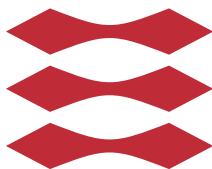


# Major League Wizardry: The Multiplayer Platform

Casper Sloth Paulsen

DTU



Kongens Lyngby 2014

DTU Compute  
Technical University of Denmark  
Matematiktorvet, building 303B,  
2800 Kongens Lyngby, Denmark  
Phone +45 4525 3351  
[compute@compute.dtu.dk](mailto:compute@compute.dtu.dk)  
[www.compute.dtu.dk](http://www.compute.dtu.dk)

# Preface

---

The project explores how to make a scalable multiplayer platform. The case uses to explores this is the mulitplayer game Major League Wizardry.

Lyngby, 28-February-2014

A handwritten signature in black ink, appearing to read 'Casper Paulsen', written in a cursive style.

Casper Sloth Paulsen



# Contents

---

<b>Preface</b>	<b>i</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Major League Wizardry</b>	<b>3</b>
<b>3 Analysis</b>	<b>5</b>
3.0.1 Gameplay . . . . .	5
3.0.2 System overview . . . . .	5
3.0.3 Data Model . . . . .	7
3.0.4 Functional Requirements . . . . .	8
3.0.5 Non-Functional Requirements . . . . .	19
3.0.6 Requirements . . . . .	22
<b>4 Technologies</b>	<b>23</b>
4.0.7 Play Framework . . . . .	23
4.0.8 Scala . . . . .	23
4.0.9 MySQL . . . . .	24
4.0.10 Akka . . . . .	24
<b>5 Design</b>	<b>25</b>
5.0.11 Application design . . . . .	25
5.0.12 Storage design . . . . .	27
5.0.13 Architectural design . . . . .	31
<b>6 Design / Implementation</b>	<b>41</b>
6.0.14 Architecture . . . . .	41
<b>7 Testing</b>	<b>43</b>
7.0.15 Functional testing . . . . .	43
7.0.16 Non-functional testing . . . . .	46
<b>8 What we have learned</b>	<b>47</b>

<b>A Glossary</b>	<b>49</b>
<b>B Abbreviations</b>	<b>51</b>
<b>C Database design</b>	<b>53</b>

## CHAPTER 1

# Introduction

---

Major League Wizardry is a cross platform multiplayer card game for Android, iOS and desktop. A prototype of the game was develop during spring 2013 in which we, the team behind the game, deemed it a good enough success, that we want to make it an international success.

This report then focuses on how to make a suitable multiplayer platform that can sustain an international success. For the platform to support the traffic of an international success, we want to make the multiplayer platform scalable to sustain Major League Wizardry's future success.

The initial goal for the multiplayer platform is to support around 1 million players playing the game on Android, iOS and desktop.

The report will describe elements which has to be taken into consideration when developing a scalable multiplayer platform which can support an international success.





## CHAPTER 2

# Major League Wizardry

---

This is an introduction to Major League Wizardry. MLW is a digital computer and mobile game, based on the rules of the physical card game of the same name. MLW is a multiplayer game and a collectable card game, which means that every player collects and extends their personal pool of Major League Wizardry cards, which they then use to play the game against other players. A player will have a variety of ways to collect cards, one of which will be by buying them from an in-game store. Since the sale of cards will be the main revenue stream, the game itself will be free to play, which means you do not need to buy access to the game. The development contains a large variety of aspects, which is needed for the game to fulfill all the set requirements. Some of these are a game engine to interpret the games rules, some form of communication between players to facilitate the multiplayer, a graphical user interface, anti-cheat to prevent players from cheating during gameplay etc.

A prototype of the game was developed during spring 2013 and was put in testing during the summer of the same year. During the development of the prototype we learned that in order to sustain 1 million players we would need a scalable platform. The initial implementation of the multiplayer platform developed during the prototype fase wouldn't support this and therefore none of the prototype implementation is used.



# Analysis

---

In the analysis we will focus on explaining what the multiplayer platform is, what the functional and non-functional requirements.

## 3.0.1 Gameplay

Before we can go into depth with the analysis of the game, we will briefly look at the gameplay of MLW. MLW is a turn-based game where two players play against each other in what is call a match. A match is a game instance where two players play against each other. A match consists of a number of turns and each turn then consist of a list of moves, where a move is an action a player performs. In each turn a player then performs his moves and when he is done performing his moves, he gives the turn to the opponent player. The match then continues to play out like this until a winner is found. This is the basic gameplay of MLW which is necessary to understand the multiplayer platform.

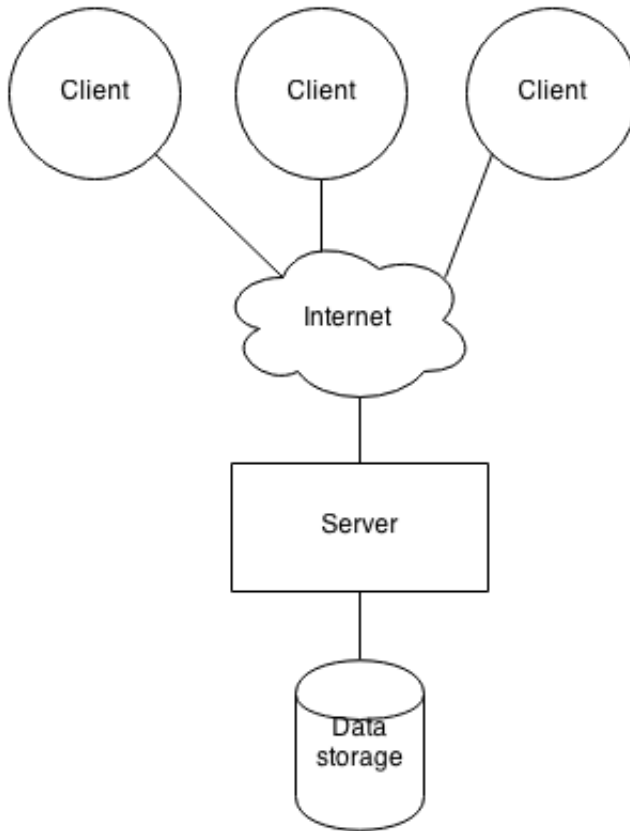
## 3.0.2 System overview

Here we will have an overview of the system at its highest level. MLW is a multiplayer game and in a multiplayer game two or more players play against each other in a match, therefore the player needs some means to play against each other. Our game is a digital game and therefor players uses digital devices as PC, telephones or tablets to MLW. Because the MLW is played across a plethora

of devices, the devices need a way of communicating with each other, in this case over the internet. We will now use the term client to describe a player and the player's devices.

As describe in Sec. 3.0.1, the game is turn based. Because MLW is turn based a player should be able to pause and reenter a match at any given time. This influences how we make the communication between the two clients in a match. When a client needs to communicate with another client over the internet there are two means of communication, peer-to-peer and server-to-client. Peer-to-Peer communication is where two clients communicates directly with each other and client-to-server is where a client communicates with the server which than relays the communication to the other client. The client-to-server communication is the most fitting as it makes it easier for client to pause the match and enter it again later. This way the clients doesn't have to remember the network address of other clients.

Since the MLW is turn-based and we decided that a match is played asynchronously, the multiplayer platform needs to store data about a match so a client can come back and continue the match. To store information we will probably use a database as it seems as the easiest thing to implement, but we will discuss this later. In Sec. 3.0.5.2 we discuss the storage requirements for a match. Fig. 3.1 shows an overview of the system. It shows that a client communicates through the internet with a server and an attached data store.

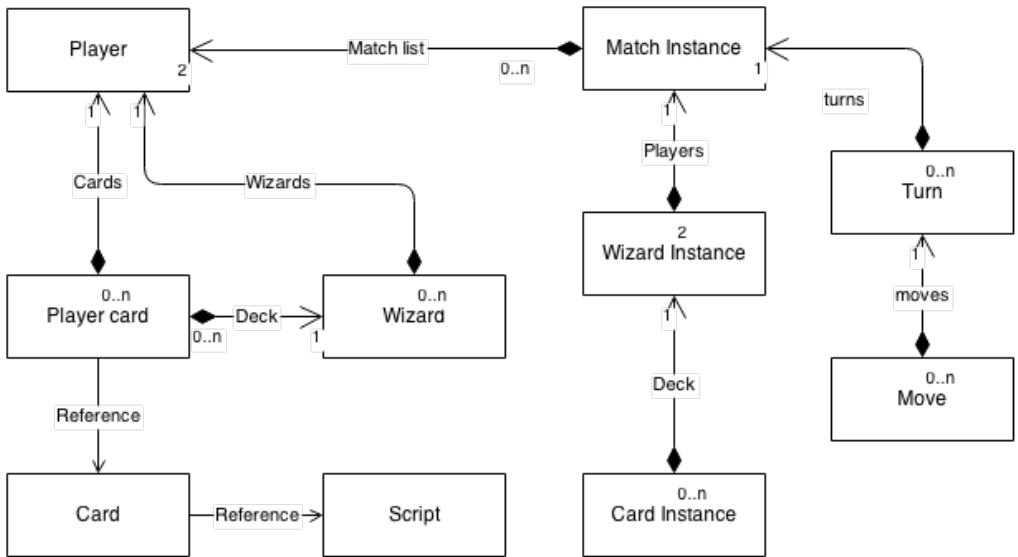


**Figure 3.1:** General setup.

The data store, can be a database or the filesystem on the server, which is used for storing match data.

### 3.0.3 Data Model

Here we will have a look at the data model for the multiplayer platform. The data model is used to explain the different types of data we have. Fig. 3.2 shows the different data types and their relations.



**Figure 3.2:** Data model.

At the top we have a player. A player has a list of cards, a list of wizards and a list of games. Each wizard has a list of the player cards. Each player then references a card which references a script. For a player to play a match, the player selects one of his wizard to use in a match along with that wizard's player cards. Once two players are match against each other a match instance is created. We use the term instance, because it describes that the server takes a copy of current data and uses that as long as the match persists. A match consists of two wizard instances and the wizard card instances. A match also consists of turns and each turn consists of moves.

The reason the server has to take a copy of all data is that the initial data should not change during the match. By initial data we mean the wizard and card instances. This is necessary because we want to be able to change the scripts of the cards, but as they are a central part of the game changing during a match could cause inconsistencies between clients and even lead to a match breaking, so it becomes unplayable.

### 3.0.4 Functional Requirements

Here we will look at the functional requirements of the multiplayer platform. First we will go through the basic use cases of account creation and login. Then we will look at the wizard creation and creating a game. At last we will look the use cases of playing a match.

### 3.0.4.1 Account creation

Account creation is a simple use case where the user input some personal information, like username, email etc. A user should not wait too long time for the account creation to finish when making the request to the server. It is also during the account creation process that the user is assigned the basic cards that the user owns. Later the user has the possibility of buying more cards and therefor the cards gain actual value for the user both in game, but also in real life, and should therefore be treated with care and must not be deleted without user consent.

<b>Use Case 1</b>	<b>Account creation</b>
<i>Scope:</i>	System-wide
<i>Level:</i>	User-goal
<i>Primary Actor:</i>	Player
<i>Preconditions:</i>	<ol style="list-style-type: none"> <li>1. No account with given username</li> <li>2. No account with given email</li> </ol>
<i>Postconditions:</i>	<ol style="list-style-type: none"> <li>1. Account is created</li> <li>2. Basic cards are assigned to the account</li> <li>3. User is inform of successful creation of account</li> </ol>
<i>Main Success Scenario:</i>	
<ol style="list-style-type: none"> <li>1. User sends a request for creating account providing a username, email and password</li> <li>2. System checks if email or username exists</li> <li>3. Account is created</li> </ol>	
<i>Extensions:</i>	

## 2.a Existing email:

1. System shows failure message
2. User returns to step 1

## 2.b Existing username:

1. System shows failure message
2. User returns to step 1

**3.0.4.2 Login**

The login use case is also very simple. A user provides his username and password used during the account creation process and is then authorized to play the game. Since this is a very simple scenario, the user of course want it to be very quick, so the user can get to play some matches.

<b>Use Case 2</b>	<b>Login</b>
<i>Scope:</i>	System-wide
<i>Level:</i>	User-goal
<i>Primary Actor:</i>	Player
<i>Preconditions:</i>	An account for the username must exist
<i>Postconditions:</i>	User is inform of the successful login and enters the system

*Main Success Scenario:*

1. User sends credentials in form of a username and a password
2. System checks if the username exists
3. System checks if the password matches that of the user
4. User is successfully logged in to the system

*Extensions:*



2.a Username doesn't exist:

1. System shows failure message
2. User returns to step 1

3.a Invalid login data:

1. System shows failure message
2. User returns to step 1

---

*Frequency of Occurrence:* 70 logins/minute. 1 login per day for 10% of 1 million users.

---

### 3.0.4.3 Create a wizard

Creating a wizard is a simple use case where the player inputs a name for the wizard and selects a school for his wizard. A player can have multiple wizards at anytime and the reason for a wizard to have name is so the player can distinguish between his wizards. The school is needed as it is core part of Major League Wizardry. After this, the player is capable of creating and playing games. This use case must also be quick, because it is simple and that is the last step before playing a match. In a sense this is critical step, due to the fact that the more steps a player has to go through the more likely it is that the user gives up and leaves the game before even having played a game<sup>1</sup>. From a player's perspective the wizard, represent some value to the user, since the wizard both is the deck and what the player uses for playing. Therefore the wizard has value from a user perspective and should not be deleted without user consent.

Use Case 3	Create a wizard
<i>Scope:</i>	System-wide
<i>Level:</i>	User-goal
<i>Primary Actor:</i>	Player
<i>Stakeholders and Interests:</i>	<ul style="list-style-type: none"> <li>• Player: he wants a wizard for playing a game</li> </ul>
<i>Preconditions:</i>	An account for the user

---

<sup>1</sup>Link to an article explaining this!

- Postconditions:*
1. Successful creation of wizard
  2. User is reported about the successful creation of the wizard
- 

*Main Success Scenario:*

1. User send a request for creating a wizard where he provides a name for the wizard and a school
  2. System checks the school
  3. System checks the wizard limit for the user
  4. System creates the wizard
  5. System adds base deck to the wizard
  6. Wizard is successfully created
- 

*Extensions:*

- 2.a Invalid school:
1. System shows failure message
  2. User returns to step 1
- 2.a Wizard limit reached:
1. System shows failure message
  2. User returns to step 1
- 

### 3.0.4.4 Creating a match

Creating a match means that the multiplayer platform sets two players to compete against each other. As described in the data model each player can have multiple matches at the same time where he competes against different opponents, other players. Creating a match is possible in two ways for a player. One is where the player seeks a random match, where his opponent is randomly selected between other seeking players, and another is where the player sends a challenge to another player.

When a user seeks a random match, he probably wont mind that there is some delay before he can play a match, this should however not affect the player in a way that the player cannot perform other task in the game, like playing another match, so the player should not wait on platform finding the player an opponent. When a player seeks a random match the player would probably like to play

against other players on the player's level, so some match making should take place otherwise the player could be pitted against an player on a higher level which would ruin the gaming experience for the player.

The other way of creating a match is using challenges. Here the player creates a challenge to another player, which then can either decline or accept the challenge. Creating a challenge consist of two parts, one being that the player creates a challenge and another being that a player accepts a challenge. In the first part the challenger, the player initiating the challenge, has to wait on the opponent to accept the challenge and therefore the platform cannot do much other than notifying the opponent that a challenge is awaiting his response, and there for the challenger probably won't mind waiting on the opponent as this is common practice. The other part of the process is accepting or declining the challenge. This step should be fast and the player who gets the first turn should be able to play the match right after it is created. If the opponent declines the challenge, the other user should be notified of this. As a player has not invested much time in making a challenge, giving it value, and that is easy to make a new challenge we can assume that the user does not care if a challenge is lost, as long as it does not happen all the time.

<b>Use Case 4</b>	<b>Creating a random match</b>
<i>Scope:</i>	System-wide
<i>Level:</i>	User-goal
<i>Primary Actor:</i>	Player
<i>Preconditions:</i>	1. Player has a wizard
<i>Postconditions:</i>	1. Game is created 2. The player getting the first turn is notified about the created match
<i>Main Success Scenario:</i>	



---

<i>Postconditions:</i>	<ol style="list-style-type: none"> <li>1. Match is created</li> <li>2. Player getting the first turn is notified that it is his turn</li> </ol>
------------------------	---

---

*Main Success Scenario:*

1. Player sends a request for creating a challenge with a wizard and the name of an opponent user
  2. System checks if wizard is owned by player
  3. System checks if opponent player exists
  4. System creates a challenge
- 

*Extensions:*

- 2.a Player don't won the wizard:
    1. System shows failure message
    2. Player returns to step 1 and corrects the errors
  - 3.a Opponent player doesn't exists:
    1. System shows failure message
    2. Player returns to step 1 and corrects the errors
- 

---

<b>Use Case 6</b>	<b>Accepting a challenge</b>
-------------------	------------------------------

---

<i>Scope:</i>	System-wide
---------------	-------------

---

<i>Level:</i>	User-goal
---------------	-----------

---

<i>Primary Actor:</i>	Player
-----------------------	--------

---

<i>Preconditions:</i>	<ol style="list-style-type: none"> <li>1. Player owns a wizard</li> <li>2. A challenge exists where player is opponent</li> </ol>
-----------------------	---

---

<i>Postconditions:</i>	<ol style="list-style-type: none"> <li>1. Game is created</li> <li>2. Player getting first turn is notified</li> </ol>
------------------------	--

---

*Main Success Scenario:*

1. User accepts the challenge and send his wizard
  2. System checks if player is part of challenge
  3. System checks if player owns wizard
  4. System creates game
- 

*Extensions:*

- 2.a User declines challenge:
    1. Challenger is notified about the decline
    2. System removes challenge
  - 3.a User not part of challenge:
    1. System shows failure message
  - 4.a User don't own wizard:
    1. System shows failure message
    2. Player returns to step 2 and corrects the errors
- 

### 3.0.4.5 Play a match

We can break playing a match into four parts. We are excluding the part where a player makes a move, because the server only distinguish between turns and not the individual moves made in a turn. The four parts are the following.

Getting all the open matches of a player.

Retrieving a match.

Ending a turn in the match.

Finally ending the match.

The first part is about getting the list of matches that a user has. This is a very common operation performed by the user very often and should be as quick as possible, as waiting a long time on retrieval of the game list can ruin the experience for the user as it happens very often. Second is retrieving a match. Retrieving a match means that the client either retrieves the match from the server along with all turns belonging to that match. This part should be quick from the platforms and user perspective. However, since loading the game also

consists of loading the assets, graphics etc., inside on the client, we can get away with a little longer retrieval time from the platform as the user probably will not notice that the client is waiting on the platform. However as this is a step performed very often it should not take too long either. The third part is ending a turn. Ending a turn means that the turn taking by a player is send to the multiplayer platform. This must also be quick as this is performed often. The last part is ending the match. Ending a match happens when a winner of the match is found. As playing, a match is a central part Major League Wizardry this is also one of the most important parts of the platform. If matches are lost, the players will leave the game.

<b>Use Case 7</b>	<b>Retrieving match list</b>
<i>Scope:</i>	System-wide
<i>Level:</i>	User-goal
<i>Primary Actor:</i>	Player
<i>Preconditions:</i>	Player has an account
<i>Postconditions:</i>	Retrieves the match list
<i>Main Success Scenario:</i>	
<ol style="list-style-type: none"> <li>1. Player sends a request for all open matches that he is a part of</li> <li>2. System finds all matches belonging to the player</li> </ol>	
<i>Frequency of Occurrence:</i>	2.100 times/minute
<b>Use Case 8</b>	<b>Retrieving a match</b>
<i>Scope:</i>	System-wide
<i>Level:</i>	User-goal
<i>Primary Actor:</i>	Player
<i>Preconditions:</i>	<ol style="list-style-type: none"> <li>1. Player has an account</li> <li>2. Player has a match</li> </ol>
<i>Postconditions:</i>	Retrieves the match

---

*Main Success Scenario:*

1. Player sends a request for a match
2. System checks if player is part of match
3. System returns the match
4. System finds all turns which are part of the match
5. System returns turns

---

*Extensions:*

- 4.a User is not part of game:
  1. System shows failure message

---

*Frequency of Occurrence:* 2.100 times/minute

---

<b>Use Case 9</b>	<b>Adding a turn</b>
<i>Scope:</i>	System-wide
<i>Level:</i>	User-goal
<i>Primary Actor:</i>	Player
<i>Preconditions:</i>	Player is part of game
<i>Postconditions:</i>	<ol style="list-style-type: none"> <li>1. Turn is added to match</li> <li>2. Opponent player is notified</li> </ol>

---

*Main Success Scenario:*

1. Player makes a turn and sends it to the system
2. System validates turn
3. System adds turn

---

*Extensions:*



---

2.a Not player turn:

1. System shows failure message

---

*Frequency of Occurrence:* 4.200 times/minute

---

### 3.0.5 Non-Functional Requirements

Here we will look at the non-functional requirements for the multiplayer platform. We have not talked about how many users the platform should be able to support. This is however a hard question to answer as we don't know how successful the game is going to be, so currently we compare our game to another game called Shadow Era<sup>2</sup> which achieved 2 million users within a year<sup>3</sup>. Therefore, we have set a goal of supporting 1 million users.

#### 3.0.5.1 Response time

Here we will look at response times and the effect on the client. First we define how response times are perceived. the numbers<sup>4</sup> in fig. 3.3 show how the user

- **0.1 second** is about the limit for having the user feel that the system is reacting instantaneously, meaning that no special feedback is necessary except to display the result.
- **1.0 second** is about the limit for the user's flow of thought to stay uninterrupted, even though the user will notice the delay. Normally, no special feedback is necessary during delays of more than 0.1 but less than 1.0 second, but the user does lose the feeling of operating directly on the data.
- **10 seconds** is about the limit for keeping the user's attention focused on the dialogue. For longer delays, users will want to perform other tasks while waiting for the computer to finish, so they should be given feedback indicating when the computer expects to be done. Feedback during the delay is especially important if the response time is likely to be highly variable, since users will then not know what to expect.

**Figure 3.3:** Response times for web and application performance

---

<sup>2</sup><http://www.shadowera.com/content.php?s=377f78bd36cfd31135af53f8e8db9f77>

<sup>3</sup><https://play.google.com/store/apps/details?id=com.wulven.shadowera&hl=da>

<sup>4</sup><http://www.nngroup.com/articles/response-times-3-important-limits/>

perceives response times within a web and desktop application. First, we have the login and account creation scenarios where we saw that from a user perspective, they should be quick and that information is stored securely and confidentially. When we look at these numbers, and take into account the amount of data transfer, during login and account creation, the process of creating an account and logging in should be less than 1 second. As the user will expect it to take some time, but the user should not really notice the delay. We will use these numbers to define the latency of other tasks.

Latency for creating a challenge or random game, when the user presses the buttons, should be less than one second

Games are considered very valuable to the user and should therefore be persistent and we don't want to accidentally delete or lose a game due to the application design. Since every user can have an arbitrary number of matches, we should design the platform in a way that it can handle the number of games and their data size. Fig. 3.3 shows the response times for each request based on the use cases.

Account creation	1 second
Create a wizard	500 milliseconds
Login	500 milliseconds
Creating a random game	500 milliseconds
Creating a challenge	500 milliseconds
Accepting a challenge	1 second
Retrieving a game list	200 milliseconds
Retrieving a game	1 second
Adding a turn	300 milliseconds

**Figure 3.4:** Response times per type of request.

### 3.0.5.2 Storage requirements

To understand the storage requirements for MLW we will look at data extracted from the alpha version of the game. Fig. 3.5 shows the numbers from the alpha..

Users	836
Matches	169
Avg. Matches / User	0.2
Avg. Turns / Matche	11
Avg. Moves / Turn	3
User Cards	135.863
Avg. Cards / User	162

**Figure 3.5:** Numbers derived from prototype.

With these numbers from Fig. 3.5 we can translate them to 1 million users.

These are of course only rough estimates, but they give picture of what is to be

Users	1.000.000
Matches	202.153
Avg. Matches / User	0.2
Avg. Turns / Match	11
Avg. Moves / Turn	3
User Cards	162.515.550
Avg. Cards / User	162

**Figure 3.6:** Numbers derived from prototype.

expected from the multiplayer platform.

Let us look at the data sizes for the system based on Fig. 3.6. This is to get a better understanding of what is required of the system in terms of data storage. Fig. 3.7 shows the data size requirements, which are based on a prototype sample.

Accounts	550 MB
Matches	13150 MB
Turns	1700 MB
User Cards	120 MB
Finished Matches (100 Million matches)	7256 GB

**Figure 3.7:** Numbers derived from prototype.

The biggest concern from Fig. 3.7 is the 100 million matches which are to be stored for analysis and retrieval by the players. The above numbers just show the raw size of data and we haven't taking into account if they could be stored better.

### 3.0.5.3 Important player data

As wizards are considered valuable to the player we have to store these in a way so that they are not deleted by accident and are persistent, we also have to enforce ownership of the wizards, which means that only player who owns the wizard can manipulate with the wizard, as they are considered personal. As a player can have an arbitrary number of wizards, we should design the application to handle the numbers of wizards that users store on the platform.

When we discussed creating a match, we talked about a random match and a match made by challenging. The process of creating a random match was not given much value from a player's perspective as it is an easy operation to perform and the player does not invest any time in it. Therefor we can treat it as this, a low value operation that does not need to be persistent and we do not mind if it is lost, as long as it does not happen too often. By often mean we mean more

one in every hundred. The same thing can be said about challenges. They do not have a high value either and if we loses some, it doesn't really matter to the user. The last thing we have to consider is players cards. Player card are item which the player uses to play the game and but it is also possible to buy new cards. Player cards must not be deleted or lost in any other way then with the player's consent and we should enforce a high level of ownership of the cards. This should be taking into account when designing the application. The player can have an arbitrary number of cards and during the account creation, the player is given 60 cards. All player cards are also considered unique, so even do two player cards have the same name and stats, they are still considered as individuals.

#### 3.0.5.4 Confidentiality

During the account creation process, the player gives us some personal information like the players name, email address and password. The player also gives us an username, which is used to identify the player. The players name and email should not be visible to the public, so we should make sure that this is not leaked to unauthorized users, other players. It should however be visible to administrators and it will be used to communicate with the user. The password is considered highly personal and should not be visible to any user of the system, not even administrators. The username is used to identify the user, is used by other user throughout the game, and is therefore visible to the public. The username named is not personal in any other way than usernames are unique.

### 3.0.6 Requirements

The following is the requirements for the multiplayer platform.

1. User data has to be persistent.
2. All matches should be persistence until they are finished, so even if a client loses connection the game is still retrievable.
3. Handle up to 1 million users.
4. Should handle up to 100 million games being stored, due to persistence and games being stored for later analytic.
5. Have an up time of 99.9%, as the impact of an outage is fatal for Major League Wizardry's success.
6. Latency should be acceptable for the clients, based on Fig. 3.4.

## CHAPTER 4

# Technologies

---

The following technologies are used for the implementation of the multiplayer platform.

### 4.0.7 Play Framework

Play is an open source web application framework, written in Scala and Java, which follows the model-view-controller architectural pattern. It aims to optimize developer productivity by using convention over configuration, hot code reloading and display of errors in the browser<sup>1</sup>.

### 4.0.8 Scala

Scala is an object-functional programming and scripting language for general software applications, statically typed, designed to concisely express solutions in an elegant, type-safe and lightweight manner<sup>2</sup>.

---

<sup>1</sup>[http://en.wikipedia.org/wiki/Play\\_Framework](http://en.wikipedia.org/wiki/Play_Framework)

<sup>2</sup> [http://en.wikipedia.org/wiki/Scala\\_\(programming\\_language\)](http://en.wikipedia.org/wiki/Scala_(programming_language))

### 4.0.9 MySQL

MySQL is a database management system. A database is a structured collection of data. It may be anything from a simple shopping list to a picture gallery or the vast amounts of information in a corporate network. To add, access, and process data stored in a computer database, you need a database management system such as MySQL Server. Since computers are very good at handling large amounts of data, database management systems play a central role in computing, as standalone utilities, or as parts of other applications<sup>3</sup>.

### 4.0.10 Akka

Akka is a toolkit and runtime for building highly concurrent, distributed, and fault tolerant event-driven applications on the JVM<sup>4</sup>.

---

<sup>3</sup><http://dev.mysql.com/doc/refman/4.1/en/what-is-mysql.html>

<sup>4</sup><http://akka.io/>

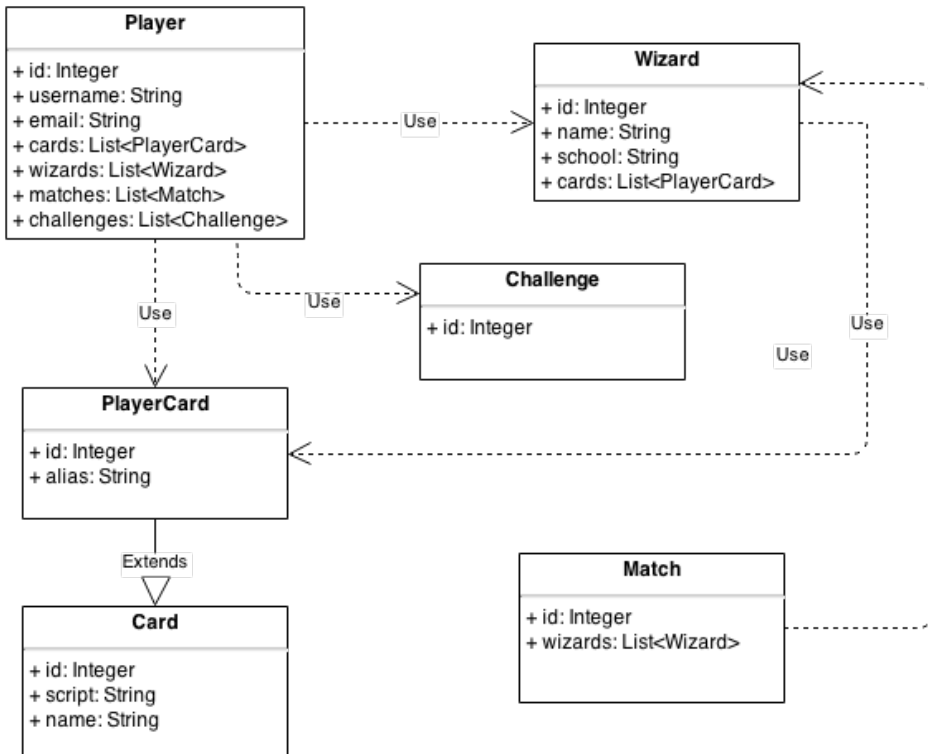
# Design

---

Here we will discuss designs for the multiplayer platform. First, we will look at the application design and then on the architectural design, which is the general setup for our platform.

## 5.0.11 Application design

We have decided to use Model View Controller, MVC, as the design pattern for our web application. We have decided to use MVC because we feel that it is a very good design pattern for structuring an application. Since we do not want to write our own web application framework, we have selected a framework called Play. Play is a MVC web framework, a collection of tools for developing a web application, and it helps with both the structuring of the application and providing abstraction layers for socket connections, routing, database connectivity etc. This way we can focus more on developing the actual application rather than building all of these abstractions. Since we have decided on using the Play framework, we are limited to use the Java Virtual Machine, JVM, since this is what Play uses as its underlying platform. This is not a problem as the JVM is a well-tested and used platform. Play is develop with the programming language Scala, which runs on the JVM, and therefore we will program the application using Scala.



**Figure 5.1:** Class diagram.

Fig. 5.1 shows the class diagram of the application based on the data model. Each player have a list of player cards, wizards, challenges and matches. Each player card then extends the base class Card.

### 5.0.11.1 Threading

As we state in Sec. 5.0.11 we are going to use the Play Framework. The interesting part of Play is that it is fully asynchronous, which means that every incoming request is handled by a single thread and therefore it is important that everything related to heavy computations or IO, should be offloaded to other threads when then returns their results to the request thread when they are done.

Why this interesting is because most of our application is IO bound, which means we do many database operations. So almost every incoming request is directed to another thread. To make our implementation asynchronous we use Akka for handling threading. Akka is a Scala threading library and is used by many third parties<sup>1</sup>.

<sup>1</sup><http://akka.io/>



Normally when using threading one would have to take great care about sharing state across the application. When using Akka and Scala we can eliminate this to some extent. The reason for this is that Akka uses the message-threading model where communication between threads are done through messaging. Combine that with immutable object and we have a safe threading model.

## 5.0.12 Storage design

In this section, we will look at how we store match instance data optimally in the multiplayer platform, We don't look at how we store player data like accounts, player cards and wizards, because we have already decided on using a database for this as it seems the most fitting. We will focus on different storage methods and how we could use these for multiplayer platform and what their advantages and disadvantages are. The storage methods have to live up to the requirements defined in Sec. 3.0.6 and taken into account the data sizes from Sec. 3.0.5.2. Once we have discussed the storage options we will conclude on what storage option is the best suited for the multiplayer platform.

### 5.0.12.1 File system

First we will take a look at the file system. A file system exists on every server and is embedded into the operating system. The file system is generally fast when storing data, that don't need relations and where concurrency can be disregarded. It also provides a direct way of storing data that needs to be persistent. Lets have a look at a possible storage design for storing match instances.

```
user -> user_id -> matches -> match_id -> match_file
                                     turn_file
```

An alternative would be this

```
user -> user_id -> matches_list_file
matches -> match_id -> match_file
```

The general idea is that each game has its own file, this way we get a nice separation of the data. Another way would be to store all games in one file, but this would become really hard to manage from a concurrency point of view and if the match file gets corrupt all matches are lost unless we have a backup. Lets look at the advantages and disadvantages of using a file system as the direct way of storing data. First the advantages

- Fast and direct way of storing data
- No additional software is needed for storing data
- Easy to store big amounts of data

Disadvantages are the following

- Structuring of data is done by the application.
- No abstraction layer for data storage, like in databases
- No indexing
- Consistency issue
- Concurrent access needs to be programmed in the application

The two major advantages of using the file system is storing large amounts of data is easy and it is fast as long as concurrent access doesn't need to be taking into consideration. The main issue with using the file system directly is that retrieving and storing data is up to the individual applications, because there is no abstraction layer to do this with. This is no problem when the data structure is never changed and we only have one application accessing the data. The problem comes when we have multiple application concurrently accessing the data either by reading or writing. Once this happen we will have to program some locks into the system which will be hard as we have multiple application accessing the data and they don't communicate with each other. We could develop a system where the applications would communicate with each other or we could implement a scheme that assign applications to certain data, but this would be complicated and wouldn't scale very good. The problem with concurrent access is that it could lead to inconsistencies in the data. Another disadvantages is if we change the data structure we will have to change the applications also and find we make new application, which might not be in Java, we would have to write a new layer for retrieving and storing the data. Whether or not this is a problem can be debated, but it will not make development any easier.

### 5.0.12.2 Database

Another way of storing data is using a database. When using a database operations are done through a connection.

We will now looking at the advantages and disadvantages of using a database. We will begin with the advantages.

- Better abstraction than using the file system

- 
- Application independent
  - Structure of data is done in the database
  - Concurrency is handled by the database and not the applications
  - Indexing of data which makes it faster when dealing with relational data

The disadvantages.

- Vertical scaling is hard
- Performance could become a problem when data set big enough

Therefore, the main benefit from using a database is that it provides an application independent way retrieving and storing data. So changes are reflected directly in the database rather than in the applications. It provides a way of adding relationship between data by the use of indexing and when indexing is done probably it improves performance. Querying language for getting data where one can get any data in a way preferred by the application. Concurrency is handled by the database rather than the application.

Using a database also has its drawbacks. The biggest being vertical scaling. Vertical scaling is the preferred way of scaling relational databases and horizontal scaling is hard to achieve. To determine if this is possible and beneficial depends entirely on the hardware and the type database being used.

We have been talking about databases in general here, but we will now list some database types we could use, because they both have advantages and disadvantages when we talk storage capabilities and performance.

- Relational databases - MySQL, PostgreSQL, SQLServer etc.
- NoSQL database - RIAK, MongoDB, CouchDB etc.
- Graph database - Neo4j, InfiniteGraph etc.

Typically one would use a relational database as the database backend an alternative is a NoSQL database. NoSQL is a new type of databases, but NoSQL is more a general term rather than categorizing the databases. The general characteristics for NoSQL is that they are distributed, highly scalable and thus not use a predefined scheme like in relational databases. NoSQL databases are mostly new and have to prove their worth like the relational databases. They are mostly used by big data firms like Google, Facebook, Amazon etc. Therefore, if one would use these, either one would have very big databases or relational databases does not meet the requirements or simple one would try something new. NoSQL in general are not ACID transactional, but it depends on the database implementation.

When looking at the requirements for the multiplayer platform we do not see any issues other than if it is reasonable to store 7 TB of data in a single database, but we will discuss this later. Concurrency is handled by the database and the database can use transactionality to make sure data stay consistent. Appx. C shows the structure of a Relational database scheme we could use for the final implementation. Before we go into details let's clarify some issues with terms in the appendix. Accounts refers to players and games refers to matches and cards account refers to player cards. What we have done here is to translate the data model and class diagram into a database scheme. Each account has a list of player cards which is represented by a table called cards account and cards account holds a reference to a table called cards which represents cards in the data model. Scripts for the cards are held in a separate table called cards script and this table is referenced from the cards table. In the data model we had a match instance and in the database scheme this is represented by the table games. The games table has a field called original state and it is this field that represents the instance of the match. The games table is referenced by a table called games turn which represents the turn instance. The last table we will discuss is the table challenges which is referenced by account through an intermediate table because of a many-to-many relationship.

### 5.0.12.3 In-memory Database

An in-memory database uses the memory of server to store data in, so data is stored in RAM. The last storage we want to discuss is that of an in-memory database. We are discussing this design because it provides some performance gains, which could be useful, but it will not be able to lift the job of being the only data store to use. Let us have a look at the pros and cons.

- Fast, very fast.
- Provides the benefit of a database

And the cons

- Maximum data size is limited to RAM size on the machine running the database
- Data is lost when server is shutdown

The major problem here is that data is lost when the server shutdown. This means we cannot use this for storing data that needs to be persistent, which is discussed in Sec. 3.0.5.3, but if we loosen the requirements a bit, we could achieve high performance gains from using an in-memory database. Like with normal databases there are different types of in-memory database. We are not

listing them here but we will discuss a possible in-memory database, which could be used without compromising persistent.

The database we are going to discuss is a NoSQL database called Redis. Redis is an in-memory database but where it differs from other in-memory database is that it actually use harddrive to persist data. Redis uses the memory to store data, and then stores the data on the hard drives by using a time interval. This way it persists data through server and database shutdowns, but there is a time span where data could be lost, so it cannot be used for critical data. Redis do have limits like other in-memory database being the RAM size limit. Another problem with Redis is that it is a key-store database so now advanced querying is possible like in Relational databases. Key-store means that the database generates or receives a key and then uses this key to identify where data is stored, this makes data retrieval really fast but makes querying hard.

#### 5.0.12.4 Which storage design to use

We have now discussed possible storage designs and now we will decide on which design to use. We will use only one of the designs in the implementation for this report, but in the final version of the multiplayer platform, we will use all three. The database scheme we will use is shown in Fig. C. If we had used all three in this report, we would have done the following. Use the Relational database for storing normal data like accounts, cards, wizards, game list etc. Then use the Redis for storing open matches and finally use the file system for storing finished matches. This way the three designs complements each other and we achieve a good foundation for a fast and reliable multiplayer platform. However maintenance becomes harder as we suddenly have three different systems to maintain. A way to remedy this is that we remove the file system design and only use the two database designs. The difference would be that the normal database would also hold the finished matches instead of the filesystem. To simplify even more we could use only the relational database which is what we are going to do for the initial design. This way we only have one system to maintain.

For the implementation, we will use the Relational database MySQL, since it is a free and open source database, which is very popular to use for web applications.

#### 5.0.13 Architectural design

The architectural design is based on the Fig. 3.1 where multiple clients are connecting through the internet to a server or more and a data store. We will look at four designs in this report, but there are others, which are based on the general setup and the requirements from Sec. 3.0.6. Once we have discusses the designs we will decide on which design to use for the implementation. The goal is with the designs is to reach scalability and that they live up to our requirements. When we are talking about scalability, there are two ways of achieving it. Either through vertical scaling or horizontal scaling. The operating system we are going

to use for the servers is Debian, which is a Linux distribution. Debian is free software, so we don't pay any licensing.

**Vertical scaling** - To scale horizontally (or scale out) means to add more nodes to a system, such as adding a new computer to a distributed software application. An example might be scaling out from one Web server system to three. As computer prices have dropped and performance continues to increase, low cost "commodity" systems have been used for high performance computing applications such as seismic analysis and biotechnology workloads that could in the past only be handled by supercomputers. Hundreds of small computers may be configured in a cluster to obtain aggregate computing power that often exceeds that of computers based on a single traditional processor. This model was further fueled by the availability of high performance interconnects such as Gigabit Ethernet, InfiniBand and Myrinet. Its growth has also led to demand for software that allows efficient management and maintenance of multiple nodes, as well as hardware such as shared data storage with much higher I/O performance. Size scalability is the maximum number of processors that a system can accommodate.

**Horizontal scaling** - To scale vertically (or scale up) means to add resources to a single node in a system, typically involving the addition of CPUs or memory to a single computer. Such vertical scaling of existing systems also enables them to use virtualization technology more effectively, as it provides more resources for the hosted set of operating system and application modules to share. Taking advantage of such resources can also be called "scaling up", such as expanding the number of Apache daemon processes currently running. Application scalability refers to the improved performance of running applications on a scaled-up version of the system.

The above two statements are taken from Wikipedia<sup>2</sup>, and explains the two ways of scaling in a system. Depending on the system one might build one is better than the other. Vertical scaling is the easiest to work with but have an upper bound on how much computation power one can get into one system. Horizontally scaling is a bit harder to achieve since we have more components in play, which need to communicate with each other, but it does not have the same type of upper bound on computation power, since one can just add another computer to the system. However, the more components in the system the harder it gets to maintain because of the complexity. Whether or not you can scale horizontally also depends on the application it self. If you have a stateful application it is harder to scale horizontally then if you have stateless application.

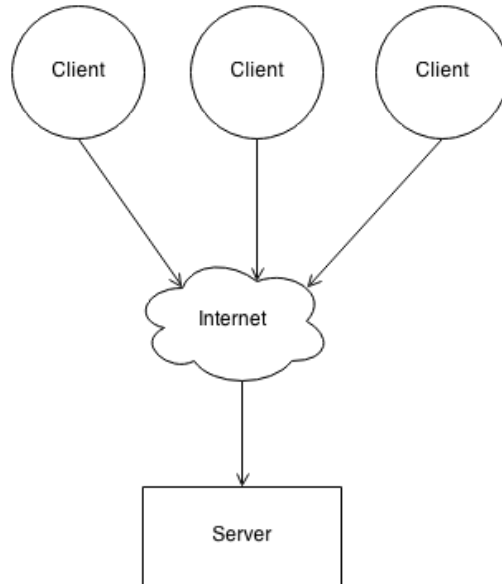
---

<sup>2</sup><http://en.wikipedia.org/wiki/Scalability>

### 5.0.13.1 One server

In this design, we will focus on a design with only one server. Most small size applications only uses a single server and will only ever need a single server. We look at this approach because it is the most basic approach and the easiest to setup, which stills lives up most of our non-functional requirements in Sec. 3.0.6, but the interesting one is if it also holds for performance, we will discuss this later.

Let us first look at the setup, which we will illustrate with Fig. 5.2.



**Figure 5.2:** Data model.

Fig. 5.2 show a setup where we have multiple clients connecting to a single server. The server then both needs to handle the incoming requests from the clients and act as the data store for the web application.

Now let us discuss the scalability of this system, referring to the single server. In Sec. 5.0.13 we discussed the two types of scalability and there pros and cons. In this design, we do not have the opportunity of horizontal scaling. This is because we are designing the multiplayer platform to use a single server, so we only have the vertical scaling available. It is not possible in this design to add more servers to the multiplayer platform as all clients connect to that single server and we do not have a way of distributing the incoming requests from clients to other servers. Therefore, as the traffic for the multiplayer platform increases beyond that single server's capacity, the only thing in this design we can do is to buy a bigger server or equip the server with more RAM or a better CPU. In the end, we will hit the upper bound of computation power we can get into that one server, and we have

not even taking into account the price of server like that.

Now that we have talked about the scalability of the system, let talk about the advantages and disadvantages of this design in terms of the scalability and our requirements. First, we will list the pros in this design.

- Simple system, only one server, which is easy to build
- Development for the web application is easy.
- Initial cost for the server is low

Now the cons of the design.

- Only vertical scaling
- Expensive to run when traffic increases, unless we rent a virtual server
- Single point of failure, unless we have a good SLA

Now let us explain why these are thought of as pros and cons. Since it is a simple system, it is easier build the multiplayer platform on it since we do not have to take distribution and synchronization between servers into account. This make it easier to do things like caching, error handling, storing data etc. The initial cost of the multiplayer platform will also be low as we only have on server to buy, and a standard low-end server could easily handle around 10.000 clients, but it depends on how the traffic is distributed. These are pros because it makes it easier for us to get a prototype build fast and get it tested by our clients and we can rapidly iterate on it. However, there are also cons in the system. The system only allows vertical scaling since the design focuses on having only a single server. So even do it is easier to setup a system like this and build on it, it becomes harder and harder to scale it as we get more traffic. Therefore, we lack in scaling when we hit a certain threshold. What this threshold is harder to estimate for now. Another problem we have with this design is that it is a single point of failure. Since we only have one server running, then when it goes down, our clients do not have another server to contact. We could fix this by having a spare server running, but we will not benefit from it from scaling perspective. Then when the main server goes down we will fall back to the backup server we have running. The problem then becomes to synchronize the data between the main server and the backup server. This does not eliminate the downtime when the main server goes down, but might reduce the downtime, and then remedy the single point of failure issue.

With the pros and cons in mind let's look at the requirements from Sec. 3.0.6. It should be possible to live up to the first two requirements, persisted user data and matches. Whether or not we can handle 1 million users and 5 matches per user per day we would have to test, before settling on this design, Currently the 5 matches per day is a calculation and we don't have any numbers suggesting

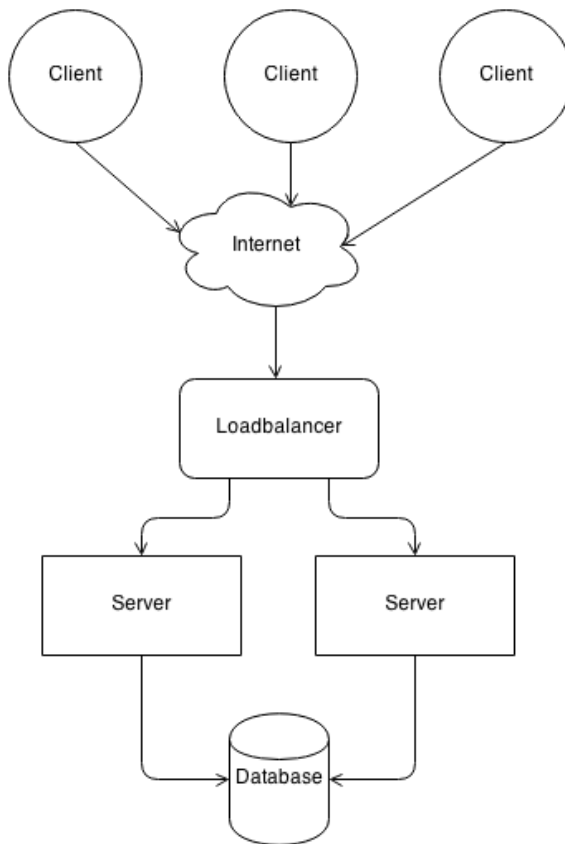


how many there will actually be, so it is a rough estimate. An uptime of 99.9% should be possible with server equipment. For latency, if we assume all 1 million users are playing, some quick math gives us, that we should handle on average 350 request/sec, referring to Sec. 3.4. We argue that the load will be evenly distributed because we are going international, so all time zones should come into play. Which should be possible with a good enough server and application.

### 5.0.13.2 Multiple servers and a database

This design focuses on having multiple servers handling requests from clients and having a database to store data. This design is a very common design, because it separates the web application servers, which are servers running the application, from the database, and it makes it possible to scale the system horizontally.

Let us first look at the setup in general, which Fig. 5.3 shows.



**Figure 5.3:** Data model.

As we can see from Fig. 5.3 we have a load balancer in front of the application servers and then behind the application server we have the database. The idea is that when a client send a request to the servers it has to go through the load balancer that then distributes the request to one of the connected servers.

Let us now discuss the scalability of this system. In this system, we have to look at two parts. One part is the application servers and the other is the database server. We do this because scaling is different for the two parts. First by introducing a load balancer before the applications servers it becomes easier to add a new server or take an existing server down because the load balancer simple redirects the traffic to another server. This of course means that have to be at least two or more application servers otherwise it will not work. One could of course start with just one application server behind the load balancer and then add more on the go, but two are needed when taking one server down. Here we both benefit from vertical scaling but also horizontal scaling because we can simple add more application servers to the system as we get more traffic. of course, we have introduced some problems due to the horizontal scaling, which we will discuss later. We also have a database server in this design. When scaling a database in this design we only have vertical scaling available, because we design with only one database. To decide if this is a limiting factor, we need testing, but most database depending on the server they are run on is more than capable of server millions of request per second. Therefore, it might not be a problem and with caching on the application servers, we can minimize the impact.

Now we will look at the advantages and disadvantages of the system. First the advantages.

- Horizontal scaling
- Load balancing
- Lower cost when traffic increases, since cost per server scale linear unlike a single server cost scales vertically

The disadvantages.

- Harder to develop for
- Harder to maintain due to higher complexity
- Vertical scaling of the database only
- Database is single point of failure
- Load balancer is single point of failure

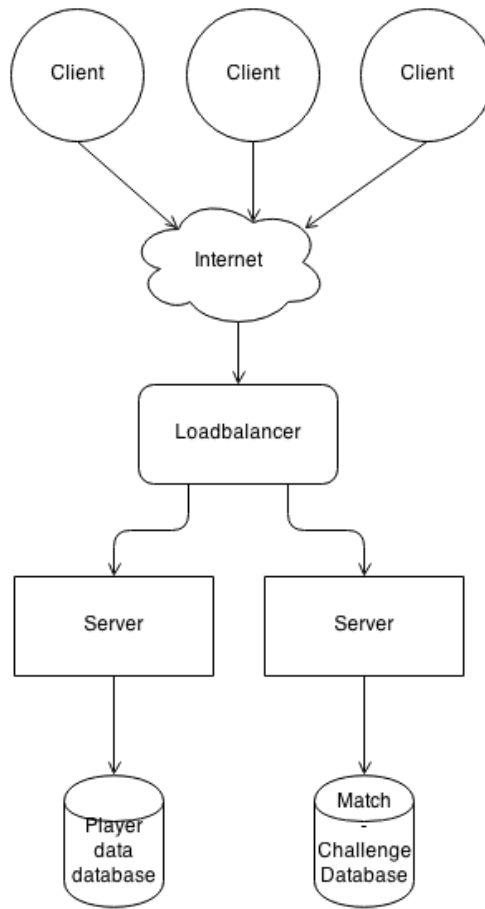
One of the biggest achievement in this design is that we now have the possibility of horizontal scaling for the application servers. So when we get more traffic for the multiplayer platform, we can simple add more application servers to the

system to handle the increased traffic, and if we want more powerful application servers, we can just scale the existing application server vertically. By adding more application servers to the system, we also eliminate the single point of failure, which we had in our single server design. The reason this works is that we have load balancer in front of the application servers, so when an application server goes down, then the load balancer redirects the requests to other working application servers. However, the load balancer is now a single point of failure instead of the application servers. However, we will neglect this for now, as we have ways of fixing that with some redundancy in form of Domain Name Servers, DNS, fail over. The most important disadvantages we have with this system is single point of failure in form of the database. As we only have one database in this design, it becomes a single point of failure. We can however work around this by have an extra database server, which is then working as a replication server. So this way we have a hot spare database running and by using some fail over mechanisms in our application servers, we can handle if the main database goes down with virtually no downtime. Nevertheless, this does not eliminate that we only have vertical scaling available for the database. A single database, depending on the hardware, which runs it, is capable of handling a lot of traffic, so it might not be a problem. We will need testing to determine if it is a problem. Another problem is that the load balancer is also a single point of failure. Since all traffic goes through the load balancer then if it goes down all communication is down. With a good enough SLA we shouldn't worry about it breaking from a hardware perspective, but more if it can handle the increased bandwidth needed to support the platform. We haven't tested the bandwidth need as of this writing. The other problem with this design is that it have a higher complexity than the single server design and is harder to develop for, but the benefits of horizontal scaling outweighs these cons of complexity and harder development. So generally, this is a good design, which is clearly better than that of the single server, seen from a scaling perspective.

When we look at the requirements, we should be able to fulfill all of them with this design.

### 5.0.13.3 Multiple servers and databases

This design focuses on having multiple application servers and multiple databases. It looks a lot like the design in Sec. 5.0.13.2 with the addition of more databases. The general idea is by using more database we can distribute the load on the databases and achieve redundancy. Fig. 5.4 shows the design.



**Figure 5.4:** Data model.

From a scaling perspective we have the benefits from Sec. 5.0.13.2, but also the benefit of additional database servers which helps with getting horizontal scaling of the database. There is not much to say about this design, which have not already been discuss in Sec. 5.0.13.2 the only new thing is the extra database. The interesting thing is than making the two or more databases work together as a single database.

- Sharding
- Cluster
- Master-Slave

So now, we will discuss the possibilities. Sharding works by sharding the database into shard, where shards can be a collection of database entries, which are stored

together in an interval of keys. The idea is that an entry in the database have a shard key, which tells the application servers where to fetch the data. This way it is possible to add more database and then assign a shard key to the database, so the servers know which database to connect to.

Clustering is another technic used for horizontal scaling of database. Here a cluster of database works together and form a unified system, which looks like a single database to the outside world. This way an application server would connect to an interface that represent the cluster and the cluster will then determine where to fetch the data and serve it back the application server.

Master-Slave is a concept where one database acts as the master server for the system and then the slaves replicates the master server. The master server's job is to handle all writes and then push these to the slaves, which then handles all reads. In this concept, there can only be one master server, but there is no limit on slaves.

So let us look at the advantages and disadvantage of this design.

- Horizontal scaling of application servers
- Load balancing
- Horizontal scaling of database
- Lower cost when traffic increases, since cost per server scale linear unlike a single server cost scales vertically

The disadvantages.

- Harder to develop for
- Harder to maintain due to higher complexity
- Load balancer is single point of failure

The advantages are a fully horizontal scalable system because both the application servers and the database can be scaled horizontally. We have a load balancer to distribute the load on the application servers. We still have the single point of failure issue with the load balancer, but the solution is a DNS solution to fix that by having more load balancers ready.

When we look at the requirements defined in Sec. 3.0.6 we are confident that this design lives up to all of them. We are confident because the design is horizontally scalable. The one major problem with this design is the complexity and development needed to setup a system like this.

#### **5.0.13.4 Which architecture we chose**

Now let us decide on which design to use for the implementation.

We do not like the design of the single server discussed in Sec. 5.0.13.1 because

of its limit in scaling and that, it is a single point of failure. The benefits of easier development and maintenance is not enough to out weight its limits in terms of scaling, so the design is rejected.

This means it comes down to the designs discussed in Sec. 5.0.13.2 and Sec. 5.0.13.3. These have horizontal scaling and do live up to the requirements defined in Sec. 3.0.6. Therefore, since it lives up to the requirements and is horizontal it comes down to whether or not we want to develop with increase on complexity of more database and if we think that one database with a replication database is enough. We think that the increased complexity of having more databases is not worth the extra development time and we think that one database should be enough for now. We could buy an off the shelf solution, but as we are a startup we don't want to pay the price. Therefore we have decided on the design with multiple application servers and a single database with a replication server for backup of the database.

## CHAPTER 6

# Design / Implementation

---

This section covers the implementation of the multiplayer platform. We will look at the implementation of the architecture and next we will look at the implementation of the actual application used to serve the clients.

### 6.0.14 Architecture

This section discuss the implementation of the architecture. This means we will focus on the setup we chose to use in Sec. 5.0.13.4.

#### 6.0.14.1 Loadbalancer

In Sec. 5.0.13.4 we decided to go with a design using a load balancer. As we discussed in the design in load balancer is a critical point. It is the a single point of failure and therefore we have to take care in the implementation. What we have done instead of setting up a single server, our selves, is that we have brought an off the shelf solution at hosting provider. The reason we did that is because we don't have to administer the load balancer our selves and we get a service level agreement, SLA, guaranteed a solution which scales up to 10.000 concurrent connections. The only thing we need to administer is to add the application servers to the load balancers routing table. The rest is taken care of by the hosting provider.

### 6.0.14.2 Application Servers

This section looks at the implementation of the application servers. We will discuss the operating system of the application servers and what type of servers we have used for the implementation.

We have already decided on using the operating system Debian. We decided on using Debian because it is free software, no licensing fees, and is commonly used for web servers. By using a commonly used OS we have a better chance of getting help and that the software we are going to use will be able to run.

We have decided to have all our servers hosted at a hosting provider instead of hosting them our shelf or use co-location. This gives us a certain flexibility because the initial cost of servers are lower, because you lease the servers, and we can easily change servers without having to worry about old servers. For the servers we are going to use virtual private servers instead of dedicated servers. By using a VPS scaling becomes easier because you are not buying hardware but resources. When using a dedicated server we would by hardware and when we would need to scale, we would either have to upgrade the hardware in the server or buy a new server. With a VPS we can simple allocate more resources to it and be done. At our hosting provider this simply means we migrate the VPS to a bigger instance. Beyond the fact that VPS are allocated as resources instead of hardware, there isn't a big difference between a dedicated server and a VPS. Administration is the same and the rest of the differences depends on the hosting provides.

In our implementation each application server is installed with Debian and then a JVM in this case OpenJDK, which is an open source implementation of the JVM. After the initial setup of the servers we upload our application to them and hook them up with the load balancer. To hook an application server up with a load balancer is simply adding the IP address of the server with the load balancer.

### 6.0.14.3 Database Server

Here we will discuss the implementation of the database server. In Sec. 5.0.12.4 we decied on using MySQL as the database backend for the multiplayer platform. The database is hosted on a separate server then the application servers. We have chosen to do this because we than have nice separation of the servers and it makes scaling of the database server easier as all resources is allocated to that server and it is easier to monitor the database. The server the database is hosted on is a VPS, because scaling of the server is easier.

The have implemented the database scheme from Appx. C. Each table in the scheme have a primary key and this key is then used as foreign key when tables references each other. This improves performance, because we can use the indexes for looking up data.



# Testing

---

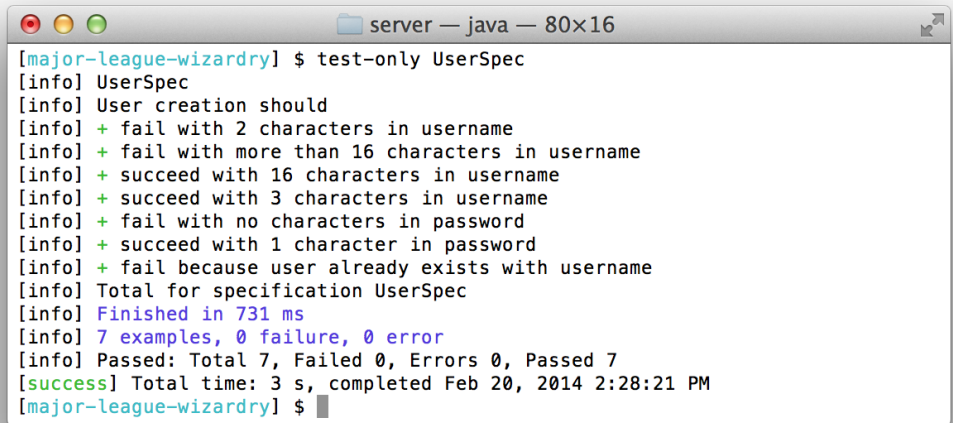
Here we will discuss the testing of the multiplayer platform. We will both test the functional requirements and non-functional requirements.

## 7.0.15 Functional testing

In the functional tests we are going to test the following use cases account creation, login, creating a wizard, creating a challenge and creating a match. Use cases are taken from Sec. 3.0.4. For testing the functional requirements, we are using unit testing with the Spec2 testing framework. Spec2 is a Scala testing framework much like JUnit.

### 7.0.15.1 Account creation

Here we test the creation of an account. The test is based on the use case Account Creation in Sec. 3.0.4.1.

A terminal window titled "server — java — 80x16" showing the output of a test command. The output is as follows:

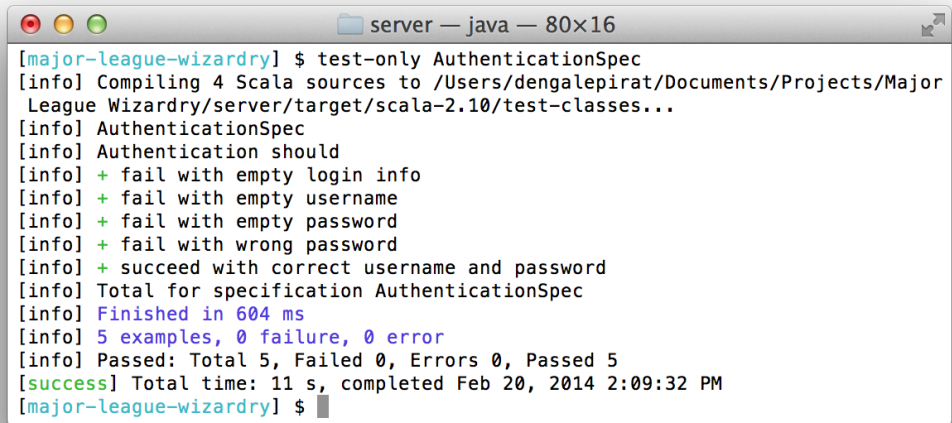
```
[major-league-wizardry] $ test-only UserSpec
[info] UserSpec
[info] User creation should
[info] + fail with 2 characters in username
[info] + fail with more than 16 characters in username
[info] + succeed with 16 characters in username
[info] + succeed with 3 characters in username
[info] + fail with no characters in password
[info] + succeed with 1 character in password
[info] + fail because user already exists with username
[info] Total for specification UserSpec
[info] Finished in 731 ms
[info] 7 examples, 0 failure, 0 error
[info] Passed: Total 7, Failed 0, Errors 0, Passed 7
[success] Total time: 3 s, completed Feb 20, 2014 2:28:21 PM
[major-league-wizardry] $
```

**Figure 7.1:** Account creation test.

What we are testing if it is possible to create an account with a username that already exist. It also test the maximum length of the username and tests if the password lives up to the requirements.

### 7.0.15.2 Login

This test focuses on login. It is based on the use case from Sec. 3.0.4.2.

A terminal window titled "server — java — 80x16" showing the execution of a test. The output is as follows:

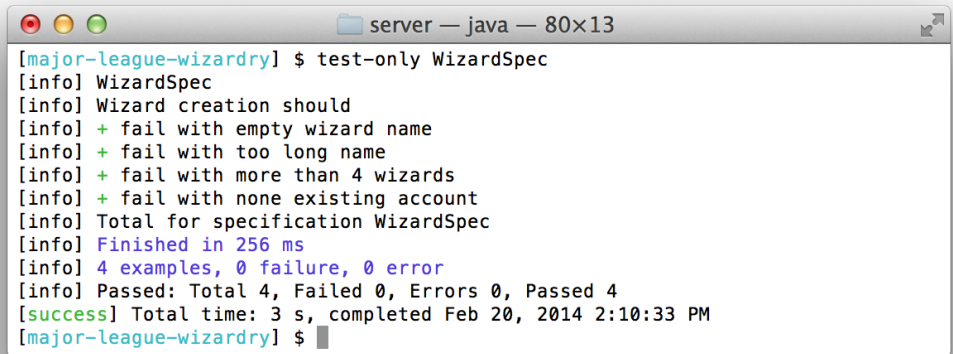
```
[major-league-wizardry] $ test-only AuthenticationSpec
[info] Compiling 4 Scala sources to /Users/dengalepirat/Documents/Projects/Major
League Wizardry/server/target/scala-2.10/test-classes...
[info] AuthenticationSpec
[info] Authentication should
[info] + fail with empty login info
[info] + fail with empty username
[info] + fail with empty password
[info] + fail with wrong password
[info] + succeed with correct username and password
[info] Total for specification AuthenticationSpec
[info] Finished in 604 ms
[info] 5 examples, 0 failure, 0 error
[info] Passed: Total 5, Failed 0, Errors 0, Passed 5
[success] Total time: 11 s, completed Feb 20, 2014 2:09:32 PM
[major-league-wizardry] $
```

**Figure 7.2:** Login test.

We tests if it is possible to login without a username and password. It also tests if it is possible to login without a correct username or password.

### 7.0.15.3 Create wizard

This test tests the creation of a wizard. It is based on the use case in Sec. 3.0.4.3.

A terminal window titled "server — java — 80x13" showing the execution of a test. The prompt is "[major-league-wizardry] \$". The command entered is "test-only WizardSpec". The output shows a series of test results, all of which are failures. The tests are: "Wizard creation should", "+ fail with empty wizard name", "+ fail with too long name", "+ fail with more than 4 wizards", and "+ fail with none existing account". The summary shows "Total for specification WizardSpec", "Finished in 256 ms", "4 examples, 0 failure, 0 error", and "Passed: Total 4, Failed 0, Errors 0, Passed 4". The final status is "[success] Total time: 3 s, completed Feb 20, 2014 2:10:33 PM". The prompt returns to "[major-league-wizardry] \$".

```
[major-league-wizardry] $ test-only WizardSpec
[info] WizardSpec
[info] Wizard creation should
[info] + fail with empty wizard name
[info] + fail with too long name
[info] + fail with more than 4 wizards
[info] + fail with none existing account
[info] Total for specification WizardSpec
[info] Finished in 256 ms
[info] 4 examples, 0 failure, 0 error
[info] Passed: Total 4, Failed 0, Errors 0, Passed 4
[success] Total time: 3 s, completed Feb 20, 2014 2:10:33 PM
[major-league-wizardry] $
```

Figure 7.3: Wizard creation test.

We tests if it is possible to create a wizard with an empty name. Tests if it is possible to create a wizard without an existing account. We also tests if it is possible to create more wizards then the limit.

### 7.0.16 Non-functional testing

For testing the non-functional requirements we will perform tests on the MySQL database to see if it can handle the amount of data we have if we were running at 1 million users and we will test the request times we specified in Fig. 3.4 using Gatling. Gatling is a stress tool.

As of this writing we haven't been able to make the above testing. The reason is that we haven't had time to setup up an additional infrastructure or the money to do so. This is of course an issue that we will have to work on but for now we just monitors the servers.

What we would have liked to do was to create a system which mimics the actual implementation on a smaller scale. Then we would have tested that system and seen how much traffic it could handle. Based on the tests we would then have been able to decide how many servers we would need and how big they should be.

# What we have learned

---

Here we are going to discuss what we have learned from this project and what we would have done differently.

When developing a scalable multiplayer platform for a possible international success it is necessary to make proper analysis of the what is needed from the platform. This means proper goals and requirements for the platform. Without the proper goals and requirements it is hard to estimate what is needed from the platform. We management to make an analysis of the requirements, but if we could we would have liked to base the analysis on a better foundation of information. We should have looked at how other cards games or games in general prepare for an international success and how they make sure that their multiplayer platforms are scalable. Another thing we think is missing in our analysis calculations on what is required in term of computation power, bandwidth and storage. We have some calculations on storage requirements but are missing calculations of the computation power and bandwidth needed. This is a problem because our final implementation only focused on scalable computation power and not bandwidth. So it is important to make a good analysis based on hard facts rather than assumptions.

Another thing we have learned from this project is to design for scalability. If you are making a multiplayer platform for an international success you have to design with scalability in mind. This means scalability in the platform in terms of architecture but also scalability within the applications them selves. The one mistake we made came from poor analysis not estimating bandwidth vs computation power. We have made sure that our application scales and that the multiplayer platform scales in terms of computation power, but we haven't design for scalability in terms of bandwidth. Had we done this then our final design might have looked a bit differently then it does now. Sp design for scalability,

but base the design on a well founded analysis.

We haven't made proper testing in this project. This is a problem because we don't have an estimate based on tests of what is required of the multiplayer platform. Without this testing we don't know for sure if the application actually scales or not in reality. We have design for scalability but testing is needed to make sure that the scalability actually works. Testing is one of the things we will strive to do more of in the future.

In general we think this project have been a success and we have learned a great deal about making a scalable multiplayer platform. As a proof the multiplayer platform runs our beta version of Major League Wizardry without problems for now, so we think it is a success.

# Glossary

---

**Game** - Referring to the game Major League Wizardry.

**Player** - A player is a participant in a game.

**Multiplayer** - A multiplayer game is where two or more players play against each other in the same game.

**Client** - Client describes a player and the player's devices.

**Peer-to-Peer** - Peer-to-Peer is a communication form where two client communicates directly with each other.

**Match** - A match is a game instance where two players play against each other.





## APPENDIX B

# Abbreviations

---

**MLW** - Major League Wizardry.

**CCG** - Collectable Card Game

**MVC** - Model View Controller

**JVM** - Java Virtual Machine

**DNS** - Domain Name Servers

**ACID** - Atomicity, Consistency, Isolation, Durability

**RAM** - Random Access Memory

**JSON** - Javascript Object Notation



## APPENDIX C

# Database design

---

Diagram showing the database design of the multiplayer platform.

