Karelia
AMMATTIKORKEAKOULU

Technical specification

WormGame

Version 1.0

Karelia	TIKO	LTP7024 Testausmenetelmät
Author: Juhani Pirinen		Printed: 7th oct 2015
Author: Teacher of the course, Public in GitHub		
Status of the document: draft		Edited: 7th oct 2015

VERSION HISTORY

Version	Date	Authors	Description
1.0	7th oct 2015	Juhani Pirinen	First version; only those server side features are documented in detail, that will be tested in this course

1. PREFACE

1.1 Purpose and extent

This is a technical specification of a multi-player worm game, written for needs of testing method course. The specification will be very limited, when scope and schedule of the course is limited. Only those parts of software, that will be tested on the course, are documented more in detail. This document will complement and extend specifications of system's requirements specification to a more technical level.

The game is already developed, in network applications development course, before writing this document. A very detailed ”waterfall-type” specification like this wasn't required in application development course, where a more agile approach was used, hence now things go in wrong order and writing this document for development purposes would make no sense. However, this document is needed in testing methods course, anyway.

1.2 Product and environment

This is a multi-player online version of the classic worm game, that will run as client-server application in web browser and web server. Only authenticated users can play the game. Additionally users can chat and see ranking lists.

Client is a web browser, that has support to HTML5 and ECMAscript version 5 of javascript. Game server runs in Node.JS framework, that users Express.JS framework, MySQL database server and Socket.IO javascript library for using web sockets protocol.

1.3 Definitions, notations, abbrevations

API	Application Protocol Interface
Client-Server	Archit. of two systems communicating over network
Ecmascript5	Edition 5 of JavaScript language
ExpressJS	NodeJS web application server framework
HTML5	Hypertext Markup Language, version 5
HTTP	Hypertext Transfer Protocol
TCP	Transmission Control Protocol
MySQL 5.5	MySQL relational database server version 5.5
NodeJS	Runtime environment for server-side web applications
Socket.IO	JavaScript library for realtime web applications
Websockets	Bi-directional communication protocol over TCP
WormGame	Multi-Player version of classic Snake video game

1.4 References

Automattic. Socket.IO library. http://www.socket.io 3rd oct 2015.

Ecma International. ECMAScript 5.1. http://www.ecma-international.org/ecma-262/5.1/ 3rd oct 2015.

IETF. Hypertext Transfer Protocol, HTTP/1.1. RFC2616. https://www.ietf.org/rfc/rfc2616.txt 3rd oct 2015.

IETF. The WebSocket Protocol. RFC6455. https://www.ietf.org/rfc/rfc6455.txt 3rd oct 2015.

Node.js foundation. NodeJS environment. https://nodejs.org 3rd oct 2015.

Oracle Corp. MySQL database server. https://www.mysql.com/ 3rd oct 2015

Pivotal Labs. Jasmine - a Behavior Driven Development testing framework for JavaScript. http://jasmine.github.io/ 3rd oct 2015.

StrongLoop, Inc. ExpressJS framework. http://expressjs.com/ 3rd oct 2015.

Wikipedia. Snake (video game). https://en.wikipedia.org/wiki/Snake_%28video_game%29 3rd oct 2015.
Word Wide Web Consortium. Cascading Style Sheets, CSS. http://www.w3.org/Style/CSS/ 3rd oct 2015.
Word Wide Web Consortium. Hypertext Markup Language, HTML5. http://www.w3.org/TR/html5/ 3rd oct 2015.

1.5 Overview to the document

The document begins with preface and overview to the system and continue with architecture description. In last part, components of the software, including their API interfaces, are specified.

2 OVERVIEW TO THE SYSTEM

2.1 Description of application area

A small classic video game that anyone can play.

2.2 System's association to it's environment

The game can be played by using a modern web browser. The game server can run in any server, that supports it's software environment.

2.3 Device environment

The game requires very little of resources. Laptop or desktop computer is enough. The game is not designed to mobile or tablet devices. Using the game in LAN is recommended, because of low security level of the prototype.

2.4 Software environment

Server environment:
MySQL server 5 or newer
NodeJS 0.10.40 or newer
Npm 1.4.28 or newer
Npm packages: ExpressJS ~4, MySql ~2.7.0, Body-parser ~1.13.0, Socket.IO ~1.3.0

Client environment:
Internet explorer 10 or newer
Edge 12 or newer
Firefox 38 or newer

Chrome 31 or newer
Safari 8 or newer

Development environment:
Any personal computer, that can run both server and client environments
Git repository and git client software
Any code editor that supports javascript and html markup
Libre Office for documentation
Visual Paradigm, MS Visio or similar for documentation

Production environment:
This prototype will not be used in production. Please run it in development environment only.

2.5 The main boundary conditions of implementation

Implementation must be limited and follow the course schedules, and meet the course requirements. All documentation is written to adapt this.

3 ARCHITECTURE

3.1 Design principles

Design of this system should follow object oriented programming paradigm. All implementations should be made as modular as possible.

3.2 Software architecture, modules and processes

This software is a distributed system, that consists of separate client and server components, that are together needed to fulfill the requirements. Client is like a dashboard interface or control panel to the game, when server runs the actual game and has all the logic and resources. Both client and server are event-based asynchronous systems, written in JavaScript.

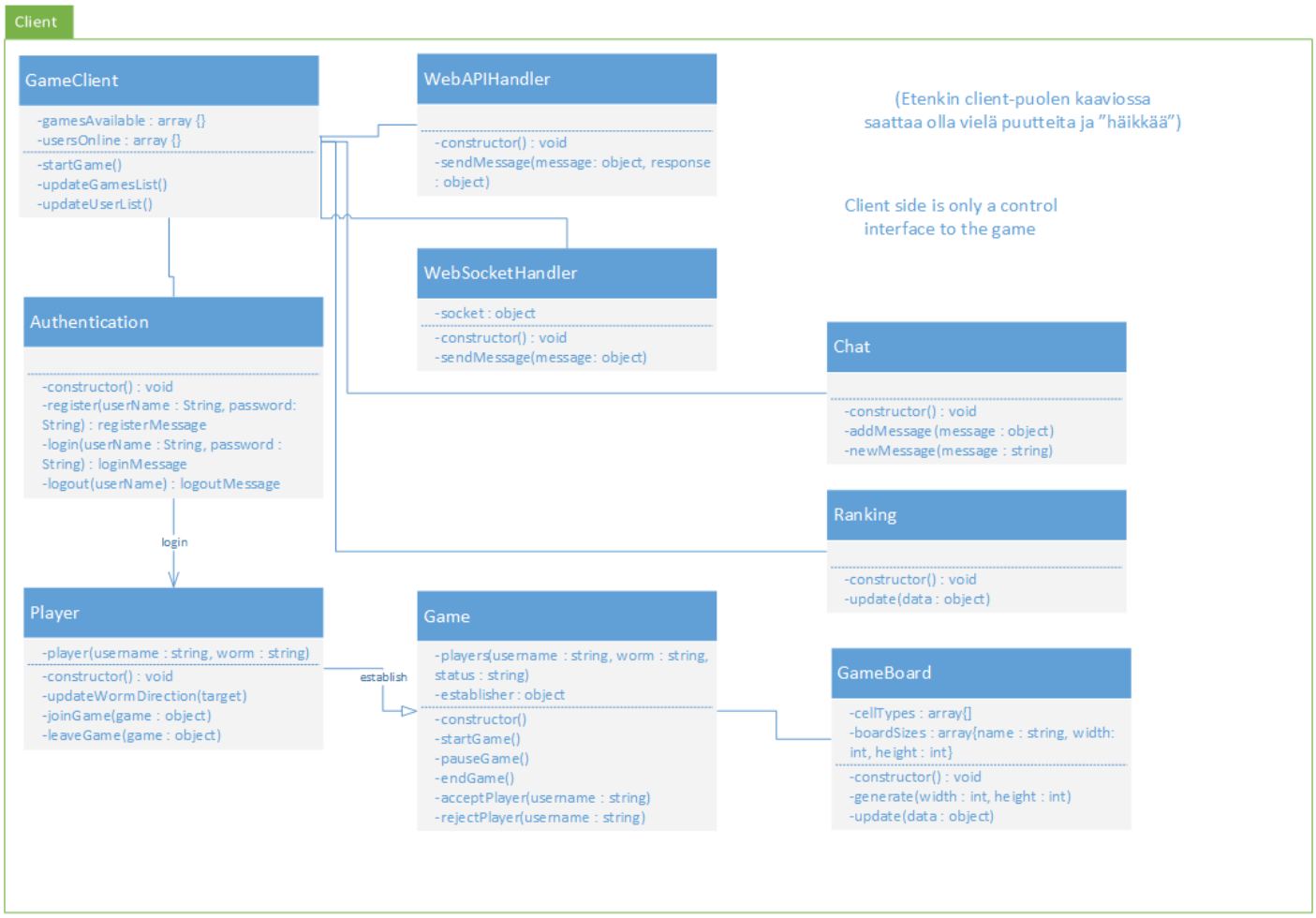
Amount of clients can be attuned according to the used server hardware, by default 30 clients are allowed. Server can have multiple game sessions and each game session can run with 1-4 clients. It might be good to limit amount of communication on server side, how much one client can push data, for example to 10 messages in second per client.

For learning purposes, both HTTP and WebSocket protocols are used for client-server communication; this is why there's also two APIs: HTTP rest API (half-duplex; client requests, server responses) and WebSocket API (full-duplex; client and server both are event-based server-like messengers, that push data to each others). When game is running, there's additionally a scheduled game loop executed on the server.

Code modularisation should be made also for parts of the software, that communicate with some external system, like database or client. Therefore using handler classes is proposed, for HTTP rest, Websockets and databases. Other classes should only use services, that they provide and and not bypass them, for example, when being in touch with databases.

The following class structures for client and server, can be improved during development process with agility:

3.2.1 Client classes overview



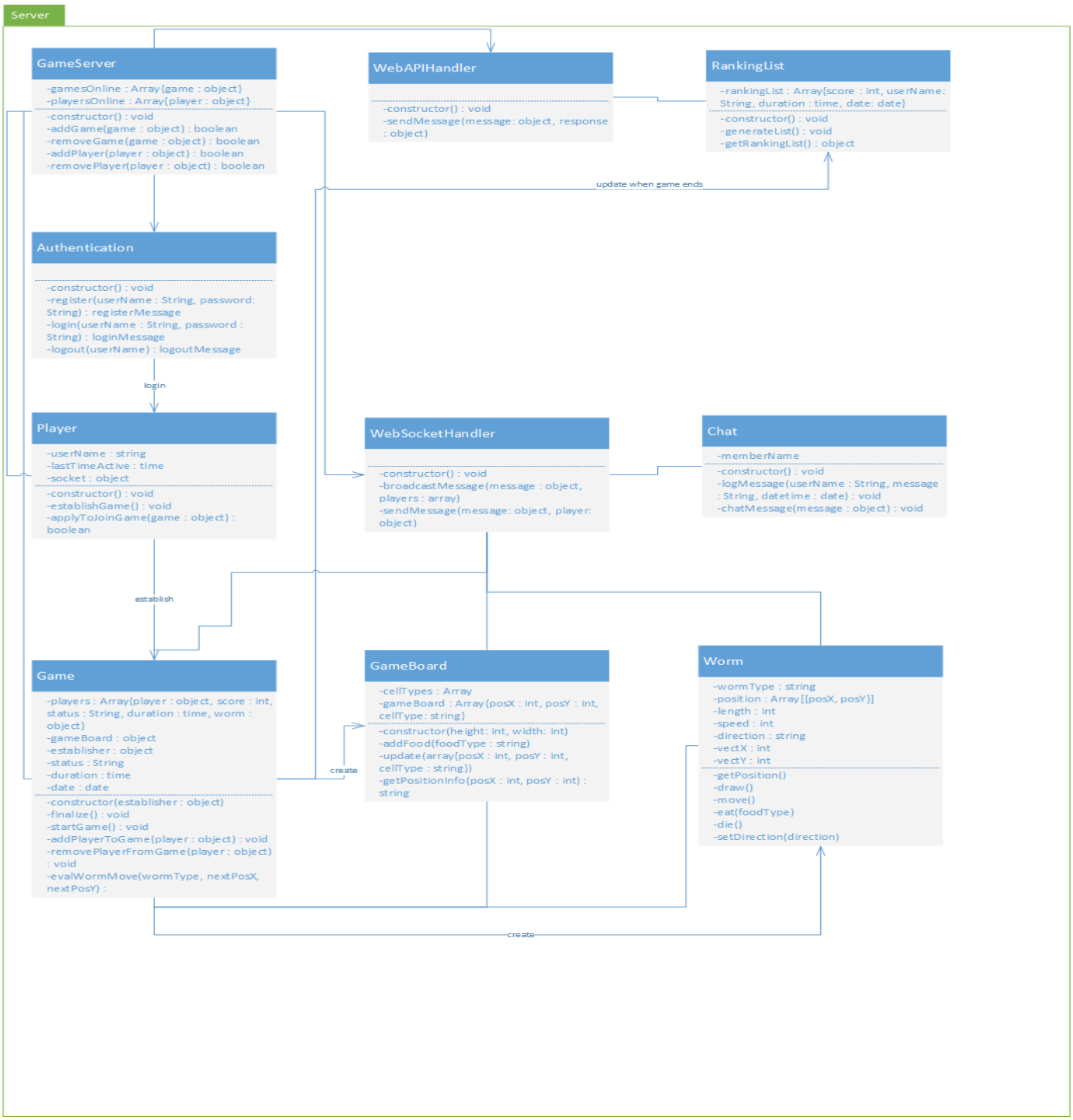
Class name	Description
GameClient	Manages existing games and users online.
Authentication	Authenticates the users against Rest API of the server. Takes care of logins and registrations.

Player	Each authenticated user is object of player class. It can join and leave existing games, and establish, remove, start and end games.
Game	Provides a game session, after it is established. State of game session can be established, running and end.
GameBoard	Generates a gameboard. Server pushes it's contents to clients. Client can determine size of gameboard, before game has started.
Worm	Worm is a player's ”avatar” that exists only, when game session is in running-state. Perhaps this could extend the player class? Not yet drawn to the diagram.
WebAPIHandler	A middleman, that makes requests to Rest API of server, and handles server's responses. Other classes just uses this. Perhaps this could be a class that implements an interface? Not yet drawn to the diagram.
WebSocketAPIHandler	A middleman, that uses Socket.IO, pushes data to server and handles what server has pushed. Other classes just uses this. Perhaps this

	could be a class that implements an interface? Not yet drawn to the diagram.
Chat	Manages everything related to chat in client's end. Also unregistered users have websocket connection and can chat.
Ranking	Manages everything related to ranking list in client's end.

For details about proposed attributes and methods, please see the client class diagram. This is an incomplete version of specification, and things may change during development project with agility.

3.2.2 Server classes overview



Class name	**Description**
GameServer	Manages existing games and users online.
Authentication	Authenticates the users against Rest API of the server. Takes care of logins and registrations.
Player	Each authenticated user is object of player class. It can join and leave existing games, and establish, remove, start and end games.
Game	Provides a game session, after it is established. State of game session can be established, running and end.
GameBoard	Generates a gameboard. Server pushes it's contents to clients. Client can determine size of gameboard, before game has started.
Worm	Worm is a player's ”avatar” that exists only, when game session is in running-state. Perhaps this could extend the player class? Not yet drawn to the diagram.

WebAPIHandler	A middleman, that offers resources of Rest API to the client, and generated responsesto client's requests. Other classes just uses this. Perhaps this could be a class that implements an interface? Not yet drawn to the diagram.
WebSocketHandler	A middleman, that uses Socket.IO, pushes data to clients and handles what client has pushed. Other classes just uses this. Perhaps this could be a class that implements an interface? Not yet drawn to the diagram.
DatabaseHandler	A middleman, that takes care of CURD operations to the databases. Only class that is in touch with databases and contains SQL. Other classes just uses this. Perhaps this could be a class that implements an interface? Not yet drawn to the diagram.
Chat	Manages everything related to chat in the server. Also unregistered users have a websocket connection and can chat.
RankingList	Manages everything related to

	ranking list in the server.

For details about proposed attributes and methods, please see the server class diagram. This is an incomplete version of specification, and things may change during development project with agility.

3.3 Database architecture

The game's need for databases is very little. Any SQL or NoSQL database would be fine, but MySQL was chosen for software environment by other students.

The following data structures are needed:

Table: Users			For saving registered users	
Column name	**Data type**	**Length**	**Allowed values**	**Notes**
Username (PK)	String	4-16	a-z, A-Z, 0-9	
Password	String	8-16	a-z, A-Z, 0-9	Should be encrypted
Highscore	Integer	0-65535	integers	
LastAccess	Datetime		in a format like dd.mm.yyyy hh:mm:ss	

Table: ChatLog			For saving chat messages	
Column name	**Data type**	**Length**	**Allowed values**	**Notes**
ID (PK)	Integer	0-4294967295	integers	
Sent	Datetime		in a format like dd.mm.yyyy hh:mm:ss	

Nick	String	4-16	a-z, A-Z, 0-9	Both registered and unregistered users have a nick (this is why Nick isn't FK)
Message	String	128		a-z, A-Z, 0-9, comma, dot, single quote, question mark, minus sign, exlamation point, space, colon

3.4 Error- and exception handling

This is just a quick prototype, so extensive error- and exception handling isn't needed. However errors should be tracked of most important issues and some information be forwarded to console.log, when necessary.

API should handle errors by sending ”NOK” as response message with a message title and an error description object. Successful response (without errors) would contain ”OK” with a message title and a set of data objects.

4 MODULES AND INTERFACES

This chapter describes both server and client modules and their interfaces. Due to a serious lack of time and limitations of course assignments, only two server side modules are described that will be tested according to testing plan. For needs of testing method course, only API is documented, not inner methods and attributes.

In prototype, port 8080 is used for both http and websocket connections.

4.1 Server-side interface: User Authentication

4.1.1 Description

User authentication is a server-side interface, that provides user authentication features:

- User registration
- User login

User Authentication uses WebAPIHandler class to provide Rest API to the client and DatabaseHandler class to read/write from/to the database.

4.1.2 User Registration

Method:	GET
Resource:	/api/user/register
Request parameters:	

Parameter	**Value**	**Description**
username	string	Freely typed by user. Allowed characters: a-z, A-Z, 0-9. Allowed length 4-16 characters.
password	string	Freely typed by user. Allowed characters: a-z, A-Z, 0-9. Allowed length 8-16 characters.
Response parameters:		
Response	**Message**	**Description**
OK	string	Registration succeed
NOK	string	Tells why registration failed: - username might exists already - username or password length isn't allowed - username or password has disallowed characters
Operation:		
1. Request is received through WebAPIHandler from client 2. Input validity is checked → if invalid, response through WebAPIHandler to client 3. Check, does username exist already through DatabaseHandler → if exists, response through WebAPIHandler to client 4. Encrypt password		

5. Insert to database through DatabaseHandler 6. Response through WebAPIHandler to client

4.1.3 User Login

Method:	GET	
Resource:	/api/user/login	
Request parameters:		
Parameter	**Value**	**Description**
username	string	Freely typed by user. Allowed characters: a-z, A-Z, 0-9. Allowed length 4-16 characters.
password	string	Freely typed by user. Allowed characters: a-z, A-Z, 0-9. Allowed length 8-16 characters.
Response parameters:		
Response	**Message**	**Description**
OK	string	Login succeed
NOK	string	Tells why login failed: - username or password was incorrect

		- some other error
Operation:		
1. Request is received through WebAPIHandler from client 2. Input validity is checked → if invalid, response through WebAPIHandler to client 3. Encrypt password 4. Check, does username and password match through DatabaseHandler → if not, response through WebAPIHandler to client 5. Create Player object into system, that contains username (see player class specification) 6. Response OK/NOK through WebAPIHandler to client		

Note: attaching user's websocket connection to player object isn't made by User Login feature, but the chat feature.

4.1.4 User Disconnection

Currently there's no log out feature required (or implemented). Client can't request for this. However user log out happens, when connection to user is lost. User disconnection event happens, when web socket connection is closed by client. When system detects this, it must remove user from websocket connected users list, as well as delete player object, that was created when user logged in.

Public chat message is sent through WebSocketAPIHandler, when some user has disconnected the system. Please see ”events of received messages” in this specification.

4.2 Server-side interface Chat

4.2.1 Description

Chat is a server-side interface, that provides chat features to the users. Chat uses WebSocketAPIHandler class to provide WebSocket API to the client and DatabaseHandler class to read/write from/to the database.

WebSocket event name is ”message”, except in socket.io default events ”connection” and ”disconnect”. Content of messages are in JSON format, following the base structure {request: request-name-here, data: data-object-here}.

4.2.2 Events of received messages

Name:	Public Chat Message		
Description:	This handler receives chat messages from clients and broadcasts them to all connected clients. Save messages to a database.		
Event:	message		
Request:	publicChatMessage		
Data:	{handle: value, chatmessage: value}		
Parameters:	**Key**	**Value**	**Description**
	handle	string	Nick of user who sent the message. Length between 4 and 16, allowed characters a-z, A-Z, 0-9.
	chatmessage	string	The actual message. Length between 2 and 128, allowed characters a-z, A-Z, 0-9, comma, dot, single quote, question mark,

			minus sign, exlamation point, space, colon.
Operation:	1. Check that nick of user match value of socket.name 2. Strip off or replace unallowed characters 3. Check for ”bad words” (not necessary to implement, unless the game is planned to take into production) → if found, push error message to client, through WebSocketAPIHandler 4. Use method of database handler of save the message and date+time → if fails, push error message to client, through WebSocketAPIHandler 5. Use WebSocketAPIHandler to broadcast as public chat message message to all connected clients		

Name:	Change client name		
Description:	Also unauthenticated users can chat, with a dummy nick. When user authenticates, the nick is changed to correspond name of the authenticated user.		
Event:	message		
Request:	changeClientName		
Data:	{username: value}		
Parameters:	**Key**	**Value**	**Description**
	username	string	New nick of user. Length between 4 and 16, allowed characters a-z, A-Z, 0-9.
Operation:	1. Check that username is valid 2. Change value of socket.name to correspond username 3. Broadcast message as public chat message through WebSocketAPIHandler to all connected clients, that user <oldname> is now known as <newname> 4. Broadcast current connected user's clientlist through WebSocketAPIHandler to all connected clients 5. Attach websocket with logged-in player's object, if username matches → if fails, send error message to connected user through WebSocketAPIHandler		

Name:	Connect a new client		
Description:	This initializes an anonymous user with a dummy nick		
Event:	message		
Request:	connectClient		
Data:	{username: value}		
Parameters:	**Key**	**Value**	**Description**
	username	string	Nick of user. Length between 4 and 16, allowed characters a-z, A-Z, 0-9.
Operation:	1. Check that username is valid 2. Change value of socket.name to correspond username 3. Broadcast message a public chat message through WebSocketAPIHandler to all connected clients as public chat message, that user <newname> has joined the chat 4. Broadcast current connected user's clientlist through WebSocketAPIHandler to all connected clients		

Name:	Disconnects a new client		
Description:	When disconnection is notified by socket.io, this method is called. Note: this is a special event that can't be requested by clients, but is caused by client.		
Event:	disconnect		
Request:	(empty)		
Data:	(empty)		
Parameters:	**Key**	**Value**	**Description**

	(empty)		
Operation:	1. Check if user was logged-in, if yes, then remove it's player object 2. Broadcast through WebSocketAPIHandler to all connected clients a public chat message, that user <socket.name> has left the chat 3. Broadcast current updated connected user's clientlist through WebSocketAPIHandler to all connected clients		

4.2.3 Handlers of pushed messages

Name:	Public Chat Message		
Description:	This handler broadcasts the public chat message to all connected clients		
Event:	message		
Request:	publicChatMessage		
Data:	{chatmessage: value}		
Parameters:	**Key**	**Value**	**Description**
	chatmessage	string	Chat message in format <time> <usernick>: <message>.
Operation:	1. Use WebSocketAPIHandler to broadcast message to all connected clients		

Name:	Send error message
Description:	This handler send error message to a specific connected client only
Event:	message
Request:	errorMessage

Data:	{error: value}		
Parameters:	**Key**	**Value**	**Description**
	error	string	Error message generated by some server-side method
Operation:	1. Use WebSocketAPIHandler to send an error message to a connected client only		

Name:	Update client list of chat users		
Description:	Broadcast un updated list of clients to all connected clients		
Event:	message		
Request:	clientList		
Data:	{list: value}		
Parameters:	**Key**	**Value**	**Description**
	list	string	Array of clients
Operation:	1. Use WebSocketAPIHandler to broadcast array of connected clients to all connected clients		