI is loaded. The system sends the move over to the opponent but the UI might not yet be fully loaded meaning that the player with black pieces never sees the first move.

The chess clock on the UI side worked fine during the chess development time. However, when the game was played against a real opponent it was soon noticed that the UI chess clock time ran out much faster than the server side chess clock. The reason for this was that with a real opponent some of the moves were made much faster than one move per second. The UI removes always at least a second from the player's or the opponent's chess clock. If moves are made every half a second then in a one minute game the UI chess clock would run out after half minute. However, this did not affect the gameplay and merely caused some confusion.

There is no perfect solution for the online computer timing dilemma where both players would get the exactly equal amount of thinking time. Network latency is different between players and the server. Furthermore, network latency can vary even between every move so manipulating the chess clock based on ping times would not make the timing perfect. In addition, the end devices and server(s) load varies in time meaning that processing the messages do not always take the same amount of time. Usability wise microseconds nor milliseconds difference has no effect. However, a better approach in AntsaChess would have been to synchronize the UI chess clock with the server side chess clock on every move. The move command JSON already contains information about how much the players have time left and it would have been less confusing to add or remove a

second to the chess clock every now and then instead of wondering why does the game not end when the chess clock goes to zero in the UI.

One of the biggest misconceptions in the development was related to WebSockets. Until the very end of the implementation part, the impression was that there was only a single instance of WebSocket endpoint in the web server that handles all client connections. Actually the Tomcat web server creates an endpoint instance for every client. If this had been known after the POC phase the architecture could have been different so that the web server could have created direct TCP/IP connection per client instance to AntsaChessServer. There would not have been need for private token identifier at all since it would have been known that a connection pipe belongs to a certain player. With the current architecture many concurrent users will most likely form some sort of a bottle neck at some point since the single connection pipe is synchronized.

However the truth is not that unambiguous. Considering a theoretical scenario where the chess world championship final match is played on robust AntsaChess and 9998 people were following the match concurrently online. With the current architecture the web server is responsible for delivering the moves for the 10 000 people around the table and ChessServer needs to send the moves only once to the web server. With an architecture where 10 000 TCP/IP pipes would be created directly from the web server to AntsaChessServer it is necessary to first transfer the same information in all TCP/IP pipes and web server would still need to deliver the data to all client WebSockets. Designing the architecture for any software would be easier if it was beforehand known how the application will be used and what the average and peak time user numbers are.

Looking at the initial requirements set for the system it can be said that five out of eight requirements were applied. All requirements with critical and high status were implemented. Measuring user experience was not done using a questionnaire as it was originally considered. Instead feedback and ideas were received during the study and real live games from the instructor of the thesis. Localisation and gathering user statistics were left for future development.

References

Mozilla Developer network, A Typical HTTP session,

<https://developer.mozilla.org/en-US/docs/Web/HTTP/Session> (Accessed Apr 17 2017)

Network Working Group, June 1999, RFC 2616 HTTP 1.1

<https://tools.ietf.org/html/rfc2616> (Accessed Apr 17 2017)

Mozilla Developer network, WebRTC data channels,

https://developer.mozilla.org/en-US/docs/Games/Techniques/WebRTC_data_channels

(Accessed Apr 17 2017)

Internet Engineering Task Force, Dec 2011, The WebSocket Protocol

<https://tools.ietf.org/html/rfc6455> (Accessed Apr 17 2017)

Apple iPhone 5, Jan 2017, Technical Specifications

https://support.apple.com/kb/SP655?viewlocale=en_US&locale=fi_FI (Accessed Apr 17 2017)

W3Schools, JSON - Introduction

http://www.w3schools.com/js/js_json_intro.asp (Accessed Apr 17 2017)

AngularJS, FAQ

<https://docs.angularjs.org/misc/faq> (Accessed Apr 17 2017)

jQuery, What is jQuery?

<https://jquery.com/> (Accessed Apr 17 2017)

Oracle, Class UUID

<https://docs.oracle.com/javase/8/docs/api/java/util/UUID.html> (Accessed Apr 17 2017)

Wikipedia, Image use policy, User-created images

https://en.wikipedia.org/wiki/Wikipedia:Image_use_policy (Accessed Apr 17 2017)

Johan Vos, Oracle, Apr 2013, JSR 356, Java API for WebSocket

<http://www.oracle.com/technetwork/articles/java/jsr356-1937161.html> (Accessed Apr 17 2017)

Johan Vos: Oracle, Apr 2013, JSR 356, Annotation-Driven Approach

<http://www.oracle.com/technetwork/articles/java/jsr356-1937161.html> (Accessed Apr 17 2017)

Techopedia, Proof of Concept (POC)

<https://www.techopedia.com/definition/4066/proof-of-concept-poc>
(Accessed Apr 17 2017)

The Apache Software Foundation, May 2015, Apache Tomcat 8,

<http://tomcat.apache.org/tomcat-8.0-doc/web-socket-howto.html>, (Accessed Apr 17 2017)

Eclipse, Jetty

<http://www.eclipse.org/jetty/> (Accessed Apr 17 2017)

caniuse, May 2015, Web Sockets,

<http://caniuse.com/#feat=websockets> (Accessed Apr 17 2017)

HTML5 final recommendation, 28 Oct 2014,

<https://www.w3.org/TR/html5/> (Accessed Apr 17 2017)

Oracle, ExecutorService execute method

<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executor.html#execute-java.lang Runnable-> (Accessed Apr 17 2017)

James Gosling, Henry McGilton 1996, The Java™ Language Environment
A White Paper (1.2.1)

www.hs-augsburg.de/informatik/projekte/mebib/emiel/entw_inf/lernprogramme/java/Tools/Java/Doc/Papers/langenviron.pdf (Accessed Apr 17 2017)

Oracle, Programming With Assertions

<http://docs.oracle.com/javase/8/docs/technotes/guides/language/assert.html>
Accessed Apr 17 2017)

WebSocket.js

```
var websocket =
    new WebSocket('ws://localhost:8080/AntsaChessWeb/chess/');
function onOpen(event) {
    console.log("websocket OnOpen" + event);
}
function onMessage(event) {
    var skooppi = angular.element(getController(event)).scope();
    skooppi.$apply(function() {
        skooppi.handleAntsaChessEvent(event);
    });
}
function getController(event) {
    var data = JSON.parse(event.data);
    if (data.message.area == "USER") {
        return document.getElementById('userId');
    } else if (data.message.area == "LOBBY") {
        return document.getElementById('lobbyId');
    } else if (data.message.area == "TABLE") {
        return document.querySelector(".gameboard");
    } else {
        alert("Module not found" + data.message.area);
    }
    return false;
}

function sendToWs(objectToWebSocket) {
    var tulos = JSON.stringify(objectToWebSocket);
    websocket.send(tulos);
    return false;
}

websocket.onerror = function(event) {
    onError(event)
};
websocket.onopen = function(event) {
    onOpen(event)
};
websocket.onmessage = function(event) {
    onMessage(event)
};

websocket.onmessage = function(event) {
    onMessage(event)
};
websocket.onclose = function(event) {
    onClose(event)
};
function onError(event) {
    alert(event.data);
}
function onClose(event)
{
    alert("Connection has been closed, reload the page and log in again");
}
```

First version of Lobby UI

Home Register Lobby		Hello, etunimifhfhfh	
		Logout	
Lobby,			
13 antsa etunimifhfhfh ququq KINGsLAYER QEQ Chrome etunimiffwqq		Pöytäosuus tähän Create new table PelaajaQQ:PelaajaWW, sec. Watch PelaajaRR:PelaajaAA, sec. Watch	

Start game with white pieces command peek

```

message      {}
  area      "TABLE"
  command   "START_GAME_AS_WHITE"
  gameInfo  {}
    blackPlayerName "Joiner"
    blackLeft      300
    tableId        "4bce2941-e24b-4ffb-86e9-df990d0056c5"
    boardPosition  {}
      positions    {}
        0          {}
          pieceType "fake"
          imageUrl  "https://upload.wikimedia.org/wikipedia/commons/b/b1/Chess_blt45.svg"
        1          {}
          pieceType "rook"
          imageUrl  "https://upload.wikimedia.org/wikipedia/commons/f/ff/Chess_rdt45.svg"
          coordinates "00"
          colorAbr    "b"
          availableMoves ""
        2          {}
        3          {}
        lines      omitted
        59         {}
          pieceType "pawn"
          imageUrl  "https://upload.wikimedia.org/wikipedia/commons/4/45/Chess_plt45.svg"
          coordinates "64"
          colorAbr    "w"
          availableMove "54,44,"
        60         {}
        69         {}
          pieceType "bishop"
          imageUrl  "https://upload.wikimedia...ns/b/b1/Chess_blt45.svg"
          coordinates "75"
          colorAbr    "w"
          availableMove ""
        70         {}
        71         {}
        whiteLeft   300
        whitePlayerName "Antsa"
        me

```

Piece image URLs

Pawn light

https://upload.wikimedia.org/wikipedia/commons/4/45/Chess_plt45.svg

Pawn dark

https://upload.wikimedia.org/wikipedia/commons/c/c7/Chess_pdt45.svg

Queen light

https://upload.wikimedia.org/wikipedia/commons/1/15/Chess_qlt45.svg

Queen dark

https://upload.wikimedia.org/wikipedia/commons/4/47/Chess_qdt45.svg

King light

https://upload.wikimedia.org/wikipedia/commons/4/42/Chess_klt45.svg

King dark

https://upload.wikimedia.org/wikipedia/commons/f/f0/Chess_kdt45.svg

Bishop light

https://upload.wikimedia.org/wikipedia/commons/b/b1/Chess_blt45.svg

Bishop dark

https://upload.wikimedia.org/wikipedia/commons/9/98/Chess_bdt45.svg

Knight light

https://upload.wikimedia.org/wikipedia/commons/7/70/Chess_nlt45.svg

Knight dark

https://upload.wikimedia.org/wikipedia/commons/e/ef/Chess_ndt45.svg

Rook light

https://upload.wikimedia.org/wikipedia/commons/7/72/Chess_rlt45.svg

Rook dark

https://upload.wikimedia.org/wikipedia/commons/f/ff/Chess_rdt45.svg

Chessboard.js JavaScript peek

```

'use strict';

function createChessBoard(isWhite) {
    var pColor = null;
    var tableAsString = null;
    var selectedSquare = null;
    var lastMovedSquare = null;
    var inTurn = isWhite;
    var selectedSquareBackUpColor = null;
    var lastMovedSquareBackUpColor = null;
    var timeout= undefined;
    var spectator=false;

    //Builds the table from boardPosition parameter.
    //Spectators might come to watch in the middle of game.
    init : function(color,boardPosition,spectator) {
        if (color) {
            pColor = "w";
            inTurn = true;
        } else {
            pColor = "b";
            inTurn = false;
        }
        //Spectators must not be able to select pieces though server side verifies the sender also.
        spectator=spectator;
        var tableBuilder = "<table id=\"boardTable\" class=\"chessBoardTable\" \" +
            \" data-ng-class=\"{chessBoardAsBlack:white==false}\"><thead> \" +
            \"<tr><th></th><th>A</th><th>B</th><th>C</th> \" +
            \"<th>D</th><th>E</th><th>F</th> \" +
            \"<th>G</th><th>H</th></tr></thead><tbody>\";
        for( var i=0;i<boardPosition.positions.length;i++){
            var pos=boardPosition.positions[i];
            if(i==0 || i % 9==0){
                tableBuilder += "<tr>";
            }
            //Browsers collapses rows that does not have piece image on them.
            //Hidden image for every <tr> -> size of the board stays the same.
            tableBuilder += "<td data-grid=\"\" +
                +pos.coordinates+\" \" data-piece-type=\"\" +
                +pos.pieceType+\" \" data-piece-moves=\"\" +
                +pos.availableMoves+\" \" data-piece-color=\"\" +
                +pos.colorAbr+\" \" data-ng-mousedown=\"antsaChessBoard.handleSquareClick($event)\">\";
            var hidden= (i==0||i%9==0)?'visibility:hidden:':'';
            tableBuilder += "<div class=\"pieceContainer \">\" +
                \"<img style=\"\"+hidden+\" \" class=\"img-responsive\" src=\"\"+pos.imageUrl+\"\">\";
            tableBuilder += "</td>\";
            //determining <tr> endings, just beautiful
            if(i==8||i==17||i==26||i==35||i==44||i==53||i==62||i==71){
                tableBuilder += "</tr>\";
            }
        }
        tableBuilder += ("</tr>");

        tableBuilder += ("</tbody></table>");
        tableAsString = tableBuilder.toString();
        return tableBuilder;
    }
}

```

```

return {
  handleSquareClick : function(event) {
    var clickedTableCell = event.currentTarget;
    $('#messageArea').html("");
    if (!inTurn || timeout!=undefined||spectator) {
      console.log("not own turn");
      return;
    }
    // User clicked square x, there is no previous selection
    if (selectedSquare == null) {
      // if clicked square contains own piece let's choose it
      if (clickedTableCell
        .getAttribute('data-piece-color') == pColor) {
        if (lastMovedSquare != null) {
          lastMovedSquare.style.border = 'none';
        }
        selectedSquare = clickedTableCell;
        selectedSquareBackUpColor = selectedSquare.style.backgroundColor;
        selectedSquare.style.backgroundColor = "green";
      }
      return;
    }
    // Square has been selected already but now we make new selection
    if (clickedTableCell.getAttribute('data-piece-color') == pColor) {
      selectedSquare.style.backgroundColor = selectedSquareBackUpColor;
      selectedSquare = clickedTableCell;
      selectedSquareBackUpColor = selectedSquare.style.backgroundColor;
      selectedSquare.style.backgroundColor = "green";
      return;
    }
    // selected square presents square to where move is targeted .
    // server has sent all available squares for all existing pieces

    if (this.isAllowedMove(selectedSquare, clickedTableCell)
      && this.isPromotionMove(selectedSquare, clickedTableCell)) {
      var tableController =
        angular.element(document.querySelector(".gameboard")).scope();
      tableController
        .showPromotionModal(
          selectedSquare.getAttribute("data-grid"),clickedTableCell);
    }else if (this.isAllowedMove(selectedSquare, clickedTableCell)) {
      var tableController =
        angular.element(document.querySelector(".gameboard")).scope();
      tableController
        .sendMove(selectedSquare.getAttribute("data-grid"),
          clickedTableCell.getAttribute("data-grid"));
    } else {
      $('#messageArea').html("ei ole sallittu siirto ");
    }
    return selectedSquare;
  },
},

```

```

updateAvailableMoves : function(availables) {
    //<table class="chessBoardTable ...>
    var cTable = jQuery('.chessBoardTable');
    var allTds = jQuery(cTable).find('td');
    jQuery(allTds).each(function() {
        jQuery(this).attr('data-piece-moves', '');
    });
    for (var i = 0; i < availables.length; i++) {
        var allAvailableMoves = "";
        for (var j = 0; j < availables[i].availableMoves.length; j++) {
            allAvailableMoves += availables[i].availableMoves[j];
            if (j != (availables[i].
                availableMoves.length - 1)) {    allAvailableMoves += ",";
            }
        }
        var correctTd = jQuery(cTable)
            .find('[data-grid="' + availables[i].piecePosition + '\"]');
        var td = jQuery(".chessBoardTable td")
            .find('[data-grid="' + availables[i].piecePosition + '\"]');
        jQuery(correctTd).attr("data-piece-moves", allAvailableMoves);
    }
},
isAllowedMove : function(fromTd, toTd) {
    var allowedMoves = fromTd.getAttribute("data-piece-moves");
    var toPos = toTd.getAttribute("data-grid");
    return allowedMoves.indexOf(toPos) > -1;
},
...

remove:function(td){
    var toTd = document.
        querySelector(".chessBoardTable td[data-grid='" + td+ "\"]");
    // FirstElement child is an image.
    if (toTd.firstChild != null) {
        // removing opponent piece from "to" td
        toTd.removeChild(toTd.firstChild);
    }
},

```

```

isPromotionMove: function(from,to){
    if(pColor=="w"){
        var row = to.getAttribute('data-grid').substring(0, 1);
        var pieceType = from.getAttribute('data-piece-type');
        if(pieceType=="pawn" && row[0]=="0"){
            return true;
        }
    }else {
        var row = to.getAttribute('data-grid').substring(0, 1);
        var pieceType = from.getAttribute('data-piece-type');
        if(pieceType=="pawn" && row[0]=="7"){
            return true;
        }
    }
    return false;
},
move : function(fromTd, toTd) {
    if(lastMovedSquare!=null){
        lastMovedSquare.style.backgroundColor = lastMovedSquareBackUpColor;
    }
    var fromTd = document
    .querySelector(".chessBoardTable td[data-grid='"+ fromTd + "']");
    var toTd = document
    .querySelector(".chessBoardTable td[data-grid='"+ toTd+ "']");
    // FirstElement child is image.
    if (toTd.firstChild != null) {
        toTd.removeChild(toTd.firstChild);
    }
    toTd.appendChild(fromTd.firstChild);
    toTd.setAttribute("data-piece-color", fromTd
        .getAttribute("data-piece-color"));
    toTd.setAttribute("data-piece-type", fromTd
        .getAttribute("data-piece-type"));
    fromTd.setAttribute("data-piece-color", null);
    fromTd.setAttribute("data-piece-type", null);
    if (selectedSquare != null) {
        // Opponent has not selected
        selectedSquare.style.backgroundColor = selectedSquareBackUpColor;
    }
    lastMovedSquare = toTd;
    lastMovedSquareBackUpColor = lastMovedSquare.style.backgroundColor;
    lastMovedSquare.style.backgroundColor = 'yellow';
    selectedSquare = null;
    return toTd;
},

```


Example of Move -command sent from backend

```

JSON      {message: {area: "TABLE", command: "MOVE", moveFrom: "71",
message    more...}}
  message {area: "TABLE", command: "MOVE", moveFrom: "71", more...}
    area  "TABLE"
    command "MOVE"
    moveFrom "71"
    moveTo "50"
    timeLeftWhite "300"
    timeLeftBlack "300"
    colorInTurn "b"
    availableMoves {0: {piecePosition: "01", availableMoves: {0: "20", 1: "22"}}, 1:
0              {piecePosition: "06", availableMoves: {0: "25", 1: "27"}}, 2:
1              {piecePosition: "10", availableMoves: {0: "20", 1: "30"}}, more...}
2              {piecePosition: "01", availableMoves: {0: "20", 1: "22"}}
3              {piecePosition: "06", availableMoves: {0: "25", 1: "27"}}
4              {piecePosition: "10", availableMoves: {0: "20", 1: "30"}}
5              {piecePosition: "11", availableMoves: {0: "21", 1: "31"}}
6              {piecePosition: "12", availableMoves: {0: "22", 1: "32"}}
7              {piecePosition: "13", availableMoves: {0: "23", 1: "33"}}
8              {piecePosition: "14", availableMoves: {0: "24", 1: "34"}}
9              {piecePosition: "15", availableMoves: {0: "25", 1: "35"}}
            {piecePosition: "16", availableMoves: {0: "26", 1: "36"}}
            {piecePosition: "17", availableMoves: {0: "27", 1: "37"}}

```

ChessRequest.java

```
package fi.antsa.chess.server.use;
import java.io.Serializable;
public final class ChessRequest<T> implements Serializable {
    private static final long serialVersionUID = 1L;
    private final ChessMessage<T> chessMessage;
    public ChessRequest(ChessMessage<T> chessMessage) {
        this.chessMessage = chessMessage;
    }
    public ChessMessage<T> getChessMessage() {
        return chessMessage;
    }
    public ChessMessageHeader getMessageHeader() {
        return chessMessage.getMessageHeader();
    }
    public boolean isLogoutMessage() {
        return this.chessMessage.getMessageHeader()
            == ChessMessageHeader.LOGOUT;
    }
    public boolean isLoginMessage() {
        return this.chessMessage.getMessageHeader()
            == ChessMessageHeader.LOGIN;
    }
    public boolean isForOne() {
        return this.chessMessage.getMessageHeader().getRecipients()
            == Recipients.ONE;
    }
    public boolean isForGroup() {
        return this.chessMessage.getMessageHeader().getRecipients()
            == Recipients.GROUP;
    }
    @Override
    public String toString() {
        StringBuilder builder = new StringBuilder();
        builder.append("ChessRequest [chessMessage=");
        builder.append(chessMessage);
        builder.append("]");
        return builder.toString();
    }
    public boolean isForEverybodyElseBut() {
        return this.chessMessage.getMessageHeader().getRecipients()
            == Recipients.EXCLUDED;
    }
}
```

ChessMessage.java

```
package fi.antsa.chess.server.use;
import java.io.Serializable;

public final class ChessMessage<T> implements Serializable {
    private static final long serialVersionUID = 1L;
    private ChessMessageHeader messageHeader;
    private final ChessMessageToken token;
    private final T params;

    public ChessMessage(ChessMessageHeader messageHeader,
        ChessMessageToken messageToken, T params) {
        this.messageHeader = messageHeader;
        this.params = params;
        this.token = messageToken;
    }

    public ChessMessageHeader getMessageHeader() {
        return messageHeader;
    }

    public <T> T getParams() {
        return (T) params;
    }

    public boolean isMoveResult(){
        return messageHeader
            !=ChessMessageHeader.RESTART_GAME_AS_SPECTATOR;
    }

    @Override
    public String toString() {
        return "ChessMessage [messageHeader=" + messageHeader
            + ", token=" + token + ", params=" + params + "];"
    }

    public ChessMessageToken getToken() {
        return token;
    }

    public void setMessageHeader(ChessMessageHeader messageHeader) {
        this.messageHeader = messageHeader;
    }
}
```

ChessMessageHeader.java

```

package fi.antsa.chess.server.use;
public enum ChessMessageHeader {
    LOGIN("login", Recipients.INCOMING_ONLY, Area.USER),
    JOIN_TABLE("join_table", Recipients.INCOMING_ONLY, Area.LOBBY),
    LOGIN_OK("login_ok", Recipients.ONE, Area.USER),
    UNPARSEABLE("fail", Recipients.ONE, null),
    LOGIN_FAILED("login_fail", Recipients.ONE, Area.USER),
    LOGOUT("logout", Recipients.ALL, Area.LOBBY),
    CREATE_TABLE("create_table", Recipients.ALL, Area.LOBBY),
    REMOVE_TABLE("remove_table", Recipients.ALL, Area.LOBBY),
    ADD_TABLE("add_table", Recipients.ALL, Area.LOBBY),
    GAME_STARTED("game_started", Recipients.ALL, Area.LOBBY),
    NEW_PLAYER("newplayer", Recipients.EXCLUDED, Area.LOBBY),
    START_GAME_AS_WHITE("start_white", Recipients.ONE, Area.TABLE),
    START_GAME_AS_BLACK("start_black", Recipients.ONE, Area.TABLE),
    START_GAME_AS_SPECTATOR("start_spectator", Recipients.ONE,
        Area.TABLE),
    RESTART_GAME_AS_SPECTATOR("restart_spectator",
        Recipients.GROUP, Area.TABLE),
    PROMOTION("promotion", Recipients.INCOMING_ONLY, Area.TABLE),
    MOVE("move", Recipients.GROUP, Area.TABLE),
    MOVE_CASTLING("move", Recipients.GROUP, Area.TABLE),
    PROMOTE_PIECE("promote_piece", Recipients.GROUP, Area.TABLE),
    PASSANT("passant", Recipients.GROUP, Area.TABLE),
    MOVE_NOT_ALLOWED("move_not_allowed", Recipients.ONE, Area.TABLE),
    WATCH("watch_table", Recipients.INCOMING_ONLY, Area.TABLE),
    TIMEOUT("timeout", Recipients.GROUP, Area.TABLE),
    SUGGEST_NEW_GAME("new_game", Recipients.ONE, Area.TABLE),
    RETURN_TO_LOBBY("return_lobby", Recipients.INCOMING_ONLY, Area.TABLE),
    RESIGN("resign", Recipients.GROUP, Area.TABLE),
    OFFER_DRAW("offer_draw", Recipients.INCOMING_ONLY, Area.TABLE),
    CHAT_MESSAGE("message", Recipients.GROUP, Area.TABLE);

    private final String commandAsText;
    private final Recipients recipients;
    private final Area module;
    private ChessMessageHeader(String commandAsText, Recipients r, Area m) {
        this.commandAsText = commandAsText;
        this.recipients = r;
        this.module = m;
    }
    public String getAsText() {
        return this.commandAsText;
    }
    public Recipients getRecipients() {
        return recipients;
    }
    public Area getModule() {
        return this.module;
    }
    public static ChessMessageHeader getCommand(String c) {
        if (c == null) {
            return null;
        }
        for (ChessMessageHeader com : ChessMessageHeader.values()) {
            if (c.equals(com.getAsText())) {
                return com;
            }
        }
        return null;
    }
}

```

ChessMessageToken.java

```

package fi.antsa.chess.server.use;
import java.io.Serializable;
import java.util.UUID;
public final class ChessMessageToken implements Serializable {
    private static final long serialVersionUID = 1L;
    private String authToken;
    private String loginHelp;
    public String getLoginHelp() {
        return loginHelp;
    }
    public void setLoginHelp(String loginHelp) {
        this.loginHelp = loginHelp;
    }
    public ChessMessageToken(String authToken) {
        this.authToken = authToken;
    }
    public String getAuthToken() {
        return authToken;
    }
    public String asText() {
        return authToken;
    }
    public UUID asUUID() {
        return UUID.fromString(authToken);
    }
    public void setAuthToken(String authToken) {
        this.authToken = authToken;
    }
    @Override
    public String toString() {
        StringBuilder builder = new StringBuilder();
        builder.append("ChessMessageToken [authToken=")
            .append(authToken)
            .append(", loginHelper=").append(loginHelp).append("]");
        return builder.toString();
    }
    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result
            + ((authToken == null) ? 0 : authToken.hashCode());
        return result;
    }
    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        ChessMessageToken other = (ChessMessageToken) obj;
        if (authToken == null) {
            if (other.authToken != null)
                return false;
        } else if (!authToken.equals(other.authToken))
            return false;
        return true;
    }
}

```

Tomcat.java

```

package fi.antsa.chess.server.client;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.Socket;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.logging.Level;
import java.util.logging.Logger;
import fi.antsa.chess.server.handler.WebClientMessageHandler;
import fi.antsa.chess.server.use.ChessMessage;
import fi.antsa.chess.server.use.ChessRequest;
public class Tomcat implements ChessServerClient {
    private static final Logger LOGGER =
        Logger.getLogger(Tomcat.class.getName());
    private ObjectInputStream in;
    private ObjectOutputStream out;
    private ExecutorService executorService;
    //Called by ChessServer after it has got connection WebServer
    public void init(Socket chessServerSocket) {
        // Smaller amount of threads would be enough
        executorService = Executors.newFixedThreadPool(10);
        try {
            this.out =
                new ObjectOutputStream(chessServerSocket.getOutputStream());
            this.in =
                new ObjectInputStream(chessServerSocket.getInputStream());
        } catch (Exception e) {
            LOGGER.log(Level.SEVERE, "Error in initialisation:", e);
        }
    }
    @Override
    public void pushToClient(ChessRequest request) {
        LOGGER.finer("Tomcat pushing timeout message to WebServer");
        executorService.execute(new WebClientMessageHandler(out, request));
    }
    @Override
    public void run() {
        while (true) {
            try {
                ChessRequest<ChessMessage> request =
                    (ChessRequest<ChessMessage>) in.readObject();
                executorService.execute(new WebClientMessageHandler(out, request));
            } catch (ClassNotFoundException | IOException e) {
                LOGGER.log(Level.SEVERE, "Tomcat runtime error ", e);
                return;
            }
        }
    }

    public ObjectOutputStream getOut(){
        return this.out; }
    @Override
    public ChessClientType getClientType() {
        return ChessClientType.WEBSERVER;
    }
}

```

ChessServer.java code peek

```

@Override
public ChessPlayer login(ChessRequest<ChessMessage> request) {
    ChessMessage message = request.getChessMessage();
    Map<String, String> params = (Map<String, String>) message.getParams();
    String nick = validateLogin(params);
    if (nick != null) {
        //Java 8 style streaming. Checks if nick exist already.
        ChessPlayer possiblePlayer = clientMap.entrySet().stream()
            .filter(e -> e.getValue().getNick().equals(nick)).
            map(Map.Entry::getValue).findFirst().orElse(null);
        if (possiblePlayer == null) {
            //WebServer has assigned this player to UUID, same is used.
            //Better would be to create the UUID here for player
            ChessPlayer cp = new ChessPlayer(nick, message.getToken().asUUID());
            if (this.clientMap
                .putIfAbsent(cp.getPlayerUtils().getPlayerId(), cp) == null) {
                //Login is ok ChessPlayer has been created
                return cp;
            }
        }
        // No login was done
        return null;
    }

private String validateLogin(Map<String, String> params) {
    String nick = params.get(AntsaChessConstants.PARAMETER_0);
    if (nick == null) {
        return null;
    }
    nick = nick.trim();
    if (nick.length() == 0) {
        return null;
    }
    if (nick.length() > 10) {
        nick = nick.substring(0, 10);
    }
    // html/javascript chars now possible to inject. Should validate more carefully
    params.put(AntsaChessConstants.PARAMETER_0, nick);
    return nick;
}

```

```

@Override
public Table createTable(String selectedTimeOptionNumber, UUID creatorId) {
    ChessPlayer cp = clientMap.get(creatorId);
    if (cp == null) {
        LOGGER.severe("can't create table to non existing player " + creatorId);
        return null;
    }
    for (Map.Entry<UUID, Table> entry : tables.entrySet()) {
        ChessPlayer cplayer = entry.getValue().getPlayerA();
        if (creatorId.equals(cplayer.getPlayerUtils().getPlayerId())) {
            LOGGER.info("Nick tried to create second table " + cplayer.getNick());
            return null;
        }
    }
    Table t = new Table(cp, Timelimits.getGameTimeInSeconds(selectedTimeOptionNumber));
    cp.setSitsInTable(t.getTableId());
    tables.put(t.getTableId(), t);
    return t;
}

// returns table only if 2 players at the table
@Override
public synchronized Table joinTable(UUID joinerPlayerId, String tableId) {
    Table chessTable = tables.get(UUID.fromString(tableId));
    if (chessTable == null || chessTable.isOnGoing()) {
        LOGGER.info("Not possible to join:" + chessTable
            + " " + tableId + " onGoing: " + chessTable.isOnGoing());
        return null;
    }
    ChessPlayer joinerPlayer = clientMap.get(joinerPlayerId);
    if (joinerPlayer == null) {
        LOGGER.info("no joining player in clients " + joinerPlayerId);
        return null;
    }
    joinerPlayer.setSitsInTable(chessTable.getTableId());
    chessTable.startGame(joinerPlayer);
    chessTable.getGameSession()
        .getGame().getChessBoardWrapper().updateAvailableMoves(Color.WHITE);
    return chessTable;
}

```



```

@Override
public Table watchTable(UUID watcherId, UUID tableId) {
    Table table = tables.get(tableId);
    table.addSpectator(clientMap.get(watcherId));
    return table;
}

//Called when player or spectator leaves table or closes browser
@Override
public UUID removeFromTable(UUID returnerId) {
    ChessPlayer cp = clientMap.get(returnerId);
    if (cp == null) {
        return null;
    }
    UUID tableId = cp.getSitsInTable();
    if (tableId != null) {
        Table table = tables.get(tableId);
        // can be null if other player has left before
        if (table != null && table.isPlayerAOrPlayerB(cp)) {
            //table is internally removed if playerA or playerB left
            //other player and spectators still see the table in UI
            //prevents others from joining anymore since there is no game
            removeTable(returnerId);
            return tableId;
        } else if (table != null) {
            //Spectator left
            table.removeFromTable(cp);
        }
    }
    return null;
}

```

```

@Override
public MoveResult move(CheeseRequest chessRequest) {
    UUID playerIdOfSender = chessRequest.getChessMessage().getToken().asUUID();
    ChessPlayer cp = clientMap.get(playerIdOfSender);
    Table table = tables.get(cp.getSitsInTable());
    Game game = table.getGameSession().getGame();
    Map<String, String> moves =
        (Map<String, String>) chessRequest.getChessMessage().getParams();
    String from = moves.get(AntsaChessConstants.PARAMETER_0);
    String to = moves.get(AntsaChessConstants.PARAMETER_1);
    MoveResult moveResult = game.move(cp, from, to);
    // everybody at the table needs update of the move -> WebServer delivers.
    moveResult.setAllInTable(table.getAllInTable());
    moveResult.setAbrvColorInTurn(game.getAbbreviatedColorInTurn());
    moveResult.setPlayerWhoMadeMove(cp.getPlayerUtils().getPlayerId());
    // UI does not currently change UI chess clock based on what server returns
    moveResult.setTimeLeftWhite((int) (game.getWhitePlayer().getTimeLeft() / 1000));
    moveResult.setTimeLeftBlack((int) (game.getBlackPlayer().getTimeLeft() / 1000));
    return moveResult;
}

@Override
public MoveResult promote(CheeseRequest chessRequest) {
    UUID playerIdOfSender = chessRequest.getChessMessage().getToken().asUUID();
    ChessPlayer cp = clientMap.get(playerIdOfSender);
    Table table = tables.get(cp.getSitsInTable());
    Game game = table.getGameSession().getGame();
    Map<String, String> moves =
        (Map<String, String>) chessRequest.getChessMessage().getParams();
    String from = moves.get(AntsaChessConstants.PARAMETER_0);
    String to = moves.get(AntsaChessConstants.PARAMETER_1);
    String pieceType = moves.get(AntsaChessConstants.PARAMETER_2);
    //should also return time lefts if UI would change the clock based server
    MoveResult moveResult = game.promote(cp, from, to, pieceType);
    moveResult.setAllInTable(table.getAllInTable());
    moveResult.setAbrvColorInTurn(game.getAbbreviatedColorInTurn());
    return moveResult;
}

```

```

@Override
public Logout logout(ChessRequest<ChessMessage> chessRequest) {
    ChessPlayer playerToBeRemoved = clientMap
        .get(UUID.fromString(chessRequest.getChessMessage().getToken().asText()));
    if (playerToBeRemoved != null) {
        String nick = playerToBeRemoved.getNick();
        UUID playerToken = playerToBeRemoved.getPlayerUtils().getPlayerId();
        clientMap.remove(playerToBeRemoved.getPlayerUtils().getPlayerId());
        // In case player who left had created a table -> remove
        Table removedTable = removeTable(playerToken);
        if (removedTable != null) {
            //Other players need to remove nickname and created table from UI
            return new Logout(nick, removedTable.getTableId());
        }
        //Only nickname needs to be removed from UI.
        return new Logout(nick, null);
    } else {
        // Remove was not made
        return null;
    }
}

@Override
public Table suggestNewGame(ChessRequest chessRequest) {
    UUID playerWhoSuggestsNewGame =
        UUID.fromString(chessRequest.getChessMessage().getToken().asText());
    UUID tableId = getPlayerTableId(playerWhoSuggestsNewGame);
    if (tableId == null) {
        return null;
    }
    Table table = getTable(tableId);
    boolean startNewGame = table
        .suggestNewGame(clientMap.get(playerWhoSuggestsNewGame));
    if (startNewGame) {
        table.startRematch();
        table.getGameSession().getGame().getChessBoardWrapper()
            .updateAvailableMoves(Color.WHITE);
        return table;
    }
    return null;
}

@Override
public Table offerDraw(ChessRequest chessRequest) {
    UUID playerWhoSuggestsDraw =
        UUID.fromString(chessRequest.getChessMessage().getToken().asText());
    UUID tableId = getPlayerTableId(playerWhoSuggestsDraw);
    if (tableId == null) {
        return null;
    }
    Table table = getTable(tableId);
    boolean gameIsDraw = table.suggestDraw(clientMap.get(playerWhoSuggestsDraw));
    if (gameIsDraw) {
        //draw by agreement by both players
        table.getGameSession().getGame().getTurn().stopTimer();
        return table;
    }
    return null;
}
}

```

WebClientMessageHandler.java code peek

```
1 package fi.antsa.chess.server.handler;
2
3 import java.io.IOException;
4 import java.io.ObjectOutputStream;
5 import java.text.SimpleDateFormat;
6 import java.util.Calendar;
7 import java.util.List;
8 import java.util.Map;
9 import java.util.UUID;
10 import java.util.logging.Logger;
11
12 import fi.antsa.chess.server.ChessServer;
13 import fi.antsa.chess.server.Logout;
14 import fi.antsa.chess.server.game.ChessBoardUtils;
15 import fi.antsa.chess.server.game.ChessPlayer;
16 import fi.antsa.chess.server.game.Table;
17 import fi.antsa.chess.server.use.AntsaChessConstants;
18 import fi.antsa.chess.server.use.BoardPosition;
19 import fi.antsa.chess.server.use.ChessMessage;
20 import fi.antsa.chess.server.use.ChessMessageHeader;
21 import fi.antsa.chess.server.use.ChessMessageToken;
22 import fi.antsa.chess.server.use.ChessRequest;
23 import fi.antsa.chess.server.use.GameActionResult;
24 import fi.antsa.chess.server.use.GameStart;
25 import fi.antsa.chess.server.use.MoveResult;
26 import fi.antsa.chess.server.use.TableInfo;
27 import fi.antsa.chess.server.use.Target;
28 import fi.antsa.chess.server.use.UsersAndTables;
29
30 public class WebClientMessageHandler implements Runnable {
31
32     private ChessRequest chessRequest;
33     private ObjectOutputStream out;
34     private static final String CLZ_NAME = WebClientMessageHandler.class.getName();
35     private static final Logger LOGGER = Logger.getLogger(CLZ_NAME);
36
37     public WebClientMessageHandler(ObjectOutputStream out, ChessRequest request) {
38         // message should never be null
39         if (request == null) {
40             throw new IllegalArgumentException("Message is null");
41         }
42         this.out = out;
43         this.chessRequest = request;
44     }
45 }
```

```

46 @Override
47 public void run() {
48     LOGGER.fine("Handler starts to handle message:" + chessRequest.getMessageHeader());
49     switch (chessRequest.getMessageHeader()) {
50     case LOGIN:
51         ChessPlayer player = ChessServer.INSTANCE.login(chessRequest);
52         Map<String, String> params =
53             (Map<String, String>) chessRequest.getChessMessage().getParams();
54         if (player != null) {
55             List<String> userNames = ChessServer.INSTANCE.getUserNames();
56             UsersAndTables u =
57                 new UsersAndTables(userNames, ChessServer.INSTANCE.getTableInfos(),
58                     params.get(AntsaChessConstants.PARAMETER_0));
59             ChessMessageToken cm =
60                 new ChessMessageToken(player.getPlayerUtils().getPlayerId().toString());
61             cm.setLoginHelp(chessRequest.getChessMessage().getToken().getLoginHelp());
62             ChessMessage<UsersAndTables> messu =
63                 new ChessMessage<UsersAndTables>(ChessMessageHeader.LOGIN_OK, cm, u);
64             write(new ChessRequest<>(messu));
65             Target target = new Target(player.getPlayerUtils().getPlayerId());
66             target.setTargetName(player.getNick());
67             ChessMessage<Target> updateOthers =
68                 new ChessMessage<Target>(ChessMessageHeader.NEW_PLAYER, cm, target);
69             write(new ChessRequest<>(updateOthers));
70         } else {
71             String reservedNick = params.get(AntsaChessConstants.PARAMETER_0);
72             LOGGER.fine("nick reserved no login done " + reservedNick);
73             ChessMessage<String> messu =
74                 new ChessMessage<String>(ChessMessageHeader.LOGIN_FAILED,
75                     chessRequest.getChessMessage().getToken(), reservedNick);
76             write(new ChessRequest<>(messu));
77         }
78         break;

```

```

case MOVE:
    MoveResult moveResult = ChessServer.INSTANCE.move(chessRequest);
    ChessMessage<MoveResult> moveMessage = null;
    if (moveResult.getGameActionResult() == GameActionResult.MOVE_DONE) {
        moveMessage =
            new ChessMessage<MoveResult>(ChessMessageHeader.MOVE,
                new ChessMessageToken(null), moveResult);
    } else if (moveResult.getGameActionResult() == GameActionResult.CASTLING) {
        moveMessage =
            new ChessMessage<MoveResult>(ChessMessageHeader.MOVE_CASTLING,
                new ChessMessageToken(null), moveResult);
    } else if (moveResult.getGameActionResult() == GameActionResult.PASSANT_DONE) {
        moveMessage =
            new ChessMessage<MoveResult>(ChessMessageHeader.PASSANT,
                new ChessMessageToken(null), moveResult);
    } else if (moveResult.getGameActionResult() == GameActionResult.TIMEOUT) {
        moveMessage =
            new ChessMessage<MoveResult>(ChessMessageHeader.TIMEOUT,
                new ChessMessageToken(null), moveResult);
    } else {
        moveMessage =
            new ChessMessage<MoveResult>(ChessMessageHeader.MOVE_NOT_ALLOWED,
                new ChessMessageToken(null), moveResult);
    }
    write(new ChessRequest<>(moveMessage));
    break;
case OFFER_DRAW:
    Table drawTable = ChessServer.INSTANCE.offerDraw(chessRequest);
    //First player suggests draw
    if (drawTable == null) {
        return;
    } //both players have suggested draw. MoveResult used for convenience
    MoveResult drawResult = new MoveResult("", "");
    drawResult.setDrawByAgreement(true);
    drawResult.setAllInTable(drawTable.getAllInTable()); //includes table spectators
    ChessMessage<MoveResult> drawMessage = new ChessMessage<MoveResult>(ChessMessageHeader.OFFER_DRAW,
        new ChessMessageToken(null), drawResult);
    write(new ChessRequest<>(drawMessage));
    break;
default:
    break;
}

public void write(ChessRequest cm) {
    try {
        // if not synchronized then sending timeout causes "stream active" exception
        synchronized (out) {
            out.reset();
            out.writeObject(cm);
        }
    } catch (IOException e) {
        LOGGER.log(Level.SEVERE, "WebClientMessageHandler failed sending message ", e);
    }
}
}

```