myVerein

Student Research Paper

Student Research Project

for the

Bachelor of Science

at Course of Studies Applied Computer Science
at the Cooperative State University Stuttgart

by

Frank Steiler

June 2014

Time of Project	34 Weeks
Student ID, Course	8216767, TINF12A
Company	Hewlett-Packard GmbH, Böblingen
Supervisor	Alfred Becker

Author's declaration

Unless otherwise indicated in the text or references, or acknowledged above, this thesis is entirely the product of my own scholarly work. This thesis has not been submitted either in whole or part, for a degree at this or any other university or institution. This is to certify that the printed version is equivalent to the submitted electronic one.

Stuttgart, June 2014

Frank Steiler

License

Preliminary note

Abbreviations or technical terms are not specially marked, but can be found in the list of abbreviations respectively in the glossary. While characterising or enumerating persons I will relinquish the feminine and just use the masculine form. This is because of simplification and should not be seen as any kind of discrimination.

This document is referring to the code uploaded to the Git repository located at `github.com/steilerDev/myVerein`. The relevant commit has the short checksum `10930fa` and was created on the 6th of June 2015.

Abstract

Within this student research paper a server client environment should be developed, which can be used to manage the members of a club and simplify and unify the communication between the board as well as between the users themselves. The need for this kind of application is based on the experience that a lot of – especially small – clubs are still paper based and the availability of comprehensive solutions is very limited.

After a careful consideration of the market demand it showed to me that a centralised open-source solution would be a perfect fit. This enables skilled members of a club to self host the application reasonably priced, as well as providing several other benefits shown in this paper.

After a thoughtful analysis the application's first version has been created, containing basic management and communication capabilities. The server was developed using the Java programming language, bundled with the Spring framework, while the iOS application was written in Swift together with Apple provided frameworks as well as third party extensions. The work respects several paradigms that should support the further development and extensibility of the environment. Nevertheless the version relevant for this paper includes all basic requirements and is fully functional, even though it is still in a beta phase, since the time did not permit an excessive user experience testing.

Contents

Chapter 1

Introduction

Especially in rural areas, today's world of clubs is dominated by small and medium size organisations. Based on my personal experience, these clubs are mainly managed using outdated techniques, while their communication channels are usually handled through existing social networks, electronic or non-electronic mail. This is due to the fact that most of the members and management boards of the societies are either not familiar enough with the complexity of the few modern club management solutions, do not know of their existence or don't see the implications of using them.

myVerein is trying to solve these problems providing an easy, comprehensive and intuitive solution for small and medium size clubs. Using the benefits of recent IT industry developments and the principles of cloud and mobility solutions it could become the management solution of the 21st century.

The following chapters will show you an analysis of the current market situation and what solutions can be found there. I will introduce the targeted group for this application and throughly discuss their wishes and needs within their club community. From there I will present the creation of the initial concept, how the process of planning and design evolved and the final implementation was done. The current state of the application will get shown and explained as well. At the end of this paper a retrospective on the whole project is given together with an outlook of what can be future improvements and work on the system.

1.1 Market analysis

A common principle in todays business is analysing all comparable products and solutions, that are used by potential customers. These solutions might not even be designed for that purpose, but not seldom misused by the user, because there is no real alternative available.

The very important part is the identification of the mistakes the competitors make and learn from their semantic errors [Hun15]. With these findings it is possible and important to draw parallels and use them to "model what is working for your competition and [...] not copy your competition directly" [Hun15].

There are several ways to identify competitors and keep an eye on their software solutions: Talk to potential customers to get a closer look at their current needs as well as to study their present working environment, use web research to find more brilliant answers to the requirements and needs of the clubs [Phi15] and take benefit of search engines to study the presentation and marketing concepts of competitors you are aware of [Phi15]. Last but not least you can gather additional information within social media platforms, at trade shows or on various email newsletter [Dah11].

At this point I'd like to thank all administrators with the *Freiwillige Feuerwehr Lohr* and *Melomania Helmstadt* for their remarkable support and feedback they gave me to analyse market needs. With their help I could identify just one substantial provider named Buhl Data Service (BDS). Nevertheless most of the clubs use combinations of various services and applications since there are hardly comprehensive and affordable remedies for them to find.

1.1.1 Buhl Data Service

The BDS company offers several solutions for the private and small to medium size business sector. Among them are applications used for the creation of tax statements and private banking. All solutions are tailored to the German market, since they only cover country specific topics [15c]. Besides these products *Buhl Data Service* also offers an application called *WISO Mein Verein*, whose purpose is to manage membership of a society as well as several other related tasks.

WISO Mein Verein was initially designed to only be used by a single user at once. Later the developer introduced a team edition, which offered the ability to upload the database backing the application to a centralised repository, but while the data is modified, it is still not possible for a second user to access the data. In general this process of "lock - modify - unlock" is very unintuitive and lacks of transparency. This reduces the usability dramatically, especially for administration without a special technical background.

The functionalities provided are widely ranged from simply listing all members, to the creation of automated debit transfers as well as storing and populating templates for non-electronic newsletter. The suite itself looks like a very mature product, only its graphical user interface and missing cross platform capabilities are not completely meeting today's standards [15b].

While this project was in its final state, BDS launched the public beta of a newly created online social network, targeting members and administrators of clubs. The service is integrated

into the *WISO Mein Verein* environment and called *Mein Verein*. An administrator now would have the possibility to import the data from his *WISO Mein Verein* suite and invite all users to join the new social network community.

The network is accessible through a web portal and an *Android* and *iOS* app. The portal is said to cover all expected functionalities of a modern social network targeted at members of clubs, including messaging, calendar functions and the ability to search, join and create societies.

Unfortunately the deadline for my research project did not permit an extensive testing of this portal but still raises questions with me: How well the managing of the club is integrated in the web portal. Specifically speaking if all functionalities known from *WISO Mein Verein* are available through *Mein Verein* or if the administration needs to maintain two separated data sets, that might become inconsistent.

Both solutions do not share a common pricing scheme. *WISO Mein Verein* is charged separately yearly, starting at 69,99 Euro for the single user version and 139,95 Euro for a team version [15b]. On the other hand the usage of the web portal *Mein Verein* is claimed to be completely free of charge and advertisement free. This leads to the question how BDS is planing of monetising that product. Since the portal handles private user data and also holds private bank details, in case it is connected to the *WISO Mein Verein* suite, it is crucial that this data is not shared with a third party.

1.1.2 Incomprehensive solution

A comprehensive solution covering both, managing a club as well as creating a unified communication channel is poorly realised through the *Mein Verein* portal. This network just recently left its beta state and is therefore surely a rather young, immature and unknown application not easy to establish in the back of *Facebook* and *WhatsApp*. Besides that you first need a license depended software package to get the full feature set combined with an unknown free-of-charge social network. Concluding this, most of the club members as well as their administrators, also aware of the high license fee, will still stay their own way of handling their business as long as they don't get any reasonable and affordable alternative.

While talking to several club members of different societies as well as from my own experience a couple of existing social networks and messaging applications are misused to connect the user and informing them about news concerning their club.

One very popular option, as already mentioned is the usage of hidden *Facebook* groups, creating a broadcasting channel for administrators as well as user. Depending on the size of these groups, the personal notification system of the user's *Facebook* will be flooded by unrelated and therefore uninteresting information. On top of that many user do not provide their actual

personal information or aren't part of the network because of privacy reasons and therefore the user register created by these groups is incomplete, especially if the administrator is trying to communicate using traditional channels as well.

Another famous way to communicate are instant messaging services like *WhatsApp*, where group messaging features are provided. Unfortunately every user receives a notification for each message send through the group. If, for example, there is a vivid discussion a user is not part of, the amount of received notification gets annoying and hard to catch up. This system also does not provide any user or event management at all. This needs to be done either through a separate channel, or the users need to manually maintain their calendar.

In total all these products have a unique use-case, but do not offer any comprehensive solution to the problem of managing a club and unifying its communication. Nevertheless there is a reason that these products are popular among users. Therefore the positive aspects should be closely examined and adopted, while resolving the shown problems.

1.2 User analysis

During the creation of a product and especially during the creation of an User Interface (UI) it is crucial to meet the requirements of the user and to never loose sight of them. A user is not going to spend much time on figuring out functionalities, which means he might be less satisfied with the product and consequentially use it less frequent, even though it implements all necessary requirements [Fra13]. Therefore every functionality needs to be arranged logically and specifically tailored to the target audience.

Usability engineering focuses on guiding through the necessary steps resulting into a satisfying user experience [Nie93]. The most important part is "the process of identifying users' needs to ensure a product can achieve specific goals effectively and efficiently, which results in overall satisfaction and success" [Fra13].

The importance of usability engineering can also be seen in the fact, that the International Organization for Standartization (ISO) is covering this topic as part of the ISO 9241 standard. The author is referring to section 210 of the standard, which is labelled "Human-centred design for interactive systems". It describes the general development cycle of an user interface, where the user analysis itself plays an important role [Gül15, p. 13].

The usability engineering describes 2 major kinds of methods to analyse an user: analytical and empiric [Gül15, p. 20]. Within an analytical approach the developer is trying to see things from a user's perspective, where empiric methods involve the user directly.

For this project the analytical *Persona* method was used to perform a user analysis. *Personae* are several relevant users, described by their archetypal properties. Additionally their behaviour

and objectives are presented [Gül15, p. 30]. This method is useful since the developer gets a better picture of the people using his application later and can therefor focus on their needs. Within this sections three key *personae* are presented, which are used in section 3.1 on page 21 to identify the requirements on the system. The presented findings are mainly based on personal experience, as well as information received by the different club administrators. The *personae* are ordered from the most to the least relevant one.

Jacob - 41 Jacob is father of two and lives in a small house with his family in the outskirts of a small German town. Besides his work at a global oriented medium sized technology company, he is a volunteering member at the local sports club. Generally speaking he is technology affine, but not an expert. His honorary post is managing all members of the club. This includes the listing of all active and passive members, as well as ensuring that everyone paid their membership fees. Additionally he has the responsibility to invite all members to events organised by the club.

The utilities Jacob is using to do his job at the club include an Excel sheet with all names and contact details, as well as a Word template providing the layout for event invitations. Recently he created an email distribution list to save the money he always spend on mailing the paper letters. Besides a handful older members everyone is receiving these emails. Nevertheless the constantly changing club structure and contact details are hard to manage and he spends a lot of time maintaining his member list.

Marie - 21 Marie is a student and recently moved out of her parent's house. She is an active soccer player since nearly 10 years at the sports club. Unfortunately she often missed special events organised by her club, because in the past she barely paid attention to the club mail, handled by her parents. Since she receives emails from her club on her smartphone she regularly updates her calendar according to the events, but slowly gets annoyed by the amount of data she has to organise besides her studies.

On top of that she is part of a *WhatsApp* group dedicated to her team where all kinds of topics are discussed, ranging from the last party to the next game. Over and above a soccer group dedicated to her club was created on *Facebook*, where additional information are shared.

Since she just shortly changed her address she is tying to get her new contact details registered at the club. Unfortunately after asking her coach and the head of the club to change the information she is currently waiting for a conformation from Jacob about the successful alternation. All in all she is not very satisfied with the current infrastructure and possibilities offered by the club.

Karl - 58 Karl is a farther of 2 and grandfather of 5. He has been living together with his wife in the same house for nearly 30 years. Even longer he had been a part of the local sports club.

Because of his age he is not as active as he used to be, but he helps everywhere he can. The electrician plays an important part in maintaining the club house and supporting teams during their game days.

Even though he owns a PC, which was a present from his children, he is not very confident using it. Therefore he does not receive any information through email, but through traditional non-electronic mail, as well as through personal contact with the responsible parties.

1.3 Unique selling point

The result of the market and user analysis above lead to the conclusion that there is a real need for a comprehensive solution. Considering all facts and findings discussed in this chapter, the key points of the application can now be derived.

The first and most important point of the application is to reduce the workload on the person managing the members of the club as well as advertising events, described as Jacob in section 1.2 on the previous page. Since this person is already spending at lot of his time on keeping the club running, mostly free of any charge, he should perform his work as efficient as possible. This should also include the possibility to delegate work to other people, e.g. every user could be responsible to keep his contact information up-to-date, without the need of the administrator being actively involved.

The second point of the application is reducing the personal organisation overhead for a member, like Marie from section 1.2 on the preceding page. Specifically talking a user should be automatically reminded of upcoming events and he has an easy way of checking a personalised, up to date schedule of his club related events.

Finally the group pressure on individuals to join one or multiple social networks or applications must get stopped by providing a safe and trustfully environment. People not able to do or not feeling confident with using latest technology should not get forced into using it. In contrast a viable alternative have to be offered to these users.

These three points basically cover the main idea behind this project. Of course, more items out of the findings in this chapter can get taken into concern but it would push the boundaries of this project to deeply discuss them. Nevertheless they are briefly shown in the outlook in chapter 5 on page 65.

Chapter 2

Concept

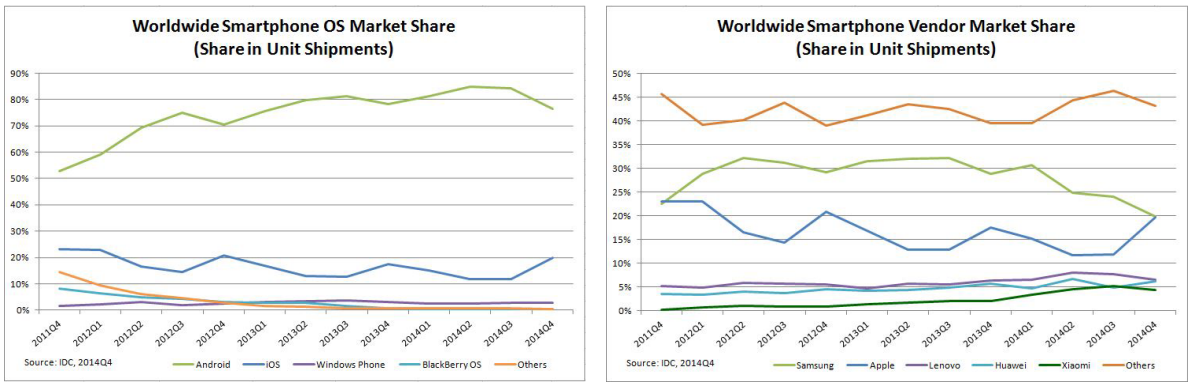
Within this chapter the initial thoughts, leading to the concept of the application, are described, respecting the findings from chapter 1. This chapter will present different approaches that can be chosen to solve the problem. An evaluation of these possibilities, based on feasibility, necessity and user value is going to be provided later in chapter 3 on page 21 and 4 on page 32.

2.1 Platform

Nowadays a solution can be created focusing on a wide area of devices and using several technologies supported by these platforms. In general, the workload increases with each platform and/or technology that is included into the development process. Concluding a well balanced trade-off, settled within reasonable time, has to be kept in view when choosing the platform and technology for a project.

The most conservative approach is developing a stand-alone desktop application. The interaction using mouse and keyboard is the most established one and possibly the only point of contact with modern Information Technology (IT) systems for non-technology oriented people. Nevertheless this platform and technology is concluding several negative aspects: A single operating system (OS) needs to be chosen and every additional one would include a massive amount of supplementary work, based on the assumption of not using cross platform technologies, like *Write once, compile anywhere (WOCA)* or *Write once, run anywhere (WORA)*. On top of that time spend online on mobile devices recently surpassed desktop application [Mur14], making them less important.

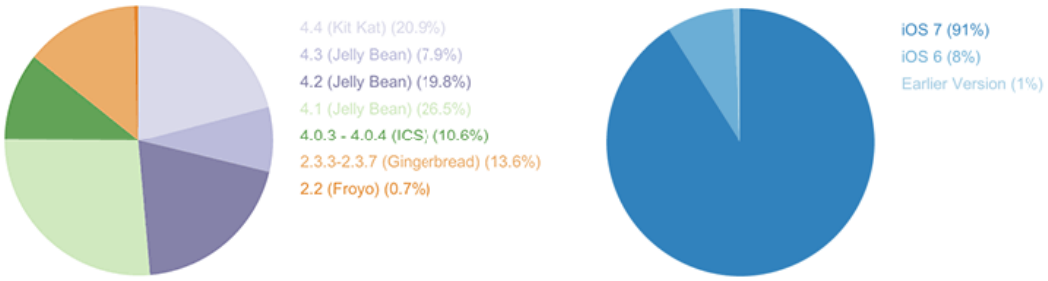
As shown in the last paragraph the popularity of mobile applications rose tremendously within the last couple of years. Therefore I believe it is essential to take this platform into consideration. When looking at the market shares of mobile OS, shown in figure 1a, it seems obvious that an



(a) Operating Systems (b) Vendors

Figure 1: Smartphone market share, Q4 2014

application should initially be developed for the *Android* platform, since it holds a market share of 76.6% [IDC15a]. Later the development for *iOS* (19.7%) and *Windows Phone* (2.8%) might be a logical consideration [IDC15a]. But when you take a closer look at vendor market shares, shown in figure 1b, this picture is not so obvious anymore. *Apple*, the only vendor distributing devices with the *iOS* OS, has nearly the same market share as *Samsung*, the biggest vendor of *Android* running mobile devices [IDC15b].

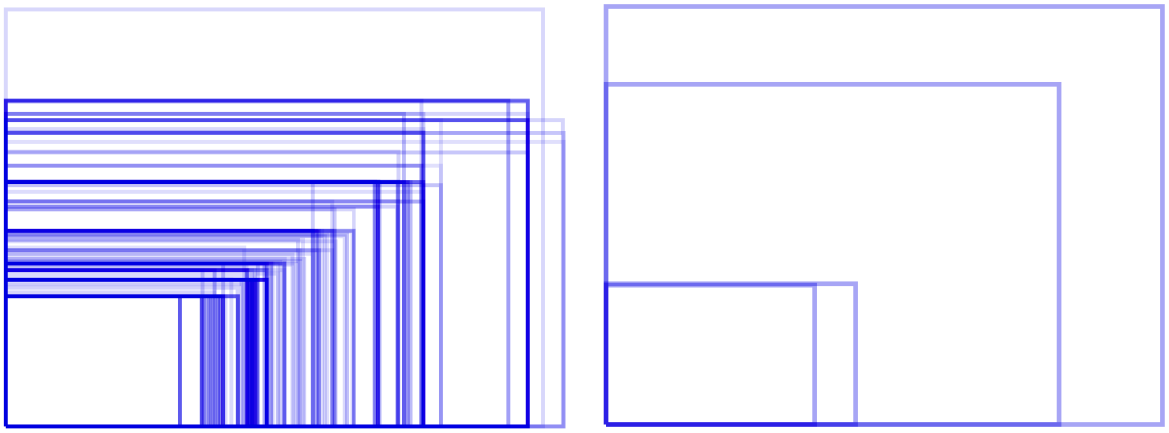


(a) Android (b) iOS

Figure 2: Mobile OS fragmentation, retrieved from [Ope14]

Figure 2 puts the myth of the wide spread market share of Android into new perspective. A lot of different *Android* versions and slangs by various distributers are on hand and these devices running on manifold software releases (see figure 2a) make it more than a complex task to support all devices by one exclusive application. Above this, the *Android* platform is used on unenumerated different mobiles and tablets of distributers you hardly know, with specifications not identically even with same software version. In figure 3 on the next page you see a diagram

giving a feeling to this diversity of equipments, by showing all available screen sizes and ratios of *Android* and *iOS* devices in 2014. With this background, a 20% market share of an OS where all devices having consistent release states (see figure 2b on the preceding page) and the amount of different consoles is clearly laid-out (see figure 3b) would make the Apple way with its *iOS* system a comfortable decision. It would lower down the developer's work and increase the quality of the software.



(a) Android (b) iOS

Figure 3: Screen sizes of mobile devices grouped by OS, retrieved from [Ope14]

Concluding, when creating an app for the *Android* platform the developer is facing several design decisions, which either increase the complexity of the program or reduce its functionality or accessibility. Choosing the *iOS* platform he can easily create an application which is optimised for all available devices, as well as exploit all recent features offered by the OS without excluding a significant amount of users, who are not running on the latest version.

Another question to answer is a possible separation of functionalities, which would decrease the complexity of a single application, but eventually increase the development workload, since the amount of several smaller application is higher.

In contrary of creating an application that includes all options offered by the system there might be a dedicated application for administrative tasks and member related tasks. This would flatten the intricacy of the respective app and hide uninteresting features from the regular user, while only creating a minor overhead for the small group of administrators. Since the tasks might be very different, even the adoption of a different platform or technology could be considered.

2.2 Architecture

The architectural decision highly depends on the platform and technology chosen. When taking this decision, it comes to choose between a multi-tier, also known as client-server, and a single-tier, also known as client-only, architecture.

Compared to a multi-tier web application that is accessed by several users simultaneously, the complexity of a single tier desktop application is surely easier to manage, though it comes along with several negative implications. The biggest impact for the administrator would be the missing possibility to delegate certain tasks to other members. Additionally, the locally stored information cannot be accessed by any user, because of availability and security limitations of a personal computer.

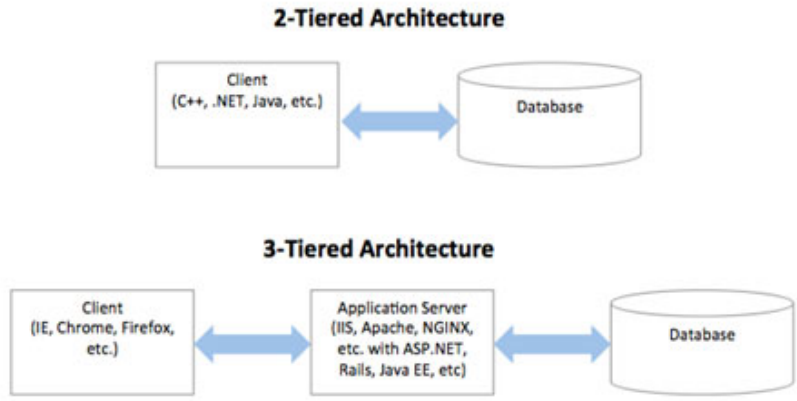


Figure 4: 2-tier and 3-tier architecture, retrieved from [Wri15]

In opposite to that, a multi-tier architecture enables the user to access and modify data through a centralised data provider. As shown in figure 4, within a 2-tier architecture, the data is directly available, where a 3-tier architecture adds another abstraction or presentation layer between the raw data and the user. The decision, which kind of architecture to choose, highly depends on the application itself.

The security of a 3-tier architecture might be more sophisticated, since the logic provided by a dynamic web server offers more possibilities to define user groups and access rights. If the policies are not well designed, this might lead to unintended access of resources. I assume that a 2-tier architecture might be a better pick, since its configuration possibilities are more basic and lead to less problems [Wri15].

Ending up the 3-tier architecture is implicitly more complex, leads to higher costs and presumably poorer maintainability. On the other hand it offers a better performance and its usability is more satisfying, since it provides a presentation layer [Wri15].

2.3 Functionality

With the key functions given in section 1.3 on page 6 we can derive the distinct functionalities of the app. Due to the limited timeframe only a subset of the presented conclusions and ideas will get realised within this confined student research project.

2.3.1 Administrator related functionalities

Initially the administrator should be able to represent the organisational structure of his club within the application. As a club is hierarchical organised, containing several divisions, with an arbitrary amount of subdivisions, its structure is highly dependent on the respective club and needs to be as customisable as possible.

As shown in chapter 1.3 on page 6 the administrator should have the ability to delegate the administration as easily as possible. Concluding it has to be technically feasible to assign a second level administrators to each division, splitting up the workload into easily manageable pieces. It is important to note that the second level administrators only have restricted access and modification rights. For example, they should only be able to view personal information of users they directly administrate.

To manage the users of the club, the administrator needs to be able to create them. Each user should be associated with one or more divisions. On top of that several personal information need to be stored together with the user. These might include his address, banking information or club related information like his membership number or the dates he became active, passive or resigned from the club. This structure is dependent on the data the club needs to operate and therefore the possibility should be given to add custom fields to the profiles of user. Upon the creation of their persons' profile, the members might be informed about their new account through an automated email, where they are introduced to all features of the system and get invited to start using it.

Another time-consuming task of an administrator is the management and promotion of internal events. Those events should be created by an administrator, who is able to invite the user, he is administrating. For organisational purposes it would be useful to receive feedback from them about their participation of certain events. To simplify the process, user should be

invited on a division based selection, concluding the administrator does not need to pick each relevant member but only the relevant division.

The process of collecting membership fees is complicated and in-comprehensive, since some people use an automated debit transfer, some are transferring the money manually and others are paying cash. This process is hard to track and the application could simplify and centralise it, by implementing a fee tracking system. A first step could include the possibility for an administrator to manually tag every member upon the payment of their membership fee. This feature could be further enhanced, by monitoring the club's bank account and automatically tag the respective user. This feature would require a high level of trust towards the application and its creator, since it would have direct access to the club's bank account.

Besides the promotion of events, a club needs to inform the users regular about news concerning certain division or the club. Some of these information might get shared with the public in parallel which would resolve in an unnecessary overhead. Concluding, besides the implementation of a personalised news feed, the system could also offer the possibility to connect to webpage hosting frameworks, like *Wordpress* or *Joomla* and social media platforms, enabling the administrator to share information with the user's of the society as well as the public.

Additionally the application should also respect users that are not using the application, because of security concerns or their aversion towards modern technology. It is also possible that certain users do not own a system that is supported by the application. Concluding the software should support a fall back solution for these persons and offer information through standardised channels, like electronic or non-electronic mail.

Finally the application could aggregate interesting information for the administrator and present them within a dashboard. This might include upcoming events or statics about the usage of the platform. Last but not least the system could provide reminder about anniversaries or milestone birthdays of users, which would enable the club to reliable provide congratulations.

2.3.2 User related functionalities

As shown in chapter 1.3 on page 6 two of the three presented key options are user oriented. Therefore the final application is probably going to have a fairly big amount of user related functionalities.

To get rid of different messaging technologies, a centralised group chat feature should be offered. These group chats should be created automatically for each division. Users that are part of the respective division will have the ability to participate within this group chats. For the sake of simplicity, a group chat should only be available for divisions where this feature is useful, so the administrator has to be able to select divisions where this feature is not available. To reduce

the amount of notification received through a group chat, the user might only be reminded about an incoming message if he is explicitly mentioned.

The application should also support the user to manage his personalised club calendar and remind him about the beginning of upcoming events. The user should only be invited to events that apply to divisions he is part of and respond to their respective invitation. A connection between the calendar system of the application and the native system of the particular OS could be given, by using a standardised technology, like *iCal*.

It is a difficult task for a club administrator to maintain the complete records of every user, especially if the user's contact details change frequently. It would be a good way to delegate this task to them. By enabling the user to update his profile on their own, the process is simplified for the administrator and the club member. It also enables the user to give his appearance a personal touch, which might be discoverable by others, permission given.

The application could also provide a counterpart for the news published by administrators. This could be a personalised newsfeed within the app, an automated email newsletter system or Rich Site Summary (RSS) feed of the club.

Another problem a lot of clubs are facing is the problem of collecting useful and up to date information that can be used to promote the club or events on their webpage, in the newspaper or internally. The system could enable every user to hand in articles written about recent activities or to upload pictures and videos that are club related. These information should only be published if they are approved by an administrator, or handed in by a trusted user.

2.4 User Interface

As discussed in chapter 1.2 on page 4, one of the most crucial parts of a system is the UI, since this is the only part of the system the user is directly interacting with. A bad designed UI can increase the complexity of an application, but also decrease user satisfaction. The design concepts for an administrator focused web and a member focused iOS application are presented in the following sections.

Because of the fact, that the navigation concept is a crucial piece of these applications, it deserves special attention. A very popular option to manage the navigation on webpages and mobile application is the so called "Hamburger Menu" [Abr14]. This widely used concept is represented by an icon consisting of three parallel horizontal lines, which indicate more, currently hidden, available options. Unfortunately there are several problems with this concept, including but not limited to lower discoverability, less efficiency, a clash with platform navigation concepts and the inability to get a glance view on the available options [Abr14]. Therefore a main objective for this section is the presentation of platform specific alternatives.

2.4.1 Web application

In general a web application is able to display its navigation information either horizontal or vertical. A combination through dropdown boxes is possible but should be avoided, because of the lower discoverability and less efficiency. [Cre15]

It is highly recommended to put only a small amount of items in the navigation bar. It is a proven fact, that the human short term memory is only able to hold an average of 7 items, making the user forget items from top of the list while scanning through a labyrinth of navigation items. When limiting the amount to 5 items this would increase the importance of the remaining items, while increasing simplicity of the webpage. [Cre15].

Psychology studies have shown that the most important items within a list are the first and last ones. Therefore a designer should consider the placement of important navigation items either at the beginning or end of the list. [Cre15]

Please also note that a user faster adopts well known structures and navigation concepts. Using a non-standard navigation concept could decrease the usability even if it solves problems of existing solutions, just because of the fact, that the user is not used to it. Especially placing the navigation items in an unfamiliar place could lead to confusion. [Cre15]

Furthermore, a user should always be aware of his current position within the application. He should be given the possibility to change his current position at any time. This requires the need of always visible navigation elements.

While all of this holds true in general the exceptions of these rules, are the concepts that a user is going to remember later. Although creating a new navigation concept, which is respecting only a subset of the above stated suggestions, or none of them, requires a lot of conceptional work. [Ham13]

Besides the consideration of different navigation concepts I also will introduce my overall design of the different functionalities. However for the sake of simplicity only their mockups and final designs are presented in chapter 3.4 on page 28, respectively 4.3 on page 59. Especially since the formal definition of all functionalities, that are going to be implemented within this project, has not been given yet, this list would exceed the scope of this paper.

2.4.2 iOS application

2.4.2.1 Navigation

When creating an *iOS* application, a lot of design elements are pre-defined by the OS. On the one hand this leads to well known design and navigation principles and a well designed piece of software. On the other hand the applications are not very distinguishable based on their design

and might become boring for the user. Concluding using an adjusted layout or navigation concept will lead to a better recognisability of the brand.

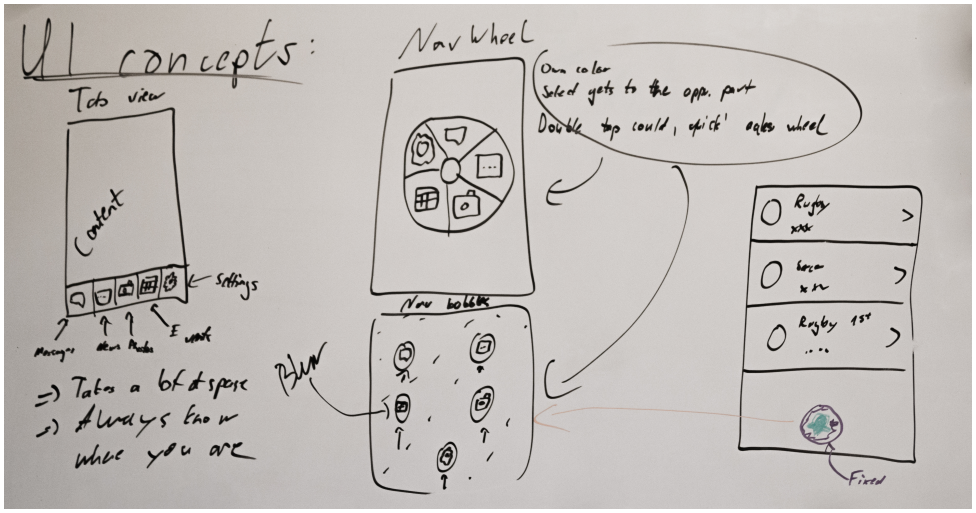


Figure 5: Early navigation concept drawing, own figure

Figure 5 shows early design concept drawings. On the left, a standard tab bar concept is presented. The bar contains a total of five elements, including *Messages*, *News*, *Photos*, *Events* and *Settings*. This design is an example of a predefined layout provided by *Apple*. It is important to note that there is no default design that makes use of the above described "Hamburger menu". The tab bar is therefore a viable alternative, which would fit into the design-system of *iOS* and consequentially might perish between the millions of other available application. Nevertheless very popular companies use this concept with minor adjustments, including *Facebook*, *Yelp* and *Twitter*.

In opposite to that the right side of figure 5 describes a very different navigation concept. The author did not find this concept in any other application, although it was inspired by *Todo Movie*'s sharing layout, as well as *Google*'s material design. The main idea is the positioning of a fixed menu button in the lower part of the screen. When pressing this visual element, one of the two frames shown in the middle of figure 5, would appear, revealing the menu. This might either be a "Navigation Wheel", that could be operated using swipe gestures, or the available items could appear by moving up.

This concept seems to have more advantages than disadvantages. By choosing a full screen presentation of the menu items, the amount of elements could be increased from a maximum of five, when using a tab bar, to about seven [Cre15]. On top of that, most of the user prefer well designed swipe gestures over taps, which might be adopted using the "Navigation wheel" menu. Since the button is a fixed item on top of most of the screens, this would attract the attention of

the user and improve their engagement. On the opposite, this concept has similar downsides as the previously presented "Hamburger menu", which is, amongst others, missing the possibility to glance at available items [Abr14].

2.4.2.2 Chat overview

For the sake of simplicity only the layout of the chat overview page is going to be discussed besides the navigation concept. Figure 6 shows these screens of the two most established chat applications on the *iPhone*, *WhatsApp* and *iMessage*. The concept of both overview pages is very similar, since they both use a table view with slightly different cell layouts. This is very efficient, when there is the need of showing a big amount of items in a precise and easy to retrievable way.

(a) WhatsApp

(b) iMessage

Figure 6: Chat overview screenshots of established messaging apps retrieved from [Inc15] and [Stü13]

Nevertheless the amount of available chats within *myVerein* is very likely being limited to a foreseeable small quantity. Therefore, other concepts, besides the established list might be considered to increase the uniqueness and recognisability as well as usability. The proposed concept includes two circles per row with a caption, each representing an available chat. The chat with the latest message is shown on the top left of the screen. Every chat bubble would have a

caption stating the name of the division, it is representing. The image within the bubble would show the avatar of the user, who sent the latest message. This helps to estimate the importance of the message. A "tap-and-hold" feature could also be included, showing a preview of the last couple of messages within the overview page without entering the chat view.

2.4.3 Visual branding concept

When you want to have a unique and recognisable brand you need to have a visual branding scheme that will be associate with it. For example, seeing a white tinted "F" in front of a blue background instantly leads to the "Facebook" brand. Therefore this section is devoted to the concept behind the creation of such a branding scheme.

2.4.3.1 Colour scheme

The field of colour theory is a very complex one, therefore the time only permits a very basic analysis on this specific matter. The first thing to note is that different colours have different, very distinct meanings. Even a slight change in saturation might lead to a completely new impression of a colour [Cha10a]. For example red is considered as a violent or aggressive colour, while green is more balancing, relaxing and down to earth [Cha10a].

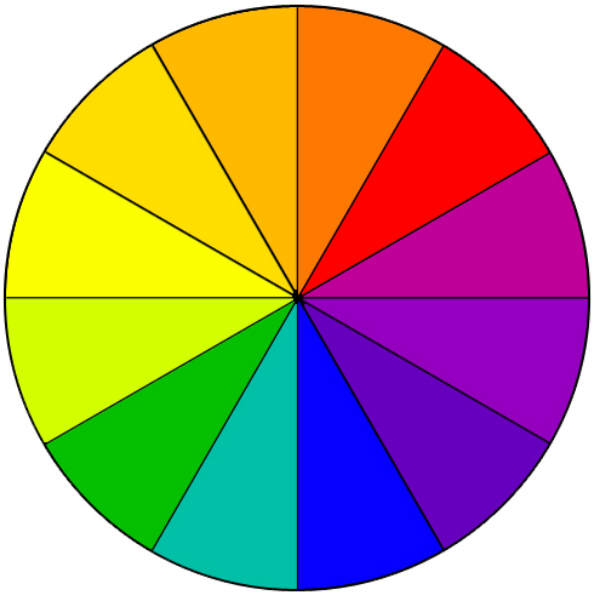


Figure 7: 12-spoke colour wheel, retrieved from [Cha10b]

After choosing an appropriate base colour we will create a colour palette, since obviously a single colour would make the application not only boring but would also restrict functionalities

as highlighting certain entries using a different tint. There are several techniques to create a colour pattern. One is using a *Monochromatic* scheme where the base colour is only altered using different tones, shades and tints [Cha10b].

Figure 7 on the previous page is showing a 12-spoke colour wheel, which can be used when creating an *Analogous* colour scheme. It is created by picking three colours that are next to each other and adjust their tones, shades and tints, similar to the *Monochromatic* method [Cha10b]. With this technique we can create a set of colours, that are very close to each other.

The *Analogous* colour scheme with its nearly similar colours is somewhere regarded to look boring, the *Complementary* scheme introduces a wider selection by choosing the ones opposite of each other on the colour wheel [Cha10b]. Dealing this contrast adds depth to the palette.

The most diverse colour scheme is called *Tetradic* or *Double-Complementary*. It is regarded as the most difficult chart when creating an efficient palette [Cha10b]. It is "based on the tetrade - the foursome of colors evenly distributed on the fourths of the color wheel" [Sta11]. Because of the introduction of four different colours, this scheme is considered more nervous than the other ones [Sta11], the primary colour should be used for the main design, while the others are only sparely used for accents [Cha10b].

2.4.3.2 App icon

Surely the most important identifying feature of an app is its icon. It needs to be well designed, since the users are going to put it on their home screens or desktops and probably see it every day. On top of that it needs to be interesting enough to regularly click on it, but not too distractive. Last and most important the user has to recognise this icon an associate it promptly with the product. [Fla15a]

Figure 8: App-Icon design within the productivity category, retrieved from [Fla15a]

Let's put the focus on creating an icon using an *iOS* app as example. The first thing a good icon needs is scalability. When developing an application for the latest *iOS* version, you have to bundle 12 different resolution for the application icon and on every appearance it has to look great. Concluding "[o]verly complicated icons that try to cram too much onto the canvas often fall victim to bad scalability" [Fla15a], therefore the designer should embrace simplicity and check that the icon suits to various different background colours. [Fla15a]

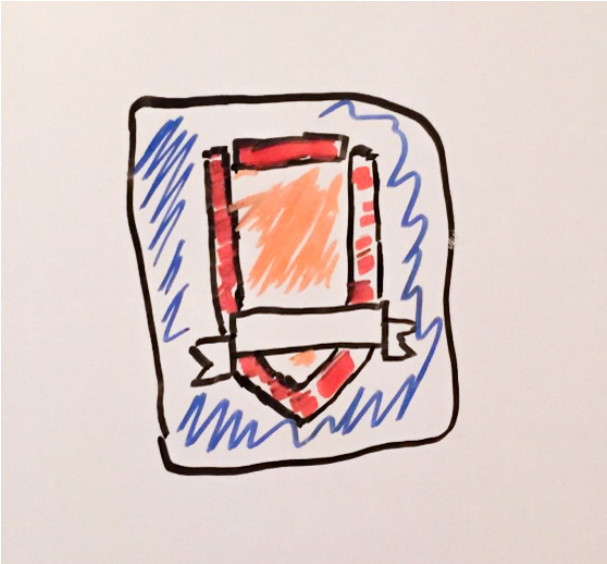


Figure 9: Early application icon sketch, own figure

While keeping the icon scalable it is important to create a design that is recognisable. Therefore the app should not be overcomplicated. When trying to create such an icon first observe competitors in the same field as well as other popular app icon designs. [Fla15a]

When looking at the competitor's app icons, please note not to copy any existing design because uniqueness is a crucial aspect of icon layout. It needs to attract the user's attention and have to stay away of any competition. A negative example is the productivity category on the app store, shown in figure 8 on the preceding page. Since all of the shown apps are trying to solve a similar problem, they all end up having a more or less identical icons. When choosing a single glyph, like a checkmark, the designer needs to make a difference by using a rich colour design. [Fla15a]

Last but not least you have to avoid text within an app icon. On the one hand, text scales very badly, on the other hand the app is always accompanied by a caption, stating its name. Consequently a logo should not be used as an icon, since these two are very distinct [Fla15b].

Nevertheless a single letter, used for example by the *Facebook* app, might be a good way to create a distinguishable brand.

After considering all of the above stated concerns and doing some brainstorming, the best suitable and unique association with a club, that I found, is an emblem. Figure 9 on the previous page is showing an early sketch for my logo. This very generic object is widely used as a foundation for many logos of different clubs. By leaving the banner empty, it gives space for the user's imagination to place his club name or description into it.

It was considered to form the emblem in a way to resemble to letter "V". This can lead to an association with the application's name *myVerein*, similar to the *Facebook* logo.

Chapter 3

Design

Using the initial thoughts and concept alternatives presented in chapter 2 on page 7 a design is going to be developed within this chapter followed up by introducing its final implementation. Concluding the discussion from the last chapter, I decided to provide the system with:

1. A three-tier architecture (see section 2.2 on page 10)
2. A member facing mobile app for *iOS* (see section 2.1 on page 7)
3. An administrator facing web application (see section 2.1 on page 7)

3.1 Requirements

The complete formal list of all objectives can be found in the software requirement specification [Ste14b] and the server requirement specification [Ste14a]. They are all based on the options shown in chapter 3. The decision leading to the final implementation are consequential deductions out of the findings from chapter 1 on page 1.

3.1.1 Obligatory requirements

This section states the obligatory requirements of the project. The functionalities described in this section are the most basic ones, that are needed to operate the system. Therefore the fulfilment of the following criteria is mandatory.

3.1.1.1 iOS application

Through the provided *iOS* app the user needs to be able to authenticate himself against the back-end server. As shown in section 1.3 on page 6 one of the main objective of this software

is the unification of the communication channels. Every registered user has to be able to use the app to review his personal event calendar, as well as sending and receiving messages. By providing this information without the need of any interaction from the user, these functionalities provide a maximum amount of usability for every persona presented in 1.2 on page 4.

By using the auto layout capability of the *iOS* development suite, an application's layout is automatically adjusted to the screen size and ratio of the user's device. Nevertheless an *iPad* has obviously a bigger screen than an *iPhone* or *iPod touch* and therefore a layout optimised for a smaller device might be less user friendly on a tablet. Since the information provided by the system are going to be checked at home as often as on-the-go, a phone app is more applicable than a tablet app. Therefore it is an obligatory requirement to optimise this application for the operation with the latest *iOS* device generation, more specifically the *iPhone 6*.

To be able to ensure the usage of the latest features provided by the *iOS* Software Development Kit (SDK) throughout the next couple of years, the application needs to be developed using the recently released programming language Swift.

3.1.1.2 Web application

An administrator obviously needs to be able to provide the information requested by the member's app through the web application. This includes the possibility to schedule an event, and create user. The administrator needs to populate each user's profile with information provided by the user. On top of that the user needs to be able to modify the basic behaviour of the system as flexible as possible.

Since the system is storing private user data it is necessary to secure the application and pursue the Least Privileges principle. Therefore the log in should only be allowed for users that are at least second level administrators and these user should only be able to view users directly managed by them.

As shown in section 1.2 on page 4, the administrator has only basic skills in using technology. Concluding the above discussed functionalities should be as straight forward as possible. It is especially difficult when trying to combine usability and flexibility. The implementation of this trade off has to be considered closely.

3.1.1.3 Back-end

Since the application needs to be accessed from different devices, only a 2- or 3-tier architecture is applicable for this system. After reviewing the findings in section 2.2 on page 10 the decision was made to use a 3-tier architecture, based on the possibility of a more sophisticated security

policy and an advanced business logic. The design should also keep scalability in mind, to ensure the flexibility to manage growing clubs.

To run the back-end application on a variety of operating systems the cross platform programming language *Java* has been chosen. To take advantage of the latest features of *Java* the 2014 released version 8 is going to be used within this project. To simplify the development of a web application with *Java* the open-source *Java Spring* framework was chosen. It is providing straight forward techniques to create a web server, which is incorporating sophisticated security features and the access of a durable data provider, which is also needed within this project.

3.1.2 Optional requirements

The requirements presented in this section are not implicitly necessary for the system to work, nevertheless the overall usability of the system would be highly reduced, if these functions would not be part of the initial implementation.

To enhance the convenience of the application, the organisation of user into divisions should be introduced. The administrator needs to be able to assign divisions to the user's profile through the web application. This includes the need of being able to create and modify the division. As discussed in 2.3 on page 11 the arrangement of the divisions needs high flexibility. This would enable the focus of received messages on certain topics, by automatically creating groups, according to the divisions a user is part of. Following the same principle, the invitation to events could be filtered using the membership of users within divisions.

The web and *iOS* applications should both have a user friendly and sophisticated user interface. This implies the need of a well designed navigation concept and a unified colour scheme. The details of the UI design are presented in section 3.4 on page 28.

The platform could also support the sharing of user taken photos with the club's administration. Then the *iOS* application might offer the ability to shoot photos, or select previously taken ones and upload them to the server. Consequently the administrator has to have the ability to review the uploaded pictures and download them, so he can use them for any club related activity.

3.1.3 Additional requirements

The following features are not required and their implementation is probably exceeding the time frame provided by this project. Nevertheless their definition is given here since they would increase the usability significantly.

The system could support the news feed feature addressed in section 2.3 on page 11. On the one side, this would require the web application to enable the administrators to publish news and relate them to existing divisions. On the other side the *iOS* application has to be able to

display a user specific news feed, presenting all relevant information. On top of that the back-end could provide a RSS feed, an automated email newsletter containing summaries of the published news, or extensions for blog hosting webpages, like *Joomla* or *Wordpress*. This would improve the integration of user, that do not own a supported smartphone, or is not participant of the system for whatever reason.

The *iOS* application could gain additional functionalities, including the possibility to access multiple *myVerein* instances from a single device. This is due to the fact that people committed in one club might also be dedicated to others. If both of them are running an instance of *myVerein* it would be very inconvenient for the user to constantly change between the accounts within the application. Above that the application could be optimised for the operation on an *iPad*, which would improve its usability at home.

3.1.4 Non-requirements

It is important to delimit the functionalities of a system and define items to NOT be part of the development at all. The following features fall into this category of non-requirements.

The *myVerein* system must not replace a club homepage. Therefore it should not provide any non-member facing functionalities or information. This means that the system is not replacing software packages like *Joomla* or *Wordpress*. Nevertheless as outlined in section 3.1.3 on the previous page, it might be possible to integrate *myVerein* into these systems.

Finally the *iOS* application is not designed to operate stand-alone, as a private club managing application. A user who wants to use the app needs to be part of a club, which serves, configures and organises the *myVerein* environment.

3.2 Architectural model

After evaluating the findings from section 2.2 on page 10 and section 3.1.1 on page 21, the decision was made to use a 3-tier architecture, keeping scalability in mind. Figure 10 on the following page shows the planned architecture of the system.

The central server software is written in *Java 8* and uses the *Spring* framework. This mature open source framework provides sophisticate security configurations as well as several access frameworks for data provider. With this platform several other additional frameworks are available, including messaging and social media connection, which can be used to easily extent the functionalities of the application. The complete business logic is implemented on top of this framework.

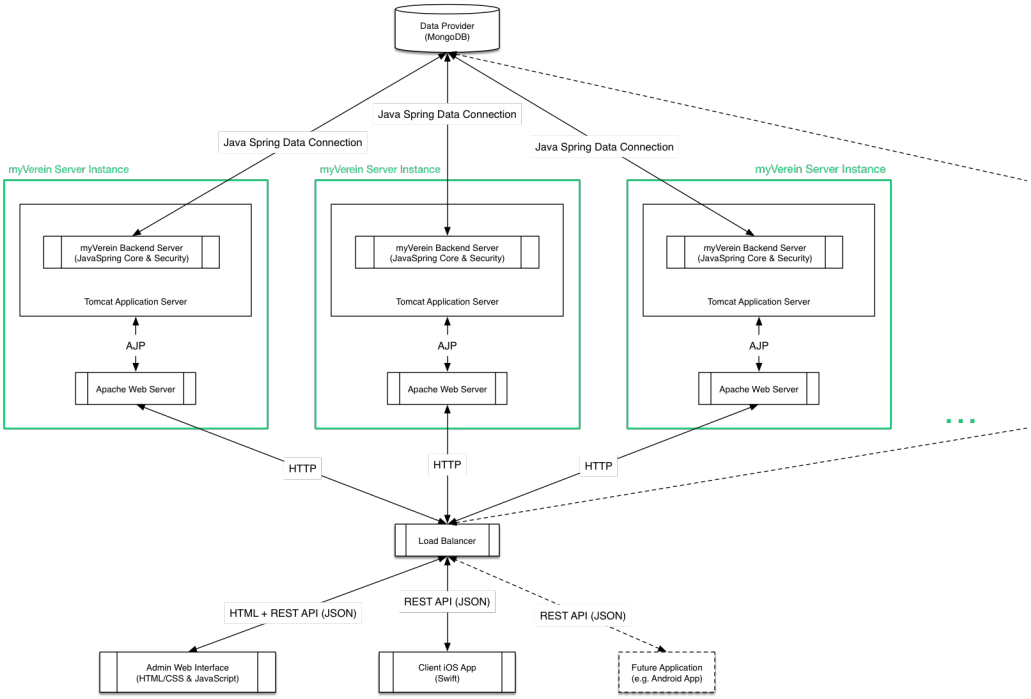


Figure 10: System architecture, own figure

Since the server application needs to be run within the Java Virtual Machine (JVM) and be able to communicate with other machines over the internet, an application server is necessary. For this test environment *Tomcat 8*, connected to an *Apache* web server using the Apache JServ Protocol (AJP), was selected to host the application.

The data provider was chosen to be a *MongoDB* not only SQL (NoSQL) database system. This schema free database enables flexibility, as well as extensibility and speed, which is far ahead of existing relational database applications [Mos14]. The database itself is connected with the server application using the *Spring Data* framework, which supports the use of *MongoDB*. This framework provides a concise and object oriented access to the database having the native *Java* driver developed by *MongoDB*. Most importantly it integrates very well into the *Spring* framework.

To ensure high flexibility the system is designed to operate behind a load balancer. Therefore each system must be stateless retrieving all necessary information and configuration from the central data provider. In this case, the necessary computing power to fulfil all requests can be automatically allocated and deallocated, by adding or removing *myVerein Server Instances*. This would allow a cost efficient, high available, large scale operation within a cloud environment.

The front end of the system consists of an *iOS* app and a web application. Both of them use a Representational State Transfer (REST) Application Programmable Interface (API) to communicate with the system. The API is designed to be used by many different application and should therefore simplify the process of adding new front-end components to the system. Implicitly the development of these components can be done independently from the server as long as the existing API is not changed.

The web application is intended to be used by the administrator to manage the system, therefore only the features relevant for administration require access. The usability would benefit, if the interface behaves like a desptop application and avoids page reloading using the principles of Asynchronous JavaScript and XML (AJAX). On top of that it should be designed to be as mobile friendly as possible. To create a rich UI and feature set, while keeping the workload on the developer manageable, several Cascading Style Sheet (CSS) and JavaScript (JS) frameworks are going to be used.

The *iOS* application's target group are the member's of the clubs using this system. The range of functions is therefore being limited accordingly. The application will get developed using *Apple*'s latest programming language Swift. Additionally the dependency manager *CocoaPods* is chosen for adding existing frameworks to the project. This is done to simplify the development of the application, while increasing the usability.

3.3 Database layout

As presented in section 3.2 on page 24 the data provider for this project was selected to be the NoSQL database system *MongoDB*. Even though this system uses a dynamic schema, it is important to model the organisation of the data in advance, to later ensure a reliant and consistent operation. Nevertheless this schema is very likely to change during development, but the dynamic aspect is very helpful within the context of the agile development approach.

A database schema can be depicted using an Entity Relationship (ER) diagram. This visual representation of a database layout was originally introduced for relational databases, anyway it was adopted within this work to the context of NoSQL databases. The diagram created during the design state to get an orientation during the development phase can be found in appendix A on page xii.

Above all, each set of entities in the context of a document oriented database like *MongoDB* - better known as document store - is going to be described in detail in the upcoming chapter as well as the connections between the documents.

3.3.1 User

The user information are stored within the *User* document store. The data is organised according to the *One-to-Many Relationships with Embedded Documents* model [Mon14, p. 141].

On the lowest level all required information about the user, like his email or hashed password, are stored. His private information, that can only be viewed by the administrator, are saved within the *PrivateInformation* array. The *PublicInformation* array is holding all optional information stated by the user. These information can be accessed by every member of the club.

As each user is part of one ore more divisions the membership within the divisions is stored as nested arrays. These data structures hold the foreign key to the division as well as the date when the user joined the division.

3.3.2 Division

Each division is an entry within the division's document store. It is defined by its name and short description.

Divisions are organised hierarchical within a club, so this structure has to be represented within the database. This data structure is posed within MongoDB using the *Tree Structures with an Array of Ancestors* model [Mon14, p. 149]. Within this model you can store the direct parent of the node, as well as an array of all ancestors, to easily query them. This behaviour is essential, since the system must be able to subscribe the user to all chats within his divisions as well as all chats above.

Each division can get administrated by up to one second level administrator, who gains access to the administrator panel through this position. In lack of him the super admin takes this role. This super user is defined by being administrator of the root division.

3.3.3 Message

Each member of a division automatically joins a group chat between all members therein. To ensure privacy for each chat, messages are only saved on the server as long as necessary. Therefore a *Message stack* document store is created. Every sent message is stored there until the recipient accesses it, or the system deletes the message after its expiration. The expiration of the message is handled by MongoDB through the functionality to set a Time-To-Life for collections [Mon14, p. 198].

Each entry contains the message to one member of the group chat. When the user syncs his application with the server, the server returns all messages from the stack for the user. The rows are deleted as soon as the user receives the message.

3.3.4 Event

A core feature of the application is the creation and management of events for each division. The events are managed within the *Event* data store. An event invites whole divisions, and every member can send a response to the invitation.

The invited divisions are stored as embedded documents according to the *One-to-Many Relationships with Embedded Documents* model [Mon14, p. 141], because it is unlikely that the user is going to add values to that field. If the amount of data added to a field extends the reserved space for the document, the database has to reallocate the space, which leads to fragmentation and a slow write performance. Concluding the responses of the users are stored according to the *One-to-Many Relationships with Document References* model [Mon14, p. 143], since these fields are constantly extended.

The database is trying to keep the used storage low and manages to delete all events that have already passed. Besides that an event has several properties, e.g. a short description, a location, the date of the event and its last change.

3.3.5 Picture

Since the user is able to upload pictures that are relevant to the club, a document store is created to manage them. The picture's metadata is handled through that store, as well as the URL pointing to the file and an array with tags for the picture. Each picture is uploaded by a user, who can associate up to one division to the picture.

3.4 Wireframes and Mockups

During the design phase wireframes have been created to get an initial sense of how the application is going to look like. The findings from section 2.4.1 on page 14 and 2.4.2 on page 14 have been included into this early UI design. The wireframes were designed, trying to meet all requirements from section 3.1 on page 21.

3.4.1 Web application

The wireframes for the web application can be found in appendix B on page xiii. The top of the page is representing the navigation area. A conservative navigation approach was chosen, because of its simplicity and the small amount of negative aspects. A total of six elements are placed in the top bar which is more than the optimum of five elements but not quite the suggested maximum amount of seven items.

The first frame is showing the log in screen. The placeholder on top of the login fields could either hold the default logo, or a custom logo provided by the club's administrator. Every user is going to be able to access this page and enter his credentials. All upcoming pages can only get entered if the user has administration rights.

The second page is showing the dashboard of the application. This page gives statistics about the system and its users and might also inform the administrator about upcoming birthdays or anniversaries. From this page the administrator is also able to publish news, that later can be received by the user.

The next pages shows the user management page, where the administrator can view, alter, create and delete users. The left hand side lists all registered users and the right hand side presents all available information about the currently selected one. Within this form the administrator is able to subscribe the user to every available division. The user list on the left can be filtered with a search field on top of the list.

The fourth page is the division management page, which shows the arrangement of all created divisions on the left in a division tree. The tree can be altered in every possible way, as long as a tree-like structure is maintained. Every division can consequentially have an unlimited amount of subdivisions. The administrator is able to rearrange the divisions to match the club's organisational structure. On top of that he can view and alter the divisions on the right side after selecting the appropriate one.

The events management page is the fifth page presented in this section. On the left hand side a calendar is presented and after selecting a date all events that are scheduled for this date are presented underneath the calendar. When choosing one of the events their details are loaded into the right hand form, where they can be modified. Within this form the invited divisions are listed and can be added or removed.

The next page is the picture management page, that shows all images uploaded by the user. The pictures are filtered according to the division they were uploaded through. The user can refine the picture using the division tree on the left side.

The last wireframe is the settings page, where only the super admin can adjust settings for the system. The user is able to upload a custom club logo and change information like the club name. Second level administrator can only change their password on this page.

3.4.2 iOS application

The mockups for the *iOS* application can be found in appendix C on page xx. The mockup is displayed using a storyboard presentation, showing the connection between each view.

For the initial implementation the decision was made to use the conventional navigation concept of a tab bar located at the bottom of the window. This is owing the fact of an alternative concept being very time consuming, needing additional user tests to ensure its usefulness.

Starting at the bottom of the mockup page, the screen shown there is the event management tab. This view shows a calendar in the upper half of the screen and the list of events, scheduled for the currently selected date, in the lower half of the screen. Only events are shown, where the user has been invited to. Additionally another screen could be implemented showing more details of a specific event.

Continuing clockwise, the news feed is the next screen. Within this view each posted news, that is applicable for the user is shown. He is able to scroll through the posts and thereby inform himself about the club.

Next is the message screen, which is showing all currently subscribed divisions that have an enabled chat functionality. Within the mockup the conservative table view approach is shown, but the implementation of the alternative overview discussed in section 2.4.2.2 on page 16 is still considered. Out of this window the user can access the chat environment. It is based on the design of the stock *iMessage* app.

The upper right screen is the photos' section, where the user is able to shoot pictures within the app. After taking the picture he is free to tag it with any division he is part of and upload it to the *myVerein* server. Allready taken picture from the camera roll can be shared as well.

The last screen, which was not described so far, is the settings tab. In this view, the member can change his personal profile and change general settings that apply for his local device. This might include preferences about event reminder, in-app notifications and other behaviours.

3.5 Licensing

When creating any kind of intellectual property, that is supposed to be published, it is always necessary to think about the licensing of the work. Today an author or developer has the possibility to choose from a variety of available license tools, each with different purpose or restrictions. The main question on this topic is around the fact if the work should be part of the public domain, and therefore enable other people to use and share it freely or not.

It is a personal attitude to enable other people to learn and benefit from my work, if the circumstances allow it. Therefore I decided to license the documentation work done within this project using a Creative Commons - Attribution - Non Commercial - Share Alike 4.0 International license (CC BY-NC-SA 4.0). This enables user to retrieve, distribute and alter the work free of charge, as long as it is not used within a commercial environment, the appropriate attributes have been given and the altered work is published under the same license. [Com15]

The code written within this project is licensed using a GNU General Public License, version 2 (GNU GPL v2). This allows every person free of charge to use, modify and distribute the work, even in a commercial application, as long as the original source is included, the changed sections have been marked and the author received the appropriate attributions. On top of that every work that uses this system needs to use a license that is compatible with the GNU GPL v2. Last but not least, a disclaimer is included, excluding the developer from his liability for damage caused by the software. [Fou91]

In general releasing a software under an open-source license has three major benefits: First, the software itself can be reviewed for security or logical errors and every user can contribute fixing bugs or adding new content to the application. Additionally user trust open-sourced software more than proprietary ones, because of the possibility of code and security audits. This is a huge benefit when handling private data. Second, since the usage of the application is free, the adoption rate of the software could be higher compared to a non-open-sourced application. Last but not least the software can help other people to educate themselves and therefore support software developers to learn from the experience gained through this work.

Chapter 4

Implementation

As the last two chapters consists of throughly discussions about the realisation of an app for small and middle-size clubs I now will introduce the final implementation of these ideas. All findings presented above were examined critically and led to the conclusive result shown in the following section.

Because of technical difficulties and a limited amount of time, certain things discussed in 3 on page 21 had to be discarded or changed. Nevertheless most of the requirements could be realised. It is important to note that the time did not permit an extensive testing phase, therefore this software should be considered as an early beta product. There are several known issues at the time of this writeup, which can be found within *Github*'s bug-tracker located at `https://www.github.com/steilerDev/myVerein/issues`.

4.1 Visual branding

As pointed out in section 2.4.3 on page 17 the branding and moreover the recognisability of a brand is one of the most important parts when creating new system, that is going to be placed on the market.

4.1.1 Colour scheme

After considering the findings in section 2.4.3.1 on page 17 the author decided to use a *Tetradic* colour scheme. The base colour should be the most dominant one, where the other three should be used to draw the user's attention.

Respecting different colour tones as base colour, the decision was made to go for a soft green colour, more specifically the one defined through the hex code `0x13CD78`. It was chosen, because of it's calming property, which qualifies it for a base colour [Cha10a]. Green is described as "a

very down-to-earth colour. It can represent new beginnings and growth. It also signifies renewal and abundance" [Cha10a]. It is very balancing and harmonising, which relates it to wealth and stability.

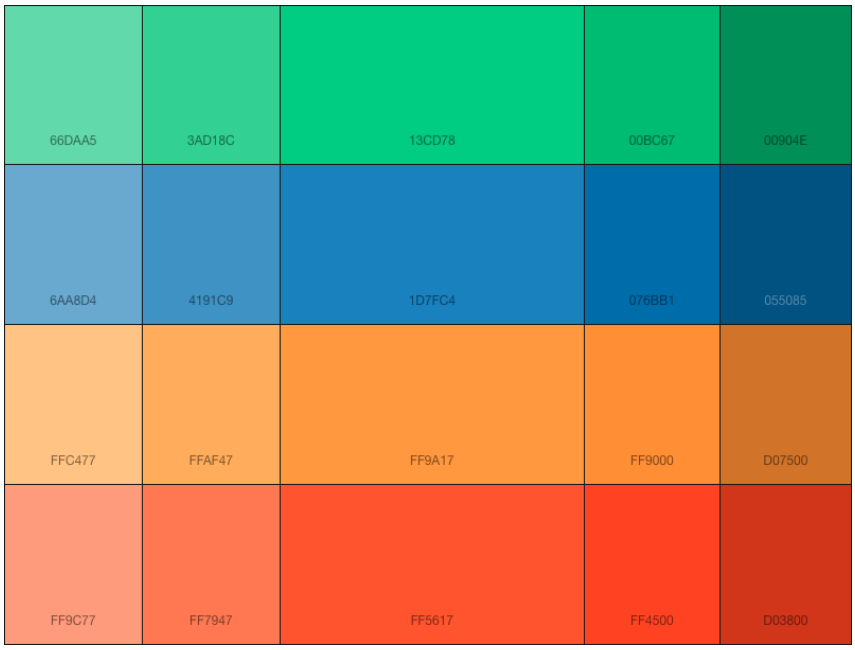


Figure 11: Colour scheme created using [15a]

Figure 11 shows the applied *Tetradic* method on the above mentioned base colour. The three side colours blue, orange and red fit well to this palette and are very suitable to draw attention in a green/white dominated environment, that is going to be created within the web and *iOS* app. The blue colours are going to be used for general information, where the orange tones are utilised for warning and the red ones for errors.

4.1.2 Logo

With our preferred colour scheme a logo, used among others as app icon, is going to be presented within this section. The concept shown in section 2.4.3.2 on page 18 was selected and enhanced.

Figure 12 on the following page introduces the final logo similar to the early one in figure 9 on page 19. The only main difference now is the adoption within our new system wide colour scheme. The logo is kept very simple and flat, while adding a 3D effect through the banner, making it more interesting. The dominant colour of the logo is the base colour of the brand's colour scheme, which is used to create a strong connection with the application itself.

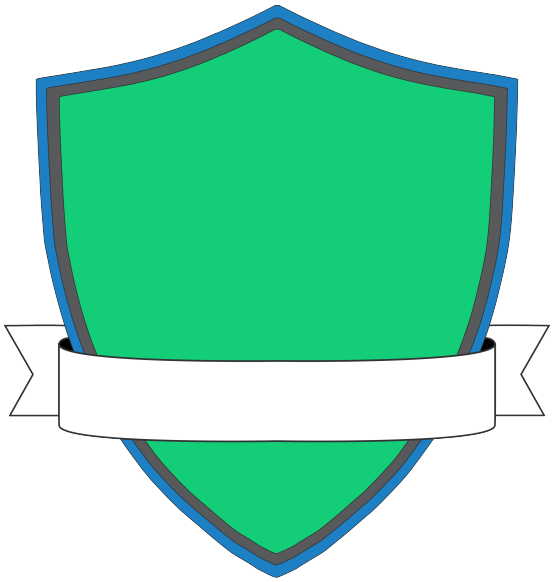


Figure 12: Final logo and app icon, own figure

In the end the simplicity and lack of writing as well as using a vector program on creation allows the logo to perfectly scale in any possible way. Figure 13 on the following page shows the final icon rendered using the different resolutions and context it is going to appear within the OS. Important to note is, that the icon itself is backgrounded by a gradient going from the green base colour on the upper left to the blue alternative colour of the scheme in the bottom right.

4.2 Back-end

The architecture of the system already presented in section 3.2 on page 24 has found its way into implementation. Therefore the back-end was created using the *Java* programming language and with the aspect of scalability in mind. For the final development of the server I will just highlight its most important impacts to the system not going into every detail.

4.2.1 Frameworks

In todays world a lot of open-source frameworks are available to simplify the creation of complex systems, by providing fundamental functionalities. To efficiently manage the growing amount of dependencies within any project, a dependency management framework is highly recommended. By using the state-of-the-art *Maven 3* framework, managing of third party libraries and their versions is highly simplified. Especially when working within a team, this management system is ensuring the usage of the same library versions across the team without the need of pushing

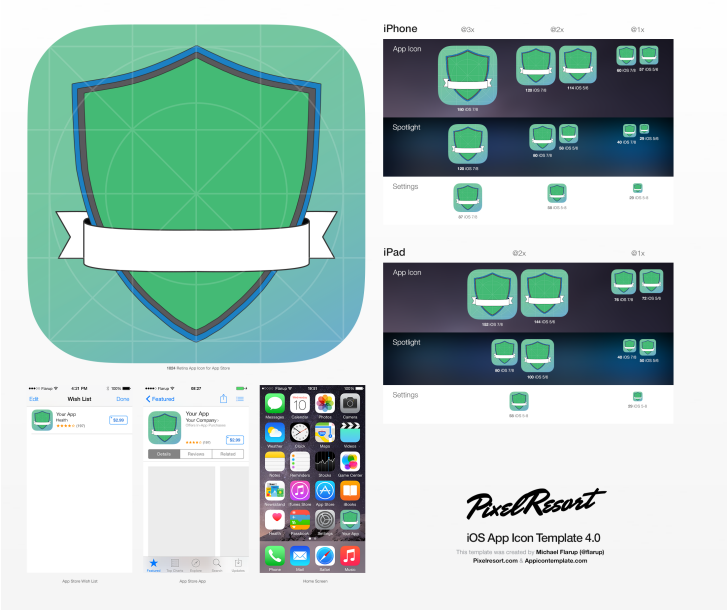


Figure 13: App icon within context and *iOS* guideline grind, created using *Pixel Resort*'s template

them into the version controlling system. It also keeps track of the currently used version and might help migrating to newer release. To simplify this project several well tested frameworks have been used.

The most dominant framework used is called *Spring*. This suite of frameworks is aiming on simplifying *Java* and *Java EE* application. Three main branches of *Spring* have been used:

1. *Spring WebMVC*: Enabling the fast development of Model-View-Controller (MVC) conform web applications
2. *Spring Data MongoDB*: Allowing easy and object oriented access of the database (see 4.2.3 on page 37)
3. *Spring Security*: Providing a concise interface for securing the platform (see 4.2.4 on page 40)

Another very important aspect during development is logging. Its overhead should be as small as possible, to reduce a performance impact on the application. Nevertheless a sufficient amount of log messages need to be written, to simplify the process of backtracking bugs and security breaches. To combine these two requirements a professional logging framework is necessary. One of the most important logging frameworks within the *Java* programming language is *Log4j2*. It provides an easy interface, is highly customisable and delivers a high performance. Additionally

the *SLF4J* bridging framework was used, enabling the inclusion of logging messages from 3rd party libraries like *Spring*.

The configuration of the logging framework is possible by manipulating the `log4j2.xml`, located within the class path. A manually added `log4j2.xml` always overwrites the default one provided by application. The configuration file for this project can be found in appendix D on page xxi. It has a third party log level of "Info" defined in line 4 and a log level of the *myVerein* application of "Trace" specified in line 22. Lines 43 and 44 define that the log is written to the `stdout` (in general this is the console) and into a rolling file called `myVerein.log`, located within the default *Tomcat* log directory. This log file can be analysed to find vulnerabilities of the application.

4.2.2 Scalability

As shown in section 3.2 on page 24, one qualification of the *myVerein* system should be the ability to run within a clustered environment. The load should be distributed on the servers using a load balancer. By enforcing this architecture the platform is able to dynamically scale, depending on the workload on the system. This is only possible if each instance of the server is headless retrieving all variable settings from a central data repository.

```
1 {
2   "clubName" : "myVerein",
3   "initialSetup" : false,
4   "customUserFields" : [
5     "Membership Number",
6     "Nickname"
7   ],
8   "rootDivision" : DBRef("division", ObjectId("553f92eb45661c558b00c0e9"))
9 }
```

Listing 1: Example layout of settings document

This behaviour is realised by putting down all settings within a dedicated document store on the *MongoDB* data provider. Listing 1 shows an example document within the settings store. The available options are currently very limited and only provide the possibility to set a club name, define the root division and add custom fields to every user profile. Note that there is only one settings document allowed within its document store, since the systems cannot know which of them to choose. As soon as an instance encounters more than one configuration, all of them except the first one is removed and a warning is issued.

Of course a stand-alone instance has to get told how to access the database. One way to implement this information would be to hard-code the access details to the data provider within the source code, but that would lead to a low customisability. Therefore every server gets a local `myVerein.properties` profile within their resources folder, which is loaded during start up. This files holds the information about the hostname, port, username, password and database name of the *MongoDB*. Concluding the only thing that needs to be modified when deploying a new unit to a cluster is this file, all remaining information are consequentially retrieved from the database.

An initial setup is only possible, if there is no document within the settings document store of the database, specified by the above mentioned properties file, or if this document's `initialSetup` flag is set to true. If either of the previous properties holds true, the access of the index page will present a page, leading through the initial setup. After specifying all required initial information, the provided database is going to remove all data from previous installations and the system needs to be restarted manually. This method is used when trying to reset the system through the settings page of the administration panel.

4.2.3 Database

In the first part of this section the final layout of the database is presented while the second part is showing the object-oriented mapping provided by the *Spring Data* framework.

4.2.3.1 Final layout

In section 3.3 on page 26 the initial layout of the data provider was introduced and explained. Even though the planning was well-conceived, some functionalities had to be changed, because of unforeseen technical difficulties or changes within the feature set. The conclusive ER diagram can be found in appendix E on page xxii. The differences between the final layout and the one presented in section 3.3 on page 26 are going to be discussed in this section.

User Within the user object only small design changes have been made. The amount of default entries within the document has been increased, e.g. every profile contains the fields "Member since", "Passive since", "Resignation date" and "Birthday". This changes have been performed to be able to create a powerful dashboard for the administrator. On top of that the dynamic "Private Information" and "Public Information" have been merged into a single array called *Custom user fields*, since a rich public profile is not going to be provided within this version. Besides that, all connections between the user and other documents, that are stored remain as shown in the design section.

Division The only change made to the division object, is the addtional member list. This element contains the IDs of all members that are part of the division. The maintenance of this list is very expensive, since it needs to be updated for every division change of a user. Nevertheless its implication is a significant speedup of the message sending process amongst others, due to the fact that this member list is needed during each send request of a message. Since this process is happening more often than a change in the division's member list, the caching of this piece of information is implemented.

Message Besides the initial idea to create a self-destroying message stack, the decision was made to store all messages permanently. This is mainly due to the fact to later be able to support the access of one account on multiple devices. On top of that this will enable the possibility of storing meta-information about the delivery status of the message. Every receiver of a message is stored within the "Receiver" map, where the key is the user ID string and the value is the delivery status. Therefore the retrieval of all non-delivered messages of a certain user is very fast and easy.

Event The event document's properties were expanded, similar to the user's. Additional information like the precise coordinates of a location (stored in "Location Lat" and "Location Lng") are now part of the object. Above that the pattern of storing the message status together with the user ID was adopted within the "Invited user" relationship, where the response is the value to the invited user's ID.

Settings As explained in section 4.2.2 on page 36 the system needs a central settings document. During the design phase this property was not considered and so missing. Within the final program it is part of the database design. It has a relationship to exact one division, which is the root division. This root division's name needs to be equal to the club name by design.

Remember-me token This document store wasn't considered either during the initial design. It is an independent document with no direct relationship to any other document and used to keep a user logged, even if his session expires. A detailed explanation is given in section 4.2.4 on page 40.

Picture Due to limited time resources the feature of uploading and inspecting photos needed to be rescheduled for a future release. Therefore the document store related to this feature is no longer part of the final database design, but might be added to the system at a later point in time.

4.2.3.2 Spring Data

To access the data from the *Java* program a developer gets several options. He either can use the official *MongoDB* driver library provided by *MongoDB*, or a state of the art object-oriented mapping library like *Hibernate*. Since the *Spring WebMVC* framework is used within the project, another part of the *Spring* framework was chosen to provide access to the database: *Spring Data MongoDB*.

This framework is build on top of *Hibernate* and provides the same mapping capabilities from *MongoDB* documents to objects within *Java*, adding *JavaEE* functionalities like annotations. Each document stored in the database should be represented by a class of the project. *Spring Data* would also been able to return a map containing the entry, but this solution is not concise at all and was not used within this project. Every document described in the previous chapter is therefore mapped to a *Java* class located within the `de.steilerdev.myVerein.server.model` package.

To enable the usage of an object within the context of *Spring Data* it needs to inherit the `BaseEntity` class. Afterwards every property of the object is mapped to the appropriate attribute of the document, unless it was annotated with `@Transient`, which is disabling the storage of the respective property in the database.

Additionally validator annotations are available, ensuring that a specific property is `@NotEmpty`, `@NotNull`, or an `@Email`. If these requirements are not met during the save process, an exception is thrown.

Within *MongoDB* the relation between two documents can be expressed by storing them either as a nested document or using a reference to the document. Both techniques have their pros and cons and can be used within *Spring Data*. When specifying the object as a member field within the stored class, this results into a nested document. In opposite to that the annotation `@DBRef` specifies a relation between two independent documents.

It is important to note that the references to other documents are automatically resolved, which simplifies the retrieval of nested documents. However this process is not done lazily, which resulted into a bad query performance for highly nested documents. *Spring Data* therefore introduced the (optional) `lazy` property of the `@DBRef` annotation, indicating that this property is loaded upon access. Unfortunately this feature is not working in every context and could not be used for every reference.

The proccess of retrieving and storing the above mentioned classes is done through repositories. One repository for each object class that needs to be stored within the database is created. These repositories are defined as interfaces and by analysing the function name or reading the provided annotation, *Spring Data* is able to create the appropriate function and query to retrieve the

objects. The repositories also provide a `save()` function durably storing a specified object or a list of objects if all validation requirements are met.

4.2.4 Security

To create a sophisticated and secure policy for the access of the resources provided by *myVerein* the *Spring Security* framework was used. This framework is providing a concise configuration interface based on user roles.

The system introduces three different roles: `ROLE_USER`, `ROLE_ADMIN` and a non-authenticated guest user. Appendix F on page xxiii shows the configuration used within this system. Lines 11 and 12 refer to the publicly available content, including static content (JS and CSS files within `/resources`) and dynamic content (Club logo within `/content`). They are both specified to use no security checking at all which improves the performance when accessing these resources.

Fortunately the data provider of the framework is very flexible and therefore the *MongoDB* could be used to provide the necessary information about the users. Line 82 - 108 specify this configuration. A plain text fallback solution is also provided in case the database is not available, but the system needs to be accessed anyway or the administrator's password got lost. By default the `myVereinUser.properties` file within the resources folder is empty, but can be modified by the server administrator to enable this feature.

Line 44 to 64 use the default behaviour of *Spring Security* to enforce the specific policies within the web app environment. In case a user is not authenticated, the server response would be a `302` HTTP status code, meaning "moved temporarily". A standard web browser redirects the user to a specified page, which is in this case the login page defined in line 51. The policies for this range are defined within line 46 and 49: Besides the login, logout and error page, only a user with the role `ROLE_ADMIN` can access the resources of the web app. It is important to note that the configuration will use the first matched rule, starting at the top, when trying to find the applicable policy.

Even though the behaviour presented in the last paragraph is very user friendly, it is not compatible with the REST standard. A RESTful API needs to return appropriate status codes according to the availability of a resources, this includes a `401` status codes for an unauthenticated user. Therefore custom handler needed to be created and applied to the Unified Resource Identifier (URI) that define the REST interface. These overwriting handlers can be found within the `de.steilerdev.myVerein.server.security.rest` package. Their definition and usage within the configuration file can be found within lines 35 - 42, 15, 22 and 23. The remaining notable entries within the REST block (lines 14 to 42) are the permission for every user to access the login, logout and initial setup URI of the REST API and the distinction between

the user's area (`/api/user`) and administrator's area (`/api/admin`). For security reasons every other resource within the `/api` URI is matched by a rule restricting the access to administrators. Therefore new functionalities, that did not get an own policy yet are secured.

As mentioned in section 4.2.3.1 on page 37 the back-end supports a remember-me functionality, keeping a user logged in, even if his session expires. Besides the regular session token, which gets deleted rather quick, a second "remember-me" cookie is set, if the user wants to stay logged in. Within this cookie the user's name, a random token and a series number is handed over and stored within the database. After the session expires, the user presents this cookie containing the username, series and token. If these information match the ones currently stored, he gets authenticated. Upon a successful authentication a new token is issued and overwrites the one currently present in the database. The series number stays the same. In case a "remember-me" cookie is presented, matching the series and username, but the token is different, a cookie theft is assumed and all cookies associated with the user are invalidated. Therefore a thief would only be able to use a stolen cookie until the user logs in the next time. Additionally the system is able to notify the user of the detection of a possible cookie theft. [Jas06]

The configuration for this "remember-me" service can be found from line 66 to 80 within the configuration from appendix F on page xxiii. The required data provider is shown within the class `de.steilerdev.myVerein.server.security.RememberMeTokenDataProvider`.

4.2.5 Business logic

A core aspect of a system is the implemented logic processing the provided and requested data. Within this section three examples will be presented. The only way to fully understand the complete code base, is by reviewing the source code and *JavaDoc*s of the back-end system.

4.2.5.1 Replace user division

Every time a user's division changes, the data needs to be updated in several places. At first the subscribed divisions of the user's object will be changed, then the list of invited users of an event needs to be updated. This is due to the fact that the user might be either no longer or newly invited to certain events. Finally the user list, stored together with the divisions, needs to be modified accordingly. Since this is obviously a very expensive procedure it needs to be implemented as efficiently as possible. Fortunately a user does not change divisions too often which compensates this complexity.

Appendix G on page xxv shows the relevant code snippet performing most of the logic of this use case. The code was taken from the `User` class located within the package `de.steilerdev.myVerein.server.model.user`. The method `replaceDivisions` takes three arguments, the

`UserRepository` and `EventRepository` used to interact with the database as well as a list of `Division` object. This list contains the new divisions for the user. The initial list of division is still held by the `User` object, where this method is executed on.

The method can be separated into four different branches. The first one is the most trivial one, presented in line 526 to 529. In this case the current as well as the new list of divisions is empty. Therefore no changes have to be made to the user, to any event or division.

The next branch from line 530 to 542 covers the case, where the new list of divisions is empty and the old one is not and therefore the user only needs to unsubscribe from all old divisions and get removed from all events. The user is removed from all of his divisions in line 533 and 536. Each event, that invited the old divisions is found and updated in parallel within line 537 to 541. The parallelism should speedup this very expensive operation.

The third branch is very similar to the last one and shown from line 543 to 555. In this case the user was not part of any division before this update and therefore only the new divisions and their events are going to be updated. The user is added to the divisions in line 546 and 547 and the events are updated in parallel within the lines 550 to 554.

The last branch is the most complex one. It covers the case where the user has been part of divisions before the change and will be part of divisions afterwards. In this case the first step is to perform set operations on the two lists. By creating the intersection in line 559, the divisions that are in both sets are determined. These ones do not need to be changed at all, since the user was part of the divisions before and will be after. From line 564 to 578 this intersection is used while iterating over the lists of new and old divisions to determine whether or not the person needs to be added or removed from the divisions. Each changed division is put into a list, specifying the need of updating the division's events. Since the collection is accessed from different threads at once, a synchronised list is needed, or otherwise an unexpected behaviour could occur.

After all divisions and events have changed, the altered user can be stored durably and no inconsistency should be present.

4.2.5.2 Getting events over period

Another very complex task is getting all events that occur over a defined period. This functionality is especially needed within the web interface, where the user specifies a date and expects all relevant events to show up. They include not only the ones taking place between the defined period but also the ones spanning completely or partially over this time frame.

Initially this task was realised using a lot of back-end computing, iterating over all events within a given timeframe. This left two down sides: On the one hand the request took a fair amount of time to finish and very long spanning events were ignored due to the fact that only the

ones within a timeframe (+/- 1 month) around the specified period were gathered and checked for their relevance. With commit `20f8155`, created on the 26th of April, this behaviour improved significantly.

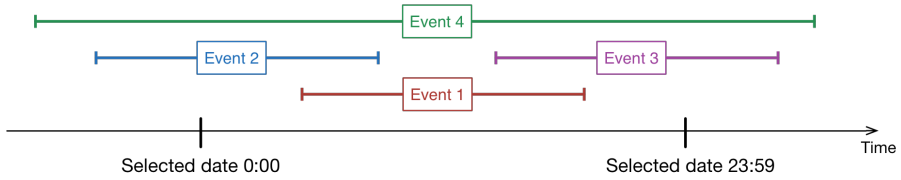


Figure 14: Possibilities of events occupying a period, own figure

This change in behaviour could be realised, by updating *Spring Data MongoDB* to the latest release, which included the support of comparing *Java 8* `LocalDateTime` objects within a query. By shifting the workload from the application to the database the process benefits from internal query optimiser, a reduced result set and an overall improved responsiveness.

```
{$or: [
  {$and: [
    {'startDateTime': {$gte: ?0}},
    {'startDateTime': {$lte: ?1}},
    {'endDateTime': {$gte: ?0}},
    {'endDateTime': {$lte: ?1}}
  ]},
  {$and: [
    {'startDateTime': {$lt: ?0}},
    {'endDateTime': {$gt: ?0}}
  ]},
  {$and: [
    {'startDateTime': {$lt: ?1}}
    {'endDateTime': {$gt: ?1}}
  ]}
]}
```

Listing 2: MongoDB query for getting events over a period

Figure 14 shows the four possibilities any event can occupy just a fraction or the complete given period. Either it is within (event 1), starts before and ends within (event 2), ends after and starts within (event 3), or starts before and ends after the period (event 4). When analysing the

problem, event 4 can actually be removed since it can be covered by a combination of event 2 and event 3 as shown below. This leads to a total of 3 relatively simple conditions that can be easily implemented as a query for *MongoDB*.

Listing 2 on the previous page is showing the query used to gather all events spanning over a period defined by a start and end time. Within the query, the placeholder `?0` and `?1` are replaced by the start, respectively end of the period during runtime. The whole query is an "OR"-connection of three different statements.

The first statement from line 2 to 7 covers event 1 from figure 14 on the preceding page. Line 3 and 4 check if the start of the event is between the start and end of the period, where line 5 and 6 assure the end of the event satisfies this condition as well.

The second block from line 8 to 11 covers event 2. By only checking if the event is spanning over the start of the period, it is possible to also match event 4, reducing the amount of cases by 25%. Finally the block from line 12 to 15 matches event 3 as well as event 4, by checking if the event is spanning over the end of the period.

This query is used within several functions in classes related to the event management. The specification of the query can be found in the `EventRepository` class located in the `de.steilerdev.myVerein.server.model.event` package. It is defined as a custom query, using the `@Query` annotation for the `findBySpanningOverPeriod` function. This function accepts two *Java 8* `LocalDateTime` objects stating the period.

4.2.5.3 User role assessment

During log-in it is important to evaluate the role of an authenticated user to be able to judge what policies apply to him. As noted in section 4.2.4 on page 40 the security back-end uses the *MongoDB* as a data provider for authentication. This is realised through the implementation of the `org.springframework.security.core.userdetails.UserDetails` interface within the user object. Besides authentication properties, like the password, username and information about the validity of the account, this interface expects the object to provide a list of granted authorities. These authorities match the user roles presented in section 4.2.4 on page 40.

Upon each authentication request, the user gets loaded by the `UserAuthenticationService` located in the `de.steilerdev.myVerein.server.security` package and gets his authorities assigned. As long as the user's session is active these authorities are cached within the system, to avoid reloading all the information. Listing 3 on the next page shows the function evaluating and returning the list of authorities. It was taken from the `UserAuthenticationService` class mentioned above.

After the authentication was successful this function is called, automatically assigning the user to the ROLE_USER group in line 10. Afterwards all divisions, administrated by the selected user, are retrieved. If this list is not empty, which means that this user is administrating at least one division, he becomes part of the ROLE_ADMIN authority. If one of the divisions within this list is the root division he get the role ROLE_SUPERADMIN and can therefore manipulate all settings within the system. Since the root division does not have a parent, the associated member field is set to NULL. Using this assumption the list is checked for containing the root division within line 15.

```
1  /**
2    * Checks and returns the user authorities. The authorities are assigned
       according to the {@link de.steilerdev.myVerein.server.security.
       UserAuthenticationService.AuthorityRoles AuthorityRoles} enum.
3    * @param user The user, whose authorities need to be checked.
4    * @return A list of the roles of the user.
5    */
6  private ArrayList<GrantedAuthority> getUserAuthorities(User user)
7  {
8    logger.trace("[{}] Checking user's granted authorities", user);
9    ArrayList<GrantedAuthority> authorities = new ArrayList<>();
10   authorities.add(new SimpleGrantedAuthority(AuthorityRoles.USER));
11   List<Division> administratedDiv = divisionRepository.findByAdminUser(user);
12   if(!administratedDiv.isEmpty())
13   {
14     authorities.add(new SimpleGrantedAuthority(AuthorityRoles.ADMIN));
15     if(administratedDiv.stream().anyMatch(div -> div.getParent() == null))
16     {
17       authorities.add(new SimpleGrantedAuthority(AuthorityRoles.SUPERADMIN));
18     }
19   }
20   logger.info("[{}] Retrieved user authorities: {}", user, authorities);
21   return authorities;
22 }
```

Listing 3: User role assessment executed upon authentication (slightly modified for readability)

Finally the list of granted authorities is returned to *Spring Security* which can proceed with the log in of the user. Additionally it is now able to evaluate if the user is allowed to access the specified resource.

4.2.6 REST API

Within this section, the available REST API is presented. Every available URI of the back-end is shown. The APIs are grouped according to their purpose. The tables are going to give information about the functionality that can be expected. Besides that the policy protecting this resource is stated as well. These API always return a JavaScript Object Notation (JSON) formatted response, if there is data expected to be returned, otherwise just the appropriate HTTP status code is present. For more details on these functionalities, please look at the source code and the *JavaDoc*.

4.2.6.1 Authentication

Table 1 shows the interface that needs to be used to authenticate against this platform. After the authentication was successful the system will send a session cookie that needs to be used on every successive request.

Table 1: Authentication API

URI	**Request method**	**Parameters**	**Description**
`/api/login`	`POST`	`username`, `password`, `rememberMe` (optional)	If the `username` and `password` match an entry within the database, a success status code is returned. Additionally details about the system are present in the header of the response. These include the `System-ID`, the `System-Version` and the user's `User-ID`. If the `rememberMe` parameter is present, the user will receive a remember-me token, discussed in section 4.2.4 on page 40. If the authentication was not successful a `401` status code is returned.

Continued on next page

Table 1 – continued from previous page

URI	Request method	Parameters	Description
`/api/logout`	`POST`		Using this API will invalidate the session and remember-me cookie and therefore log the user out of the system.

4.2.6.2 User

This section will cover all functionalities provided by the REST API concerning user retrieval or manipulation.

Member facing Table 2 will focus on all member facing functionalities of the REST API, within the context of this section. They are implemented in the `UserController` class, which can be found in the `de.steilerdev.myVerein.server.controller.user` package. These functionalities can only be accessed by a user if he is authenticated and part of the `ROLE_USER` group.

Table 2: Member facing user API

URI	Request method	Parameters	Description
`/api/user/user`	`GET`	`id`	Returns the complete profile of the user identified by his user ID. The fields of the returned profile are populated according to the security settings of the user.
`/api/user/user`	`POST`	`deviceToken`	This function updates the device token associated with the currently logged in user. This functionality is needed as soon as a notification service for the *iOS* app is established. .

Administrator facing Table 3 is showing all administrator related API, within the context of this section. The implementation can be found in the `UserManagementController` class, located inside the `de.steilerdev.myVerein.server.controller.admin` package. Since they are only performing administrator related tasks, the user needs to be part of the `ROLE_ADMIN` authority.

Table 3: Admin facing user API

URI	Request method	Parameters	Description
`/api/admin/user`	`POST`	`email`, `firstName`, `lastName`, `birthday`, `password`, `gender` (optional), `street`, `streetNumber`, `zip`, `city`, `country`, `activeMemberSince`, `passiveMemberSince`, `resignationDate`, `divisions`, `IBAN`, `BIC`, `userFlag`, `cuf_*`	Saves/creates a user. The `userFlag` needs to state the ID of the user who needs to be changed or `true` if a new user is created. The `password` field is only respected when creating a new user, concluding you can't change the password of a user using this API. All other parameter specification can be found within the corresponding *JavaDoc*.
`/api/admin/user`	`GET`	`term` (optional)	Returns a list of all user. If the `term` parameter is present, the list is filtered and only users, where the first name, last name or email contains `term` are returned.
`/api/admin/user`	`DELETE`	`email`	Deletes the user identified by his `email`.

Continued on next page

Table 3 – continued from previous page

URI	Request method	Parameters	Description
`/api/admin/user`	`GET`	`email`	Returns the user identified by `email`. If the user is not allowed to modify his profile, all private information are hidden and a parameter is set including a message telling the application that the selected user cannot be manipulated.

4.2.6.3 Division

The following tables are focused on the ability to create, modify, retrieve and delete divisions through the REST API.

Member facing Table 4 shows all division related tasks, that are available for a regular member. The `DivisionController` class, providing the presented functionalities can be found in the `de.steilerdev.myVerein.server.controller.user` package. To access the resources shown in the table the user needs to be part of the `ROLE_USER` authority.

Table 4: Member facing division API

URI	Request method	Parameters	Description
`/api/user/division`	`GET`	`id`	This function returns all available information about a division identified by its ID.
`/api/user/division`	`GET`		This function returns a list containing all division ID's, the user is part of.

Administrator facing For modifying, creating and deleting divisions a user needs to be part of the `ROLE_ADMIN` authority. Table 5 on the next page is presenting the API needed to perform these

tasks. The presented functionality is implemented within the `DivisionManagementController` within the `de.steilerdev.myVerein.server.controller.admin` package.

Table 5: Admin facing division API

URI	**Request method**	**Parameters**	**Description**
`/api/admin/division`	`POST`	`name`, `description`, `admin`, `ID`	Saves the division defined by its `ID`, using the provided parameters. The division needs to exist.
`/api/admin/division`	`POST`	`new`	Creates a new division, located directly underneath one of the user's administrated divisions. The request returns a JSON map, containing a success message and the name of the new division.
`/api/admin/division`	`DELETE`	`ID`	Deletes the division defined by its `ID`. The user needs to administrate the division or a parent division of the stated division.
`/api/admin/division`	`GET`	`term` (optional)	Returns a list of all divisions. If the `term` parameter is present, the list is filtered and only divisions, whose division name contains `term` are returned.
`/api/admin/division`	`GET`	`ID`	Returns the division defined by its `ID`, including its administrator, reduced to his email and name.

Continued on next page

Table 5 – continued from previous page

URI	Request method	Parameters	Description
`/api/admin/division/divisionTree`	`POST`	`moved_node`, `target_node`, `position`, `previous_parent`	Moves the division defined by `moved_node` from `previous_parent` to `target_node`, while `position` defines the relation between the division and its new parent.
`/api/admin/division/divisionTree`	`GET`		Returns the division tree administrated by the current user, represented by a nested JSON object.

4.2.6.4 Event

The API that can be used to retrieve and manipulate events is going to be presented within this section.

Member facing The member facing part of the event API is shown in table 6. To access events, the user needs to be authenticated as `ROLE_USER`. The API is provided by the `EventController` class within the `de.steilerdev.myVerein.server.controller.user` package.

Table 6: Member facing event API

URI	Request method	Parameters	Description
`/api/user/event`	`POST`	`ID`, `response`	Saves the user's response to the event specified by the `ID`. The `response` needs to be a string representation of a value from the enumeration defined in `de.steilerdev.myVerein.server.model.event.EventStatus`.

Continued on next page

Table 6 – continued from previous page

URI	Request method	Parameters	Description
`/api/user/event`	`GET`	`lastChanged` (optional)	Returns the IDs of all events the user is invited to. If the `lastChanged` parameter is set, only events that changed after this date are returned. The `lastChanged` parameter needs to be formatted according to the pattern `YYYY-MM-DDTHH:mm:ss`.
`/api/user/event`	`GET`	`ID`	Returns the event specified by the `ID`, containing all available information.
`/api/user/event`	`GET`	`ID`, `response`	Returns a list of user IDs that gave the response specified by `response` for the event specified by `ID`. The `response` needs to be a string representation of a value from the enumeration defined in `de.steilerdev.myVerein.server.model.event.EventStatus`.

Administrator facing The management of the events is done through the administrator facing API shown in table 7 on the next page. The usage of this interface requires the user to be authenticated as `ROLE_ADMIN`. The functionality presented here is implemented within the `EventManagementController` located inside the `de.steilerdev.myVerein.server.controller.admin` package.

Table 7: Admin facing event API

URI	**Request method**	**Parameters**	**Description**
`/api/admin/event/month`	`GET`	`year`, `month`	Returns a list of dates within the specified month. Each date is converted to a String using the `toString()` method of `LocalDateTime`. Each date of the list indicates, that an event is scheduled on the specific date. This function considers all events stored in the system and not only the ones that are user specific.
`/api/admin/event/date`	`GET`	`date`	Returns a list of events, scheduled for the `date`. The `date` parameter needs to be formatted according to the pattern `YYYY/MM/DD`.
`/api/admin/event`	`GET`	`ID`	Returns an event identified by its `ID`. If the user is not allowed to change the event, an administration-not-allowed parameter is present.
`/api/admin/event`	`DELETE`	`ID`	Deletes the event specified by the `ID`. An event can only be deleted, if the user either created it, or the user is the super admin.

Continued on next page

Table 7 – continued from previous page

URI	Request method	Parameters	Description
`/api/admin/event`	`POST`	`eventFlag`, `eventName`, `eventDescription`, `startDate`, `startTime`, `endDate`, `endTime`, `location`, `locationLat`, `locationLng`, `invitedDivisions`	This function updates the event identified by its ID within the `eventFlag` field, or creates a new event, if `eventFlag` is set to true. All other parameter specification can be found within the corresponding *JavaDoc*. An event can only be modified, if the user either created it, or the user is the super admin. If the event was saved successfully a JSON map is returned containing a success message and the ID of the event.

4.2.6.5 Message

The messages API does not have any administrator facing functionalities, therefore only member facing ones are available. Concluding to use the messaging service of the system, the authenticated user needs to be part of the `ROLE_USER` authority. Table 8 shows all provided functionalities. The implementation of the message handling can be found in the class `MessageController` within the package `de.steilerdev.myVerein.controller.user`.

Table 8: Message API

URI	Request method	Parameters	Description
`/api/user/message`	`GET`	`ID`	Returns the complete message specified by `ID`. If the message is delivered the first time, its message status is going to be set to delivered.

Continued on next page

Table 8 – continued from previous page

URI	Request method	Parameters	Description
`/api/user/message`	`GET`	`all` (optional)	Returns a list with the IDs of all unread messages of the currently logged in user. If the `all` parameter is set, the list will contain all messages ever send to the user.
`/api/user/message`	`POST`	`division`, `content`, `timestamp`	Sends a message with a specified `content` to the server. The receiving `division`, specified by its ID, needs to exist and the user needs to be part of it. If the send was successful the response contains the send message, with the message's system generated ID as well as the timestamp assigned by the server.

4.2.6.6 Initial setup

Since there is no user specified during the initial setup, this resource is, in theory, accessible by every user, even if he is unauthorised. Nevertheless, the system will only accept requests using this API if either the database is unavailable, or the initial setup flag is set. The functionality presented in table 9 on the next page is implemented by the `InitController` class, located within the `de.steilerdev.myVerein.server.controller.init` package.

Table 9: Initial setup API

URI	Request method	Parameters	Description
`/api/init`	`POST`	`clubName`, `firstName`, `lastName`, `email`, `password`, `passwordRe`, `createExample` (optional)	If this call is valid, the function clears the database specified within the local `myVerein.properties` file, creates a new user specified by `firstName`, `lastName` and `email` securing the account with the `password`, as well as a root division with the name `clubName`. The newly created user is becoming administrator of the root division and therefore super admin. If the `createExample` parameter is set to "on", the system is going to get populated with example users, divisions, events and messages.

4.2.6.7 Settings

This section is presenting the API that can be used to alter the system settings. To use this interface shown in table 10 on the following page the user needs to be authenticated as `ROLE_ADMIN`. Unfortunately a regular admin only has a limited set of options to alter the settings, to use the full set he needs to be `ROLE_SUPERADMIN`. The presented functionality is provided by the `SettingsController` class within the `de.steilerdev.myVerein.server.controller.admin` package.

Table 10: Settings API

URI	Request method	Parameters	Description
`/api/admin/settings`	`POST`	`currentAdmin`, `adminPasswordNew` (optional), `adminPasswordNewRe`, `clubName` (optional), `clubLogo` (optional), `cuf_*` (optional), `currentPassword`	If the `currentPassword` is correct and the user is the super admin, all parameters from this request are saved. The `clubLogo` needs to be a multipart image file, either a `JPG` or `PNG`. The parameters prefixed with `cuf_` are the custom user fields that are going to be available in all user profiles. The `currentAdmin` parameters defines the new super admin, unless the current user is only a second-level administrator. In this case the field needs to match the user's email. A regular administrator can only change his password through this API..
`/api/admin/settings`	`DELETE`	`password`	This functionality can only be used by the super admin. If the password is correct, this function will set the `initialSetup` flag within the settings and therefore enable a new initial setup, leading to the deletion of all information within the system.

Continued on next page

Table 10 – continued from previous page

URI	Request method	Parameters	Description
`/api/admin/settings`	`GET`		This function returns all available settings, unless the user is not the super admin, then the response will only contain his email, name and ID together with an administration-not-allowed message..

4.2.6.8 Content

This system takes advantage of *MonogDB's GridFS*, which allows a user to efficiently store big binary data within the database. The content API presented in table 11 allows access to files stored for the *myVerein* system. The files are currently publicly available and not secured in any way. This interface is provided by `ContentController`, located within the `de.steilerdev.myVerein.server.controller` package.

Table 11: Content API

URI	Request method	Parameters	Description
`/content/clubLogo`	`GET`	`defaultLogo` (optional)	This resource returns the current logo of the system. If the `defaultLogo` parameter is present or there is no custom logo the default *myVerein* logo, shown in 4.1.2 on page 33 is going to be served..
`/content/clubLogo`	`DELETE`		This interface can only be used if the user is authenticated as super admin. It will delete the currently stored custom club logo and show the default one.

Continued on next page

Table 11 – continued from previous page

URI	Request method	Parameters	Description
`/content/clubName`	`GET`		Returns the current name of the club.

4.3 Front-end

As discussed in section 3.2 on page 24, there were two front-end application developed for this system. The first one is the administrator facing web interface, enabling the management of the club, the other one is the member facing *iOS* app, containing messaging and a personalise club schedule. Unfortunately not all features requested in section 3.1 on page 21 were realised, nevertheless all of the obligatory requirements and most of the optional requirements were implemented.

4.3.1 Web-Application

Strictly speaking the web application is part of the back-end server, since it's files are hosted there and the resources are protected using *Spring Security*. The user is able to browse to the root of the server's Unified Resource Locator (URL) and will be presented with the login form of the interface. By providing his credentials, the user is able to access the secured area, as long as he is part of the `ROLE_ADMIN` authority.

The presented interface is designed to operate without any page reload, and therefore behaves like a desktop application within the browser. This is done by using the AJAX principles and taking full advantages of JS and several libraries for efficient visual manipulation of the content. Every requested content is asynchronously loaded and presented, so we can always ensure the requested data to be up-to-date.

The performance is improved by using aggressive caching techniques and reducing the loading time by aggregating all required JS and CSS resources within a single, minified file. By combining all of these techniques, the portal should behave and feel as responsive as possible and therefore increase the usability tremendously.

During the development the "mobile first" approach of the *Bootstrap* library was enforced and a mobile view is available. It is yet to be optimised and therefore not completely tuned for production use. Nevertheless the basic work has been done and therefore the workload of improving the interface shouldn't be too high.

4.3.1.1 Template Engine

To manipulate a web page on the server side within the *Java* environment, a template engine needs to be used. A developer can choose between the default Java Sever Pages (JSP) engine or pick one of the many 3rd party libraries. At the beginning of the development process the possibilities have been evaluated and the author of this work chose to use the *Thymeleaf* engine.

This library was specifically designed to remove the problem of not being able to accurate display the JSP templates within a browser, without needing an application to process it first. This is due to the fact, that JSP adds tags, which are not compatible with the Hypertext Markup Language (HTML) syntax. This leads to the inability of a correct rendering of the webpage. In contrast to that *Thymeleaf* introduces additional attributes within the valid HTML tags, which are ignored by a browser. Therefore the design of a dynamic web page can be inspected, without the need of a functional server.

Within this project the template engine was mainly used to enable the localisation of the webpages. Unfortunately the time did not permit a full execution of this process, but the `init.html` webpage is implementing the skeleton needed to easily translate the content. For example every tag containing text has the `th:text` attribute added, which is telling *Thymeleaf* to replace the text within the tag with the one specified by this attribute. The value of `th:text` is pointing to a resource bundle containing the translations. The correct resource bundle, if available, is automatically chosen by *Spring Web* depending on the language requested by the browser.

The other benefit of *Thymeleaf* is the ability to have a runtime combination of multiple template files. This enables the developer to separate functionalities, even though they are later delivered to the user within a single file. This improves the structure of the project, as well as the readability.

Finally the produced HTML is very well structured. *Thymeleaf* enforces the uses of *XHTML* and gives a clean output. In opposite to that, JSP creates an unvalidated HTML containing a lot of white spaces, unnecessarily blowing up the size of the request.

4.3.1.2 UI

During the development of the UI, the presented wireframes in section 3.4 on page 28 were respected. As shown in appendix H on page xxvii the wireframes and the actual implementation are very similar.

The main difference are the missing views. Unfortunately the time did not permit to implement the photo library nor the dashboard. Because of that, it is not possible to present these views.

The web page itself is endorsing the colour scheme and branding introduced in section 4.1 on page 32. The dominant colour is the green base colour. In several contexts the tone was slightly modified to meet the requirements, e.g. the month view in the calendar header is darker than the weekday view, to separate the both. The complementary colours, not visible on the screenshots, are used within notifications, signalling errors or the need of user actions.

4.3.2 iOS Application

The more important front-end application is the *iOS* app, since it is going to be used much more often than the web interface. The application was written in *Apple*'s latest programming language Swift, providing a concise language interface, which is secure and of high performance. The application is not released to the *App Store* yet, because the whole system is still in a beta stadium and therefore not ready for a release.

Comparable to the web application above, not all of the features discussed in section 3.1 on page 21 could be realised, due to the missing amount of time. Nevertheless a lot of the principles discussed earlier could be implemented, like the alternative layout of the chat overview discussed in section 2.4.2 on page 14.

4.3.2.1 Frameworks

Having the same motivation as already presented in section 4.2.1 on page 34, a dependency management software was chosen for the *iOS* application as well. The industry standard is *CocoaPods*. Fortunately the support for the Swift programming language was added shortly before the work on the app was started.

Thanks to the use of 3rd party libraries the amount of work needed for the app was highly reduced. Several frameworks were used to simplify the usage of *Apple*'s legacy APIs within the modern Swift environment. They include the access of the keychain and the user defaults.

The more complex frameworks were used to create a sophisticated UI, that wasn't provided by the default ones. This includes the chat view as well as the calendar view, but also the default avatar images, consisting of the initials of the user, are created dynamically by a third party library. All of these helped, to create a rich interface without the need of a professional designer, or a lot of work on the actual implementation.

Finally logging and networking frameworks are used behind the scenes to simplify the process as well as improving its performance.

4.3.2.2 Networking

When running a mobile app that needs to connect itself to a server, reliable networking is one of the most hardest thing to do, especially if a user needs to get logged into a system in order to perform any task. For simplifying the networking layer itself, the *AFNetworking* library was chosen. This library is one of the most widely used frameworks within the *iOS* community. Besides providing an easier API for performing network related tasks, it also offers the ability to perform SSL pinning and automated response parsing.

The process of sending a request is depicted as a flow chart in appendix I on page xxxii. Obviously this process is very complicated and therefore I will only highlight some of its important properties. First of all thanks to the *AFNetworking* library and some own effort, every request is executed non-blocking. That means that after the execution flow issued the wish to execute a network request, everything is dispatched asynchronously to a separate thread, not disturbing the responsiveness of the main queue.

Secondly the amount of running requests is tracked within a global counter. This value is made thread-safe, by installing a lock, protecting the critical region. This counter is increased, before the request is send, and decreased before the success or failure callbacks are executed.

Finally the system is able to distinct between a general network error and an authentication problem. In case of a general error, the failure function defined during the request creation is executed and a warning is shown to the user, where in case of an authentication error another process is started: The current and all upcoming requests, until the user is logged in, are put onto a thread safe queue, to postpone the sends until he is authenticated. Meanwhile the application tries to execute the login request using the stored user credentials. If this execution succeeds, the requests that were earlier put on the queue are getting resend. In case the authentication failes, the failure callback for each queued request is executed and the user is presented with a log-in screen, since obviously his credentials are not valid anymore.

4.3.2.3 Multi-Threading

As hinted in the last section, multi-threading is a very important part of a responsive mobile application. In case of the *iOS* operating system this is mainly based on the fact that the UI library is not thread safe and only executed on the main queue. Concluding a computing intensive execution or a blocking network request might freeze the UI, leading to a bad user experience. Every task that might block the execution for a recognisable amount of time, needs to be executed on a separate queue.

The easiest way to achieve high performance multi-threading with Swift, is by using the legacy Grand Central Dispatch (GCD) API. This low-level interface is inherited from the C language.

Implicitly this API is not using any object-oriented paradigms, but follows procedural oriented flows.

By taking advantage of the Swift feature, that enables a developer to overload or create own operators, a thin layer API is introduced based on the marshal operator described by [Smi14]. The implementation of this operator is shown in appendix J on page xxxiii. As described in lines 33 to 39, the operator can be used either infix, postfix or prefix.

The most simple version is the "∼>" operator, called "simple marshal operator". The closure left of the operator is always executed on the background thread, and the one on the right is always executed on the main thread.

When the "simple marshal operator" is used as a prefix operator it runs the closure asynchronously on the main queue, which is defined by lines 47 to 52. On the opposite it can be used as a postfix operator, where the closure is asynchronously run on a background queue to see from line 54 to 70. Since the *CoreData* framework is not thread safe, the retrieval of information on a background thread is not possible unless a `ManagedObjectContext` is created on the respective background queue. The postfix marshal operator provides a version, where the application is able to easily access the database from a background thread, by offering a suitable `ManagedObjectContext`.

However the most important application is the usage as an infix operator, defined in lines 71 to 107, where at first the closure on the left is executed on the background queue followed by the execution of the right hand closure on the main queue. This model is widely used within the *iOS* world. A version of this operator is provided, which is handing an arbitrary object from the background queue to the main queue.

The "inverse marshal operator", defined by line 109 to 144, is used to asynchronously execute a closure infinitely every couple of seconds. The infix or postfix operator uses the "<∼" sign, where the left hand side is a closure and the right hand side is an optional double value. This value is defining the amount of seconds, the application should wait in between the executions of the closure. If the amount is not specified the current default value is 50 seconds. When performing this operation a `MarshalThreadingObject` is returned. As long as this object is referenced, the execution of the thread will proceed. Without any reference to this object, the thread is going to get destroyed by the garbage collector.

Last but not least the regular marshal operator can have a double value instead of a closure on its left hand side. This indicates, that the execution of the right hand queue should be delayed by the amount of time defined by the value. The delayed execution is done on the main queue. This behaviour is defined in line 146 to 154.

4.3.2.4 UI

During the creation of the UI the mockups discussed in section 3.4.2 on page 29 were closely analysed. Unfortunately not all of them could be realised, due to the reduced set of functionality, as well as the limited project resources. Nevertheless a suitable interface was created, shown in appendix K on page xxxvii. This representation is similar to the original mockups in appendix C on page xx, showing a storyboard with all reachable states from each view.

The storyboard is only showing the most important screens of the application. One of them is the chat overview, which is adopting the alternative design proposed in section 2.4.2.2 on page 16. Another view, which was not considered during the initial design is the event details view, showing all information available for a specific event. This includes the event times, a short description and a map view with the precise location of the event.

Finally the colour scheme and app icon shown in section 4.1 on page 32 were adopted. The dominant green colour was created for its similarity to the web application. The alternative colours of the scheme are only used to attract attention to notifications and warnings.

Chapter 5

Retrospection and future work

I am very happy about the progress of this project. Even though a couple of features that were specified as optional requirements and none of the additional requirements made it into the version handed in with this paper, a lot of fundamental research and work has been done. This is going to simplify the upcoming tasks of this project.

Up to this point none of the major design decisions were regretted, nevertheless latest insights showed that a document oriented database might not have been the most suitable back-end store for this system. Maybe a graph database or even a conventional relational database could have been a better pick. Fortunately no major drawback, that would have lead to a change of technology, has been experienced so far.

The next steps will mainly include the implementation of the missing requirements specified in section 3.1 on page 21. Additionally a restructuring of the back-end server is considered. Strictly separating the web application from the back-end logic would increase the maintainability and might even improve performance by decreasing the workload on the core service. Nevertheless this might imply some design changes within the administrator interface that need to be considered first.

After completing the implementation of the requirements, the main objective is stability. It will be a top priority to get the platform run smoothly and free of complication, to prepare for a public testing phase, followed by an initial release. Together with this release, a streamlined process needs to be established to enable an automatic setup of instances. This might include the possibility to allocate resources on demand within a cloud environment.

Based on the feedback during this time, the future of the platform is determined. In case of a positive reception, the monetarization of the software might be considered. This can be done, by providing a Software-as-a-Service (SaaS) concept, where the technical resources of the back-end as well as the required knowledge to setup and run the system is provided. By charging

an appropriate monthly fee, either based on the size of the club and/or the traffic produced by the users, the costs of running the servers and allocating the workforce are getting compensated.

Following these steps the continuous improvement of the system must not be ignored. As shown in section 2.3 on page 11, there are many functionalities, that haven't been considered so far, since their implementation was not any requirement at all but would give another push to ameliorate customer's satisfaction. By adding them to the platform, the introduction of *myVerein* is surely going to be more and more appealing.

Finally it is important to note, that the future work presented in this chapter is going to be very time consuming. Since the development of the system is currently done by a single person, this process might take years to finish and without additional support from other software engineers the system will be most certainly outdated before its initial release. From the early beginning the system's license was fully intended to be open-source to enable independent developers to contribute to this project. Therefore an important task of the near future will include the acquisition of skilled people sharing my vision of disrupting the management of clubs and increasing the size of the team working on *myVerein*.

List of Figures

List of Tables

Listings

Bibliography

[15a] *Colour Scheme*. 2015. URL: http://paletton.com/#uid=7320u0kt4Gxh5M7n8IdGe-SH-q5.

[15b] *Leistungsübersicht WISO Mein Verein*. 2015th ed. Buhl Data Service. 2015.

[15c] *Produkte*. 2015. URL: https://www.buhl.de/produkte/alle/page.html.

[Abr14] Luis Abreu. *Why and How to Avoid Hamburger Menus*. May 2014. URL: https://lmjabreu.com/post/why-and-how-to-avoid-hamburger-menus/.

[Cha10a] Cameron Chapman. *Color Theory for Designers, Part 1: The Meaning of Color*. 2010. URL: http://www.smashingmagazine.com/2010/01/28/color-theory-for-designers-part-1-the-meaning-of-color/.

[Cha10b] Cameron Chapman. *Color Theroy For Designers: Creating Your Own Color Palettes*. Feb. 2010. URL: Monochromatic.

[Com15] Creative Commons. 2015. URL: https://creativecommons.org/licenses/by-nc-sa/4.0/.

[Cre15] Andy Crestodina. *Are you making these common website navigation mistakes?* 2015. URL: https://blog.kissmetrics.com/common-website-navigation-mistakes/.

[Dah11] Darren Dahl. *10 Tips on How to Research Your Competition*. 2011. URL: http://www.inc.com/guides/201105/10-tips-on-how-to-research-your-competition.html.

[Fla15a] Michael Flarup. *How to Design Better App Icons*. Mar. 2015. URL: http://blog.appicontemplate.com/how-to-design-better-app-icons/.

[Fla15b] Michael Flarup. *Icons and Logos are not the Same*. Jan. 2015. URL: http://blog.appicontemplate.com/icons-and-logos-are-not-the-same/.

[Fou91] Free Software Foundation. *GNU General Public License, version 2*. 1991. URL: http://www.gnu.org/licenses/gpl-2.0.html.

[Fra13] Dana Frank. *What is Usability and Why is it Important to Application Development?* Tech. rep. Segue Technologies, 2013. URL: http://www.wiso-meinverein.de/LeistungsuebersichtMeinVerein.pdf.

[Gül15] Dr.-Ing. Thomas Gülzow. *Vorlesung Interaktive Systeme - Usability Engineering.* Presentation. 2015.

[Ham13] Sam Hampton-Smith. *22 great examples of website navigation.* 2013. URL: http://www.creativebloq.com/web-design/website-navigation-4132549.

[Hun15] Micheal R. Hunter. *3 Reasons your competitors are your best friends.* 2015. URL: http://michaelrhunter.com/why-knowing-your-competition-is-important/.

[IDC15a] IDC. *Smartphone OS Market Share, Q4 2014.* 2015. URL: http://www.idc.com/prodserv/smartphone-os-market-share.jsp.

[IDC15b] IDC. *Smartphone Vendor Market Share, Q4 2014.* 2015. URL: http://www.idc.com/prodserv/smartphone-market-share.jsp.

[Inc15] Apple Inc. "Messages. Unlimited texting. Unlimited fun." In: (2015). URL: http://www.apple.com/ios/messages/.

[Jas06] Barry Jaspan. *Improved Persistent Login Cookie Best Practive.* Nov. 2006. URL: jaspan.com/improved_persistent_login_cookie_best_practice.

[Mon14] Inc. MongoDB. *MongoDB Documentation.* Release 2.6.4. MongoDB, Inc. Sept. 2014.

[Mos14] Buzz Moschetti. *MongoDB vs SQL.* Nov. 2014. URL: http://www.mongodb.com/blog/post/mongodb-vs-sql-day-1-2.

[Mur14] Rebecca Murtagh. *Mobile Now Exceeds PC: The Biggest Shift Since the Internet Began.* 2014. URL: http://searchenginewatch.com/sew/opinion/2353616/mobile-now-exceeds-pc-the-biggest-shift-since-the-internet-began (visited on 2015).

[Nie93] Jakob Nielsen. *Usability engineering.* Morgan Kaufmann, 1993.

[Ope14] OpenSignal. *Android Fragmentation 2014.* Tech. rep. OpenSignal, 2014. URL: http://opensignal.com/assets/pdf/reports/2014_08_fragmentation_report.pdf.

[Phi15] Stepen Philips. *Keeping your eyes on the competition.* 2015. URL: http://www.marketingdonut.co.uk/marketing/market-research/market-analysis/keeping-your-eyes-on-the-competition.

[Smi14] Josh Smith. *Custom Threading Operator in Swift.* 2014. URL: http://ijoshsmith.com/2014/07/05/custom-threading-operator-in-swift/.

[Sta11] Petr Stanicek. *Color Schemes*. Aug. 2011. URL: http://www.colorschemedesigner.com/blog/color-schemes/.

[Ste14a] Frank Steiler. *myVerein - Server Specification*. Tech. rep. DHBW Stuttgart, 2014.

[Ste14b] Frank Steiler. *myVerein - Software Requirement Specification*. Tech. rep. DHBW Stuttgart, 2014.

[Stü13] Moritz Stückler. *Update: WhatsApp endlich im iOS-7-Design erhältlich*. 2013. URL: http://t3n.de/news/whatsapp-update-endlich-513599/.

[Wri15] Mark Wright. *2-tier vs. 3-tier application architecture? Could the winner be 2-tier?* May 2015. URL: http://www.nitrosphere.com/2015/05/14/2-tier-vs-3-tier-application-architecture-could-the-winner-be-2-tier-2/.

Acronyms

AJAX Asynchronous JavaScript and XML.

AJP Apache JServ Protocol.

API Application Programmable Interface.

BDS Buhl Data Service.

CC BY-NC-SA 4.0 Creative Commons - Attribution - Non Commercial - Share Alike 4.0 International license.

CSS Cascading Style Sheet.

ER Entity Relationship.

GCD Grand Central Dispatch.

GNU GPL v2 GNU General Public License, version 2.

HTML Hypertext Markup Language.

ISO International Organization for Standartization.

IT Information Technology.

JS JavaScript.

JSON JavaScript Object Notation.

JSP Java Sever Pages.

JVM Java Virtual Machine.

MVC Model-View-Controller.

NoSQL not only SQL.

OS operating system.

REST Representational State Transfer.

RSS Rich Site Summary.

SaaS Software-as-a-Service.

SDK Software Development Kit.

UI User Interface.

URI Unified Resource Identifier.

URL Unified Resource Locator.

WOCA write once, compile anywhere.

WORA write once, run anywhere.

Glossary

Apache JServ Protocol

A binary protocol used to route requests from a web server to an application server, supporting load balancing mechanics.

Asynchronous JavaScript and XML

Concept of asynchronous data transmission between a browser and web server, leading to the possibility of loading additional data without the need of refreshing the webpage.

JavaScript Object Notation

Compact, non-binary data format used to exchange information between applications.

Least Privileges

Fine granular regulated access to data, only for users that need the data obligatorily.

Model-View-Controller

Software engineering model, strictly separating data provider (model), presentation (view) and business logic (controller).

NoSQL

Category of database system, that do not use a relational approach for organising and storing data also called structured storage.

Representational State Transfer

Programming paradigm within the world wide web, describing properties a server has to fulfil within the context of machine-to-machine communication.

Rich Site Summary

Also known as *Really Simple Syndication* is a web standard, to publish a feed of frequently changing information.

Swift

An object-oriented programming language for iOS and OSX, developed by Apple and beta-released June 2014.

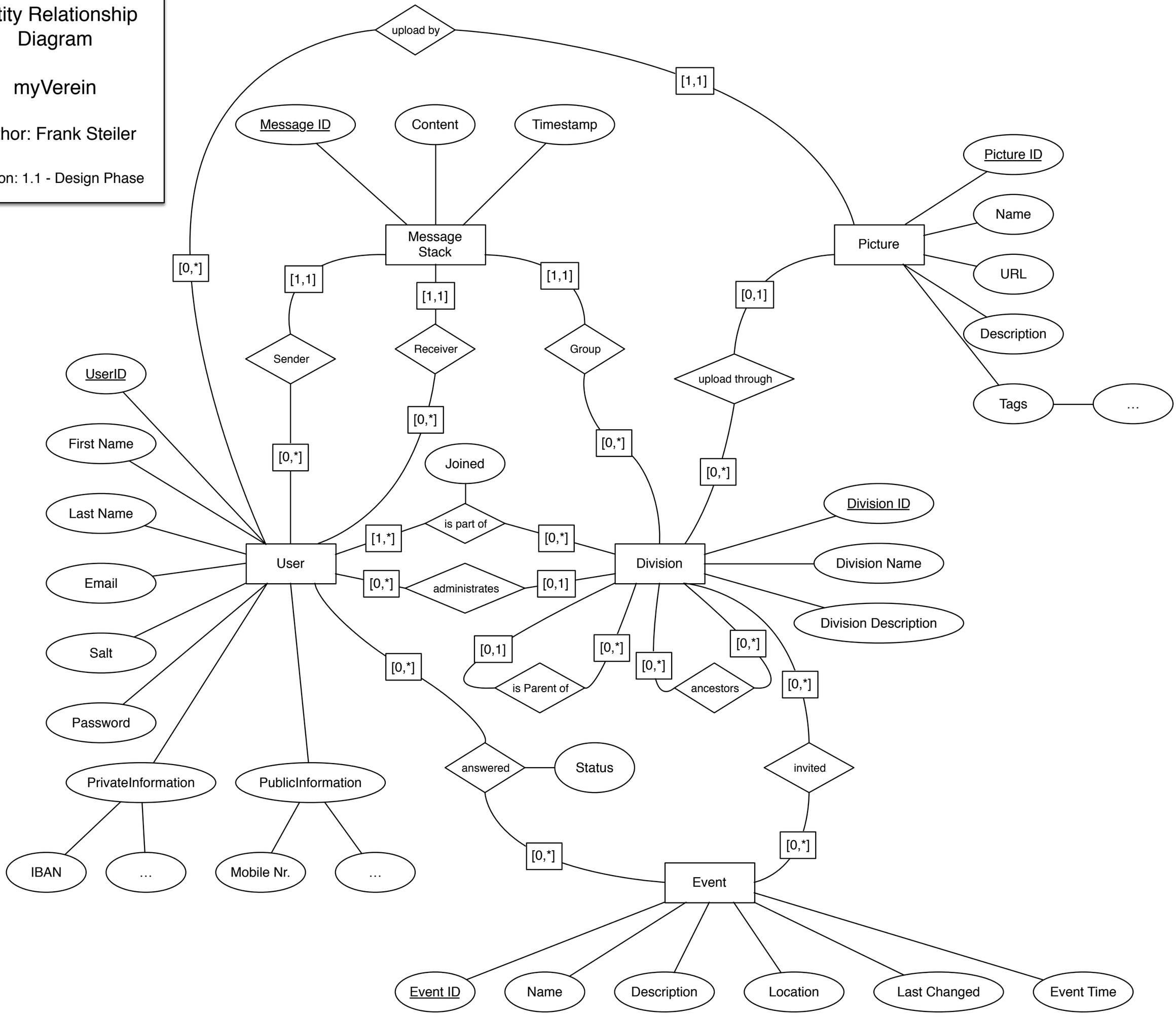
Appendices

Entity Relationship Diagram

myVerein

Author: Frank Steiler

Version: 1.1 - Design Phase

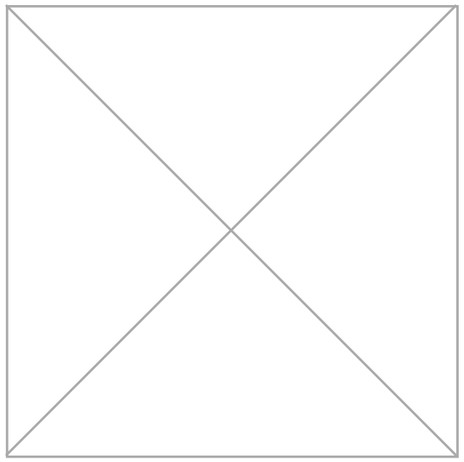


myVerein - Login

http://verein.myVerein-app.de

Google

| myVerein | Users | Divisions | Events | Photos | Settings |



E-mail

Password

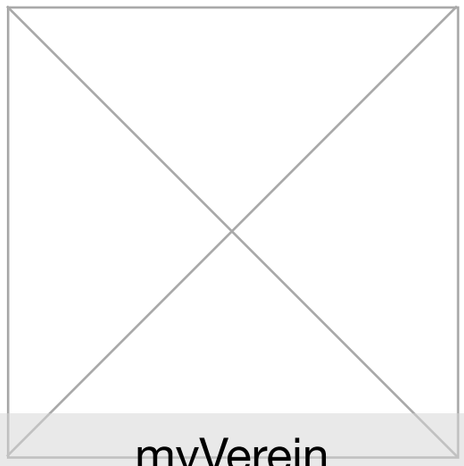
Login

myVerein

http://verein.myVerein-app.de/

Google

| myVerein | Users | Divisions | Events | Photos | Settings |



myVerein

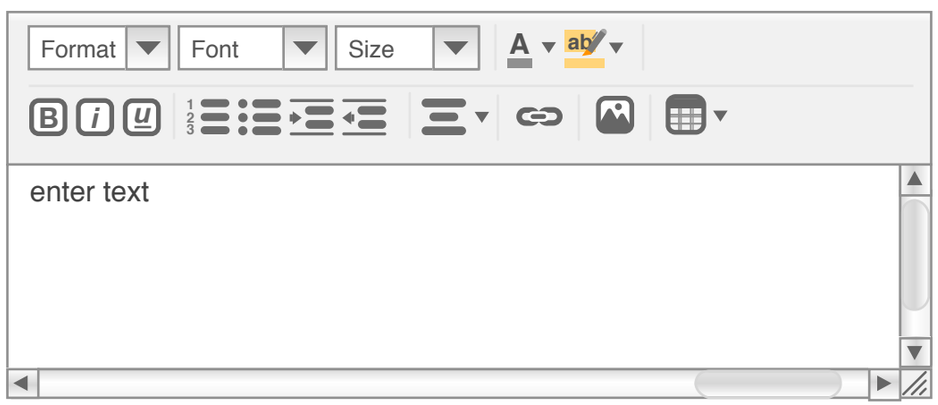
230 user in 40 divisions

Recent activities

5 new photos uploaded

3 upcoming events

Publish news



myVerein – User Management

http://verein.myVerein-app.de/users

Google

myVerein | Users | Divisions | Events | Photos | Settings

Find User

Find

- John Doe
- Bruce Wayne
- Peter Griffin
- Obi-Wan Kenobi
- Eric Cartman
- Luke Skywalker
- Lin Beifong
- Ted Mosby
- Marshall Erikson
- Marc Zuckerberg
- Homer Simpson
- Bart Simpson
- Marti McFly
- Karl May
- Steve Jobs
- Larry Page
- Ben Horowitz
- Charlie Chaplin
- Peter Parker
- Max Mustermann
- Georg Lukas
- Stanley Kubrick

First Name: John

Last Name: Doe

Email: john.doe@icloud.com

IBAN: DE00000000000000000000

BIC: BYLADEM1111

Member since: 01.01.1978

Add field

Divisions: Soccer, Rugby

Mobile user | Email newsletter | Reset Password | Save

myVerein – Division Management

http://verein.myVerein-app.de/divisions

Google

| myVerein | Users | Divisions | Events | Photos | Settings |

- mySportVerein
  - Ball sports
    - Basketball
      - Basketball - 1st Team
      - Basketball - 2nd Team
    - Soccer
      - Soccer - 1st Team
      - Soccer - 2nd Team
      - Soccer - 3rd Team
      - Soccer - 4th Team
    - Football
    - Rugby
    - Baseball
    - Handball
    - Hokey
    - Ice-Hokey
  - Gymnastics division
    - Artistic gymnastics
      - High Bar
      - Bars
      - Pommel horse
    - Floor exercises
  - Swimming division
    - Sprint
      - 100m freestyle
      - 100m breaststroke
      - 200m
    - Long distance
  - Athletics
    - High Jump

Add division

Division Name

Basketball - 1st Team

Description

–

Administrator

John Doe

Activate group chat for this division

Delete

Save

myVerein - Event Management

http://verein.myVerein-app.de/events

Google

myVerein | Users | Divisions | Events | Photos | Settings

Your club calendar

December 2008

S	M	T	W	T	F	S
30	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31	1	2	3

Select Today

- Training - 1st Team Basketball - 19:00 - 21:00
- Training - 1st Team Soccer - 18:00 - 21:00

Create new event

Event name

Training - 1st Team Basketball

Event Description

The 1st team of the Basketball team's training.

Start time

22.11.2008, 19:00

End time

22.11.2008, 21:00

Location

Sports hall, Park Drive

Invited Divisions

Basketball 1st team

Delete Save

myVerein - Photo Management

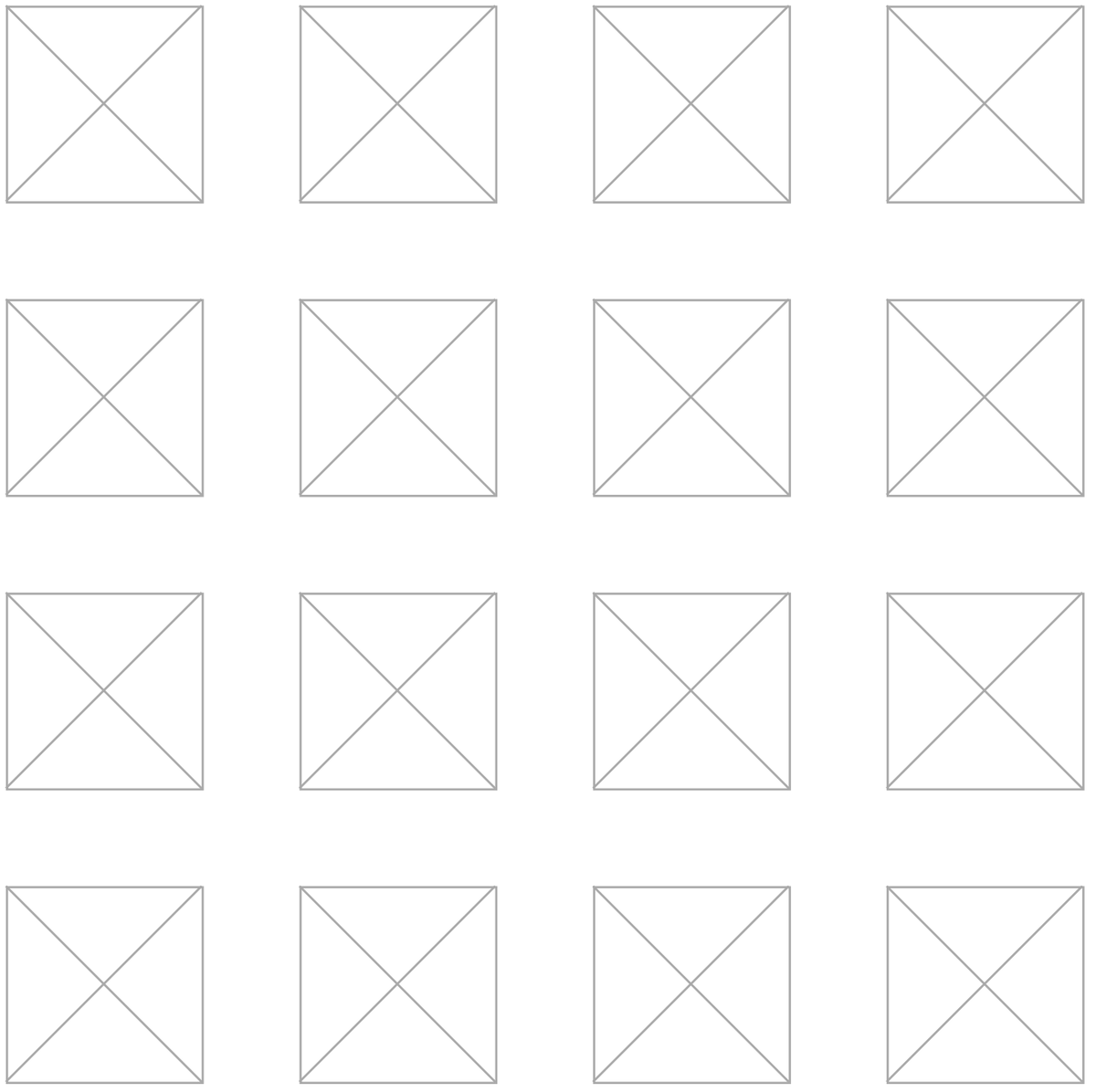
http://verein.myVerein-app.de/photos

Google

| myVerein | Users | Divisions | Events | Photos | Settings |

- mySportVerein
  - Ball sports
    - Basketball
      - Basketball - 1st Team
      - Basketball - 2nd Team
    - Soccer
      - Soccer - 1st Team
      - Soccer - 2nd Team
      - Soccer - 3rd Team
      - Soccer - 4th Team
    - Football
    - Rugby
    - Baseball
    - Handball
    - Hokey
    - Ice-Hokey
  - Gymnastics division
    - Artistic gymnastics
      - High Bar
      - Bars
      - Pommel horse
    - Floor exercises
  - Swimming division
    - Sprint
      - 100m freestyle
      - 100m breaststroke
      - 200m
    - Long distance
  - Athletics
    - High Jump

Pictures of Basketball



View selected | Download selected | Delete selected

myVerein - Settings

http://verein.myVerein-app.de/settings

Google

| myVerein | Users | Divisions | Events | Photos | Settings |

Super admin user name

john.doe@icloud.com

Super admin passwort

••••••••••••••••••••••••••••••

Re-enter super admin passwort

••••••••••••••••••••••••••••••

Database host

https://localhost:12345

Database user

root

Database password

••••••••••••••••••••••••••••••

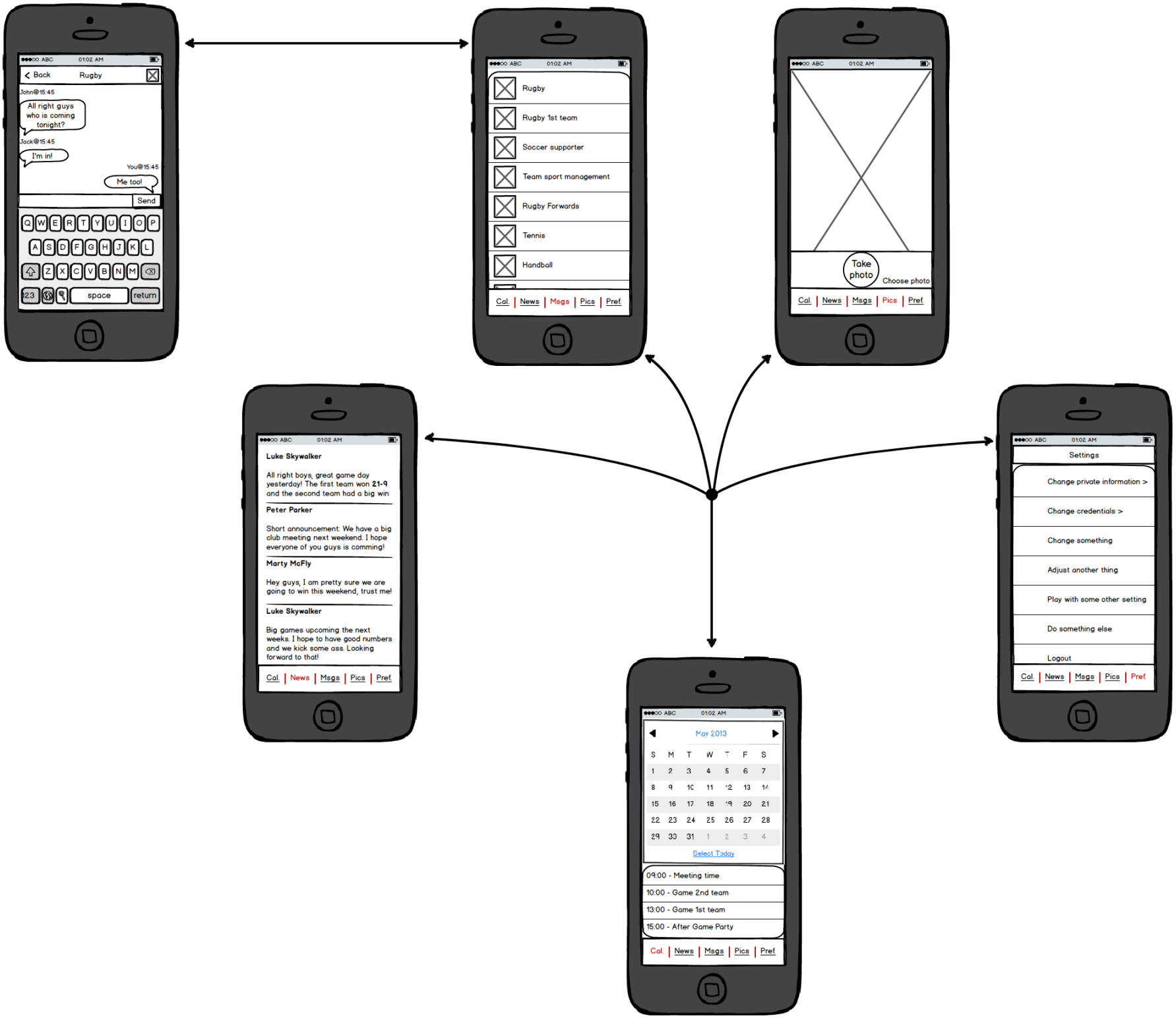
Club name

myVerein

Club logo

Upload logo

Save


Back
Rugby
John@15:45
All right guys who is coming tonight?
Jack@15:45
I'm in!
You@15:45
Me too!
Send
Rugby
Rugby 1st team
Soccer supporter
Team sport management
Rugby Forwards
Tennis
Handball
Take photo
Choose photo
Cal. | News | Msgs | Pics | Pref.
Luke Skywalker
All right boys, great game day yesterday! The first team won 21-9 and the second team had a big win
Peter Parker
Short announcement: We have a big club meeting next weekend. I hope everyone of you guys is comming!
Marty McFly
Hey guys, I am pretty sure we are going to win this weekend, trust me!
Luke Skywalker
Big games upcoming the next weeks. I hope to have good numbers and we kick some ass. Looking forward to that!
May 2013
Select Today
09:00 - Meeting time
10:00 - Game 2nd team
13:00 - Game 1st team
15:00 - After Game Party
Settings
Change private information >
Change credentials >
Change something
Adjust another thing
Play with some other setting
Do something else
Logout

log4j2.xml

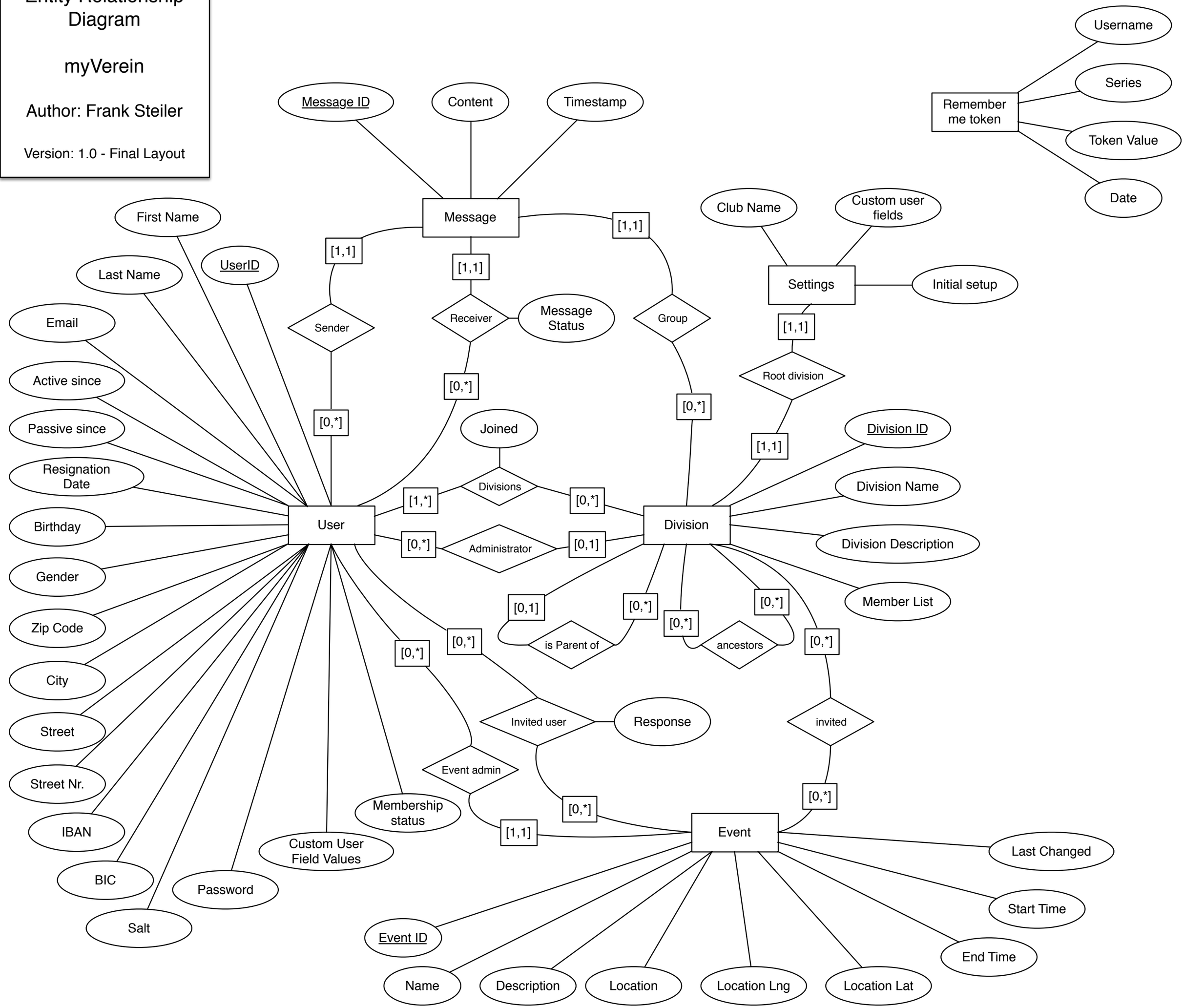
```xml
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
    <Properties>
        <Property name="third-party-log-level">info</Property>
    </Properties>

    <Appenders>
        <Console name="console" target="SYSTEM_OUT">
            <PatternLayout pattern="%d{ISO8601} %p [%t] %c{1} [%M] - %m%throwable{short}%n"/>
        </Console>

        <RollingFile name="tomcat-file-log" fileName="${sys:catalina.base}/logs/myVerein.log" filePattern="myVerein-%d{yyyy-MM-dd}.log" append="true">
            <PatternLayout pattern="%d{ISO8601} %p [%t] %c{1} [%M] - %m%throwable{short}%n"/>
            <Policies>
                <TimeBasedTriggeringPolicy interval="1" modulate="true"/>
                <SizeBasedTriggeringPolicy size="50 MB" />
            </Policies>
        </RollingFile>
    </Appenders>

    <Loggers>
        <Logger name="de.steilerdev.myVerein" level="trace" />

        <Logger name="org.springframework.core" level="${third-party-log-level}" />

        <Logger name="org.springframework.data.mongodb" level="${third-party-log-level}" />

        <Logger name="org.springframework.beans" level="${third-party-log-level}" />

        <Logger name="org.springframework.context" level="${third-party-log-level}" />

        <Logger name="org.springframework.http" level="${third-party-log-level}" />

        <Logger name="org.springframework.web" level="${third-party-log-level}" />

        <Logger name="org.springframework.security" level="${third-party-log-level}" />

        <Logger name="org.thymeleaf.TemplateEngine" level="${third-party-log-level}" />

        <Logger name="com.relayrides.pushy" level="trace" />

        <Root level="error">
            <AppenderRef ref="console"/>
            <AppenderRef ref="tomcat-file-log"/>
        </Root>
    </Loggers>
</Configuration>
```

Entity Relationship Diagram

myVerein

Author: Frank Steiler

Version: 1.0 - Final Layout

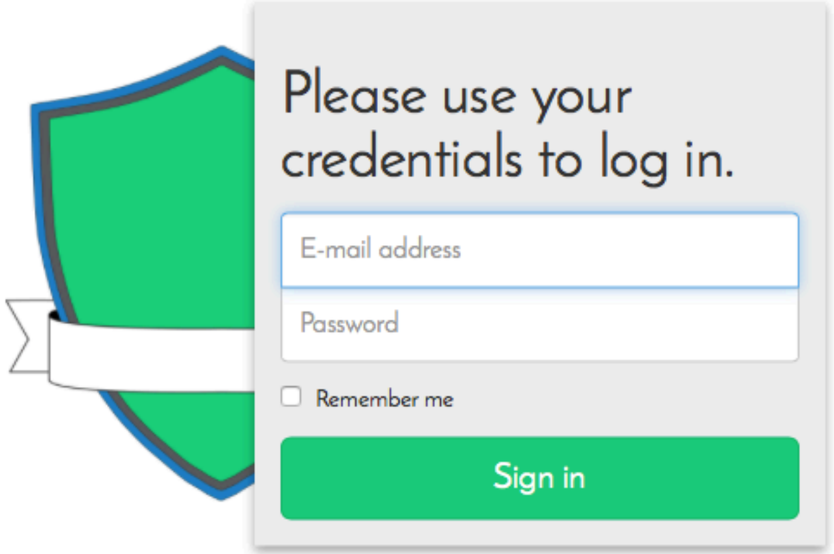


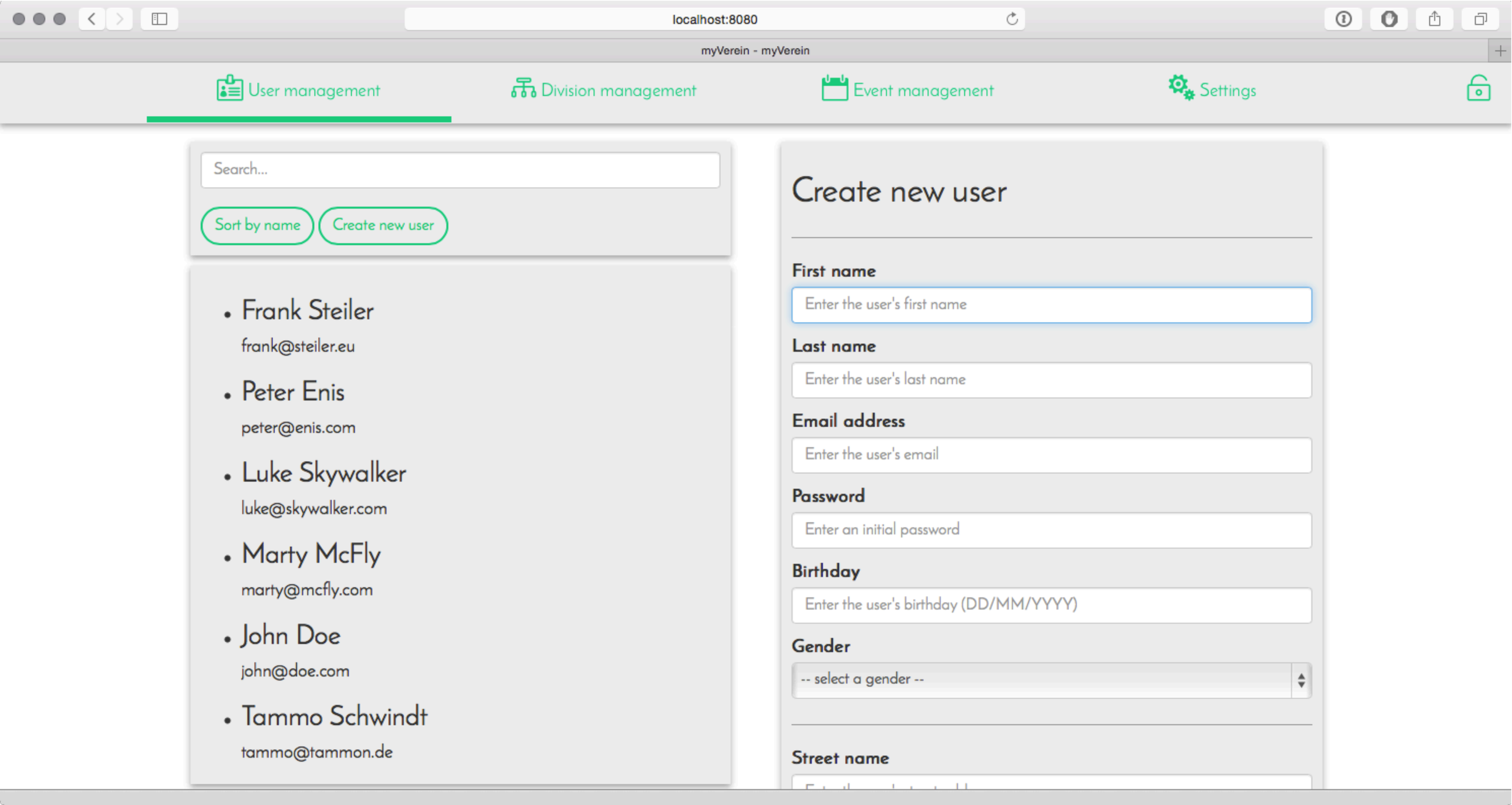
```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/security"
             xmlns:beans="http://www.springframework.org/schema/beans"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="  http://www.springframework.org/schema/beans
                                   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
                                   http://www.springframework.org/schema/security
                                   http://www.springframework.org/schema/security/spring-security.xsd">

        <!-- Unsecured content -->
        <http pattern="/resources/**" security="none" /> <!-- Static content (JS/CSS/etc.) -->
        <http pattern="/content/**" security="none" /> <!-- Variable content served by controller -->

        <!-- REST API specially secured, to comply with REST standards -->
        <http auto-config="false" use-expressions="true" pattern="/api/**" entry-point-ref="restAuthenticationEntryPoint">
            <intercept-url pattern="/api/login" access="permitAll" />
            <intercept-url pattern="/api/logout" access="permitAll" />
            <intercept-url pattern="/api/init/**" access="permitAll" />
            <intercept-url pattern="/api/user/**" access="hasRole('ROLE_USER')"/>
            <intercept-url pattern="/api/**" access="hasRole('ROLE_ADMIN')"/>

            <form-login authentication-success-handler-ref="restAuthenticationSuccessHandler"
                        authentication-failure-handler-ref="restAuthenticationFailureHandler"
                        username-parameter="username"
                        password-parameter="password"
                        login-processing-url="/api/login"/>

            <remember-me services-ref="rememberMeServices" />

            <logout logout-url="/api/logout"
                    invalidate-session="true"
                    success-handler-ref="restLogoutSuccessHandler" />
        </http>

        <beans:bean id="restAuthenticationSuccessHandler"
                    class="de.steilerdev.myVerein.server.security.rest.RestAuthenticationSuccessHandler"/>
        <beans:bean id="restAuthenticationFailureHandler"
                    class="de.steilerdev.myVerein.server.security.rest.RestAuthenticationFailureHandler"/>
        <beans:bean id="restLogoutSuccessHandler"
                    class="de.steilerdev.myVerein.server.security.rest.RestLogoutSuccessHandler"/>
        <beans:bean id="restAuthenticationEntryPoint"
                    class="de.steilerdev.myVerein.server.security.rest.RestAuthenticationEntryPoint"/>

        <!-- Web pages secured through the standard implementation of spring security -->
        <http auto-config="false" use-expressions="true" access-denied-page="/login?accessdenied">
            <intercept-url pattern="/login" access="permitAll" />
            <intercept-url pattern="/logout" access="permitAll" />
            <intercept-url pattern="/error" access="permitAll" />
            <intercept-url pattern="/**" access="hasRole('ROLE_ADMIN')" />

            <form-login login-page="/login"
                        default-target-url="/"
                        username-parameter="username"
                        password-parameter="password"
                        authentication-failure-url="/login?error"
                        login-processing-url="/login_check"/>

            <logout logout-url="/logout"
                    logout-success-url="/login?logout"
                    delete-cookies="myVereinSessionCookie"
                    invalidate-session="true" />

            <remember-me services-ref="rememberMeServices" />
        </http>

        <!-- Remember me configuration -->
        <beans:bean id="rememberMeServices"
                    class="org.springframework.security.web.authentication.rememberme.PersistentTokenBasedRememberMeServices">
            <beans:constructor-arg value="myVereinRememberMeKey"/>
            <beans:constructor-arg ref="userAuthenticationService"/>
            <beans:constructor-arg ref="tokenDataProvider"/>
            <beans:property name="cookieName" value="myVereinRememberMeCookie"/>
            <beans:property name="tokenLength" value="32"/>
            <beans:property name="parameter" value="rememberMe"/>
            <beans:property name="tokenValiditySeconds" value="604800" />
            <!-- Todo: enable secure cookie -->
            <!--beans:property name="useSecureCookie" value="true" /-->
        </beans:bean>
        <!-- Permanently storing remember me tokens within the database -->
        <beans:bean id="tokenDataProvider" class="de.steilerdev.myVerein.server.security.RememberMeTokenDataProvider" />

        <!-- Authentication configuration -->
        <authentication-manager>
              <!--  Plain text authentication without database as a fall back solution if
                    a password got lost, or the system compromised -->
              <authentication-provider>
                    <user-service properties="classpath:myVereinUser.properties" />
              </authentication-provider>
              <!-- Authentication against database -->
              <authentication-provider user-service-ref="userAuthenticationService">
                    <password-encoder ref="passwordEncoder">
                          <salt-source user-property="salt" />
```

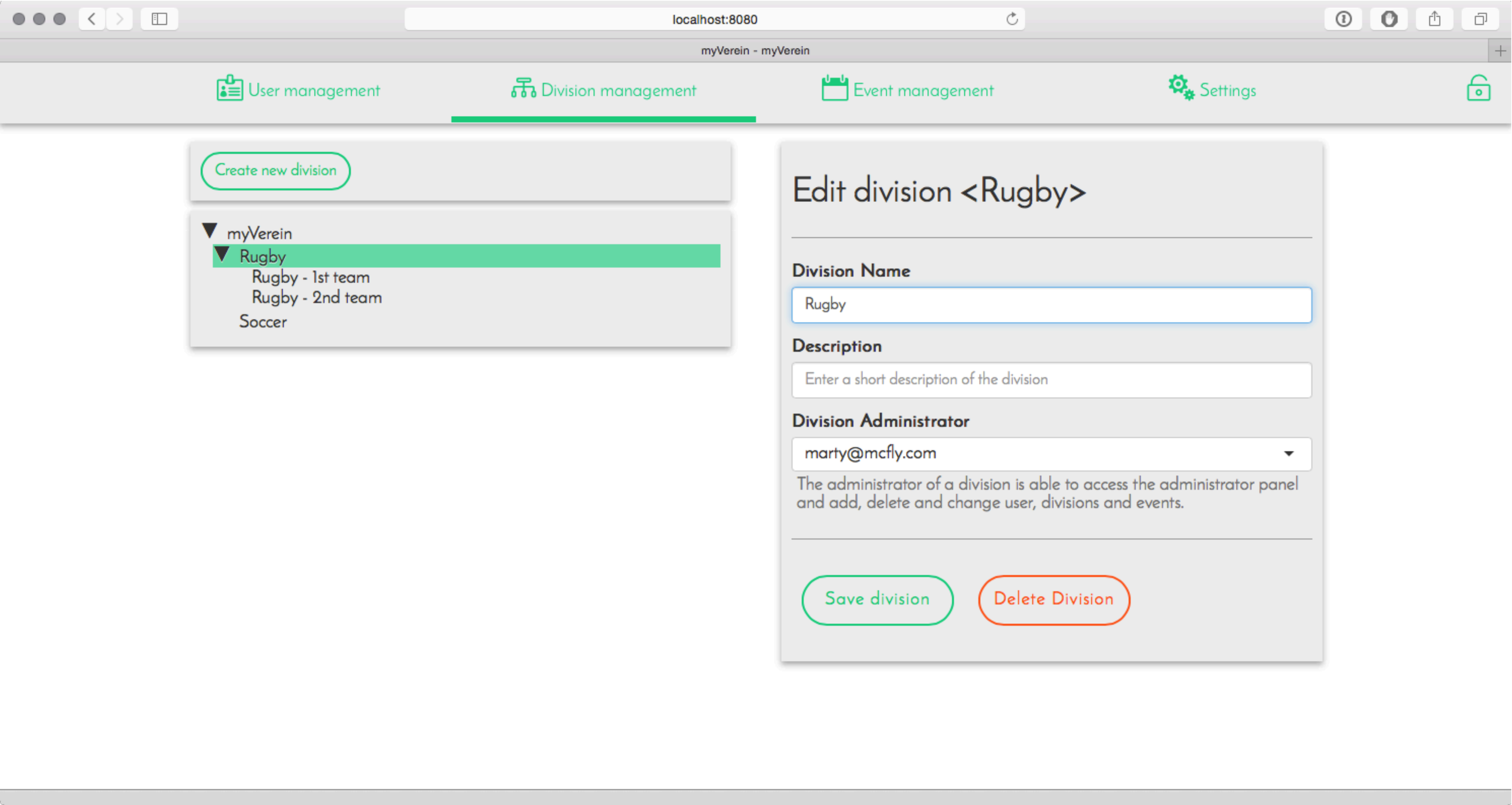
```
                    </password-encoder>
                </authentication-provider>
                <authentication-provider ref="rememberMeAuthenticationService"/>
        </authentication-manager>

        <!-- Password encoder with SHA-512 hashing method and 1000 iterations using 8byte salting -->
        <beans:bean id="passwordEncoder"
                    class="de.steilerdev.myVerein.server.security.PasswordEncoder" />

        <beans:bean id="userAuthenticationService"
                    class="de.steilerdev.myVerein.server.security.UserAuthenticationService" />

        <beans:bean id="rememberMeAuthenticationService"
                    class="org.springframework.security.authentication.RememberMeAuthenticationProvider">
            <beans:constructor-arg value="myVereinRememberMeKey"/>
        </beans:bean>
</beans:beans>
```

```java

    /**
     * This function replaces the set of divisions by the stated divisions. The function guarantees that the inverse membership is handled correctly.
     * @param divisionRepository The division repository needed to save the altered divisions.
     * @param eventRepository The event repository needed to save the altered events.
     * @param divs The new list of divisions for the user.
     */
    public void replaceDivisions(DivisionRepository divisionRepository, EventRepository eventRepository, List<Division> divs)
    {
        logger.debug("[{}] Replacing division set", this);

        List<Division> finalDivisions = DivisionHelper.getExpandedSetOfDivisions(divs, divisionRepository);
        List<Division> oldDivisions = divisions;

        if((finalDivisions == null || finalDivisions.isEmpty()) && (oldDivisions == null || oldDivisions.isEmpty()))
        {
            logger.debug("[{}] Division sets before and after are both empty", this);
            divisions = new ArrayList<>();
        } else if(finalDivisions == null || finalDivisions.isEmpty())
        {
            logger.debug("[{}] Division set after is empty, before is not. Removing membership subscription from old divisions", this);
            oldDivisions.stream().forEach(div -> div.removeMember(this));
            divisionRepository.save(oldDivisions);

            //Updating events, affected by division change
            oldDivisions.parallelStream().forEach(div -> {
                List<Event> changedEvents = eventRepository.findByInvitedDivision(div);
                changedEvents.parallelStream().forEach(event -> event.updateInvitedUser(divisionRepository));
                eventRepository.save(changedEvents);
            });
            divisions = new ArrayList<>();
        } else if(oldDivisions == null || oldDivisions.isEmpty())
        {
            logger.debug("[{}] Division set before is empty, after is not. Adding membership subscription to new divisions", this);
            finalDivisions.stream().forEach(div -> div.addMember(this));
            divisionRepository.save(finalDivisions);

            //Updating events, affected by division change
            finalDivisions.parallelStream().forEach(div -> {
                List<Event> changedEvents = eventRepository.findByInvitedDivision(div);
                changedEvents.parallelStream().forEach(event -> event.updateInvitedUser(divisionRepository));
                eventRepository.save(changedEvents);
            });
            divisions = finalDivisions;
        } else
        {
            logger.debug("[{}] Division set after and before are not empty. Applying changed membership subscriptions", this);
            List<Division> intersect = finalDivisions.stream().filter(oldDivisions::contains).collect(Collectors.toList()); //These items are already in the list, and do not need to be modified

            //Collecting changed division for batch save
            List<Division> changedDivisions = Collections.synchronizedList(new ArrayList<>());

            //Removing membership from removed divisions
            oldDivisions.parallelStream()
                    .filter(div -> !intersect.contains(div))
                    .forEach(div -> {
                        div.removeMember(this);
                        changedDivisions.add(div);
                    });

            //Adding membership to added divisions
            finalDivisions.parallelStream()
                    .filter(div -> !intersect.contains(div))
                    .forEach(div -> {
```

```
                        div.addMember(this);
                        changedDivisions.add(div);
                    });

            divisionRepository.save(changedDivisions);

            //Updating events, affected by division change
            changedDivisions.parallelStream().distinct().forEach(div -> {
                List<Event> changedEvents = eventRepository.findByInvitedDivision(div);
                changedEvents.parallelStream().distinct().forEach(event -> event.updateInvitedUser(divisionRepository));
                eventRepository.save(changedEvents);
            });
            divisions = finalDivisions;
        }
    }
```


Please use your credentials to log in.
E-mail address
Password
Remember me
Sign in


localhost:8080
myVerein - myVerein
User management
Division management
Event management
Settings
Search...
Sort by name
Create new user
Frank Steiler
frank@steiler.eu
Peter Enis
peter@enis.com
Luke Skywalker
luke@skywalker.com
Marty McFly
marty@mcfly.com
John Doe
john@doe.com
Tammo Schwindt
tammo@tammon.de
Create new user
First name
Enter the user's first name
Last name
Enter the user's last name
Email address
Enter the user's email
Password
Enter an initial password
Birthday
Enter the user's birthday (DD/MM/YYYY)
Gender
-- select a gender --
Street name


localhost:8080
myVerein - myVerein
User management
Division management
Event management
Settings
Create new division
myVerein
Rugby
Rugby - 1st team
Rugby - 2nd team
Soccer
Edit division <Rugby>
Division Name
Rugby
Description
Enter a short description of the division
Division Administrator
marty@mcfly.com
The administrator of a division is able to access the administrator panel and add, delete and change user, divisions and events.
Save division
Delete Division

Create new event

‹ APRIL 2015 ›

S	M	T	W	T	F	S
29	30	31	1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	1	2
3	4	5	6	7	8	9

EVENTS

11th Apr 04:00 - 15th Apr 04:05: Super Event 4

Edit event <Super Event 4>

Event name

Super Event 4

Description

Enter a short description for the event

Start date & time

11/04/2015

04:00

End date & time

15/04/2015

04:05

Location

Enter the location of the event

Change system settings

Super admin user

frank@steiler.eu

Change your password

Enter a new password to change it

Re-type the new password to change it

If you want to change your password enter a new one, otherwise leave this field empty.

Club name

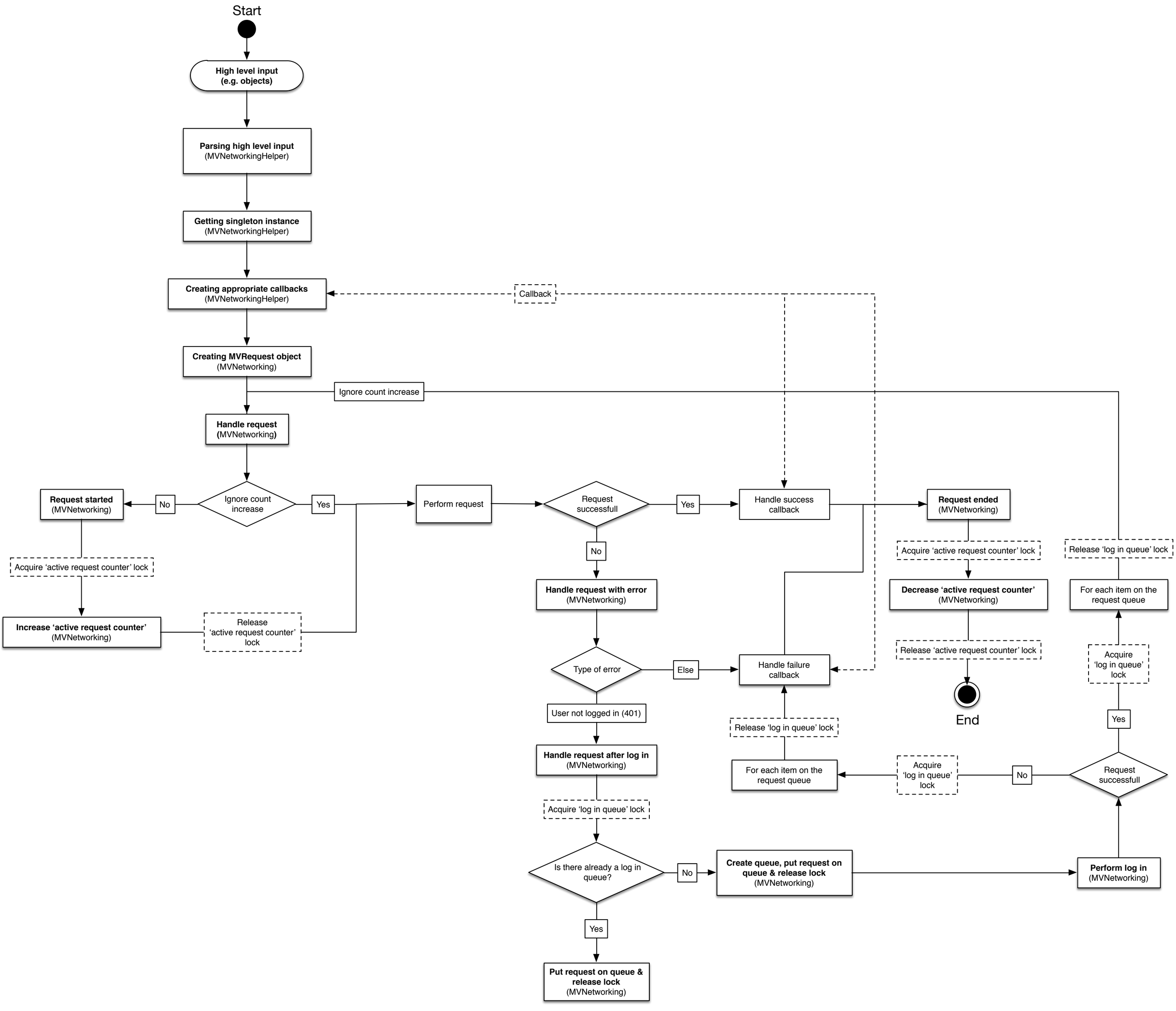
myVerein

Club logo

Browse No file selected

Upload a new logo, if you want to override your current one

If you want to change the database settings, you need to adjust the properties file on your host and restart the application

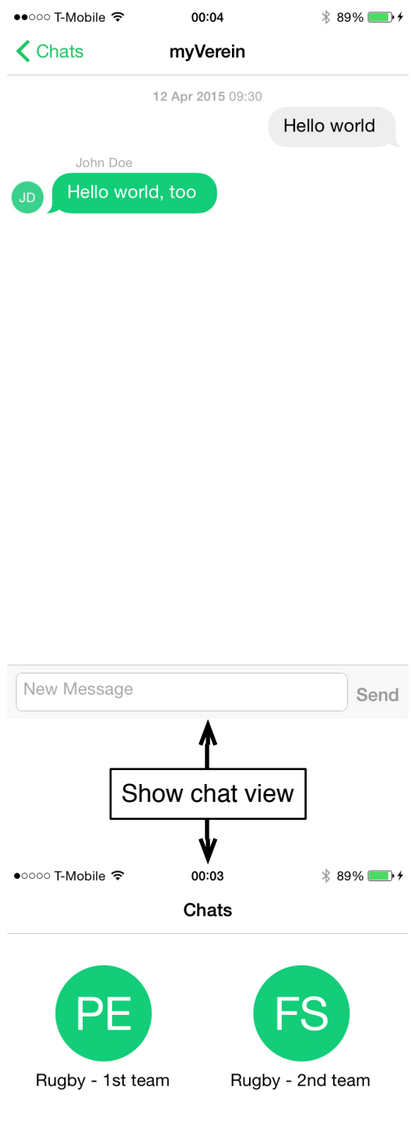

Start
High level input
(e.g. objects)
Parsing high level input
(MVNetworkingHelper)
Getting singleton instance
(MVNetworkingHelper)
Creating appropriate callbacks
(MVNetworkingHelper)
Callback
Creating MVRequest object
(MVNetworking)
Ignore count increase
Handle request
(MVNetworking)
Ignore count increase
No
Yes
Request started
(MVNetworking)
Acquire 'active request counter' lock
Increase 'active request counter'
(MVNetworking)
Release
'active request counter'
lock
Perform request
Request
successfull
Yes
No
Handle success
callback
Handle request with error
(MVNetworking)
Type of error
Else
Handle failure
callback
User not logged in (401)
Handle request after log in
(MVNetworking)
Release 'log in queue' lock
Acquire 'log in queue' lock
Is there already a log in
queue?
No
Create queue, put request on
queue & release lock
(MVNetworking)
Yes
Put request on queue &
release lock
(MVNetworking)
Request ended
(MVNetworking)
Acquire 'active request counter' lock
Decrease 'active request counter'
(MVNetworking)
Release 'active request counter' lock
End
Release 'log in queue' lock
For each item on the
request queue
Acquire
'log in queue'
lock
No
Request
successfull
Yes
Acquire
'log in queue'
lock
For each item on the
request queue
Release 'log in queue' lock
Perform log in
(MVNetworking)

```
//
// Copyright (C) 2015 Frank Steiler <frank@steilerdev.de>
//
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation, either version 2 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this program.  If not, see <http://www.gnu.org/licenses/>.
//

//
//  MarshalOperator.swift
//  This file contains the definition of the Marshal operator, that is
    intended to simplify multi threading using GCD, by providing a thin
    abstraction layer, while still hiding the underlying C API.
//
//  The marshal operator (~>) is an in-, pre- and postfix operator used on
    closures. The operator was first suggested by Josh Smith, whose initial
    implementation is the foundation for this extension to the Swift
    programming language: http://ijoshsmith.com/2014/07/05/custom-threading-
    operator-in-swift/.
//
//  The idea is that everything in front of the operator is executed on a
    serial background queue, while everything behind the operator is executed
    on the main queue. As an infix operator the background task is executed
    first and the task run on the main queue afterwards. There are variations
    that offer the passing of a variable from one closure to the other, as
    well as a background closure provided with a ManagedObjectContext usable
    by the background queue.
//
//  Another operator is the inverted marshal operator (<~), used postfix only.
    As soon as the closure is finished executing (and therefore hitting the
    operator) it's execution is repeated after 5 minutes. With the infix
    operator '<~/' the user is able to specify the amount of time in seconds
    between executions through the right hand value.
//

import Foundation
import CoreData
import UIKit

// Defining the operators
infix operator ~> {}
prefix operator ~> {}
postfix operator ~> {}

postfix operator <~ {}
infix operator <~/ {}

// Defining a typealias to hide the C API on return values
typealias MarshalThreadingObject = dispatch_source_t

// The queue used by the marshal operator.
```

```
private let marshalQueue: dispatch_queue_t =
    dispatch_queue_create("de.steilerdev.myverein.marshal-queue",
    DISPATCH_QUEUE_SERIAL)

/// Using the Marshal operator as a prefix operator, means that the closure is
    executed on the main queue.
///
/// :param: mainClosure The closure executed on the main queue.
prefix func ~> (mainClosure: () -> ()) {
  dispatch_async(dispatch_get_main_queue(), mainClosure)
}

/// Using the Marshal operator as a postfix operator, means that the closure
    is executed on a background queue.
///
/// :param: backgroundClosure The closure executed on a background thread.
postfix func ~> (backgroundClosure: () -> ()) {
  dispatch_async(marshalQueue, backgroundClosure)
}

// Using the Marshal operator as a postfix operator, means that the closure is
    executed on a background queue. This operator provides a managed object
    context created on the background queue and can therefore be used without
    any concurency problems. The core data objects (especially the persistent
    store coordinator) need to be handled through the app delegate.
///
/// :param: backgroundClosure The closure executed on a background thread
    providing a managed context usable on this thread.
postfix func ~> (backgroundClosure: (NSManagedObjectContext) -> ()) {
  dispatch_async(marshalQueue) {
    var backgroundContext = NSManagedObjectContext()
    backgroundContext.persistentStoreCoordinator = (UIApplication.
        sharedApplication().delegate as! AppDelegate).
        persistentStoreCoordinator!
    backgroundClosure(backgroundContext)
  }
}

/// Executes the left-hand closure on the background queue, upon completion
    the right-hand closure is executed on the main queue. This operator
    provides a managed object context created on the background queue and can
    therefore be used without any concurency problems.
///
/// :param: backgroundClosure The closure executed on a background thread,
    providing a managed context usable on this queue.
/// :param: mainClosure The closure executed on the main thread, after the
    background thread is finished.
func ~> (backgroundClosure: (NSManagedObjectContext) -> (), mainClosure: () ->
    ()) {
  dispatch_async(marshalQueue) {
    var backgroundContext = NSManagedObjectContext()
    backgroundContext.persistentStoreCoordinator = (UIApplication.
        sharedApplication().delegate as! AppDelegate).
        persistentStoreCoordinator!
    backgroundClosure(backgroundContext)
    dispatch_async(dispatch_get_main_queue(), mainClosure)
  }
}

/// Executes the left-hand closure on the background queue, upon completion
```

```
    the right-hand closure is executed on the main queue.
///
/// :param: backgroundClosure The closure executed on a background thread.
/// :param: mainClosure The closure executed on the main thread, after the
    background thread is finished.
func ~> (backgroundClosure: () -> (), mainClosure: () -> ()) {
  dispatch_async(marshalQueue) {
    backgroundClosure()
    dispatch_async(dispatch_get_main_queue(), mainClosure)
  }
}

/// Executes the left-hand closure on the background queue, upon completion
    the right-hand closure is executed on the main queue using the return
    value of the left-hand closure.
///
/// :param: backgroundClosure The closure executed on a background thread.
/// :param: mainClosure The closure executed on the main thread.
func ~> <R> (backgroundClosure: () -> (R), mainClosure: (R) -> ()) {
  dispatch_async(marshalQueue) {
    let result = backgroundClosure()
    dispatch_async(dispatch_get_main_queue()) {
      mainClosure(result)
    }
  }
}

/// Executes the provided closure every 2 minutes. The execution might be
    defered by up to 30 seconds. The closure is executed as long as the
    returned object is referenced by the developer. If you want to stop the
    execution you therefore have to set the variable holding this object to
    nil.
///
/// :param: repeatedClosure The function that should be repeatedly executed.
///
/// :returns: The threading object, that needs to be referenced as long as the
    execution should continue.
postfix func <~ (repeatedClosure: () -> ()) -> MarshalThreadingObject {
  return repeatedClosure<~/50.0
  // TODO: Change back to 120.0!!
}

/// Executes the provided closure every x seconds. The execution might be
    defered by up to 30 seconds. The closure is executed as long as the
    returned object is referenced by the developer. If you want to stop the
    execution you therefore have to set the variable holding this object to
    nil.
///
/// :param: repeatedClosure The function that should be repeatedly executed.
/// :param: intervalBetweenExecutionInSeconds The amount of time between
    executions in seconds.
///
/// :returns: The threading object, that needs to be referenced as long as the
    execution should continue.
func <~/(repeatedClosure: () -> (), intervalBetweenExecutionInSeconds: Double)
    -> MarshalThreadingObject {
  // Amount of time the system can defer the execution of the closure
  let deferedTime = UInt64(30 * Double(NSEC_PER_SEC))
  // Amount of time between executions
  let intervalTime = UInt64(intervalBetweenExecutionInSeconds * Double
```

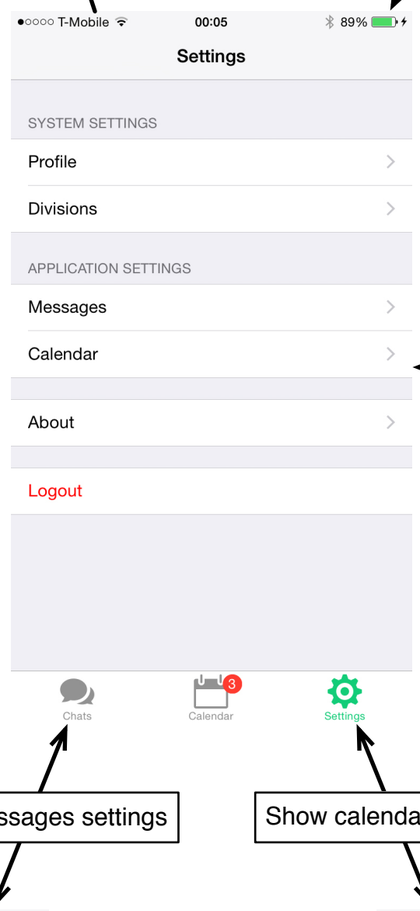
```
        (NSEC_PER_SEC))
    // Initial delay for first execution
    let startDelay: dispatch_time_t = dispatch_time(DISPATCH_TIME_NOW, 0)

    // Create a new timer on the marshal queue
    let timer: dispatch_source_t = dispatch_source_create
        (DISPATCH_SOURCE_TYPE_TIMER, 0, 0, marshalQueue)
    // Define the behaviour of the timer
    dispatch_source_set_timer(timer, startDelay, intervalTime, deferedTime)

    // Set the event handler (the closure that should be executed everytime the
        timer is firing)
    dispatch_source_set_event_handler(timer, repeatedClosure)

    // Starting the queue
    dispatch_resume(timer)
    return timer
}

/// Executes the right hand closure after waiting the amount of seconds
    defined by the left hand double. The closure is guaranteed to be executed
    on the main queue.
///
/// :param: waitInSeconds The amount of time, the execution gets delayed.
/// :param: mainClosure The closure that is going to be executed after the
    application waited the defined amount of time.
func ~>(waitInSeconds: Double, mainClosure: () -> ()) {
    dispatch_after(dispatch_time(DISPATCH_TIME_NOW, Int64(waitInSeconds *
        Double(NSEC_PER_SEC))), dispatch_get_main_queue()) {
        mainClosure()
    }
}
```

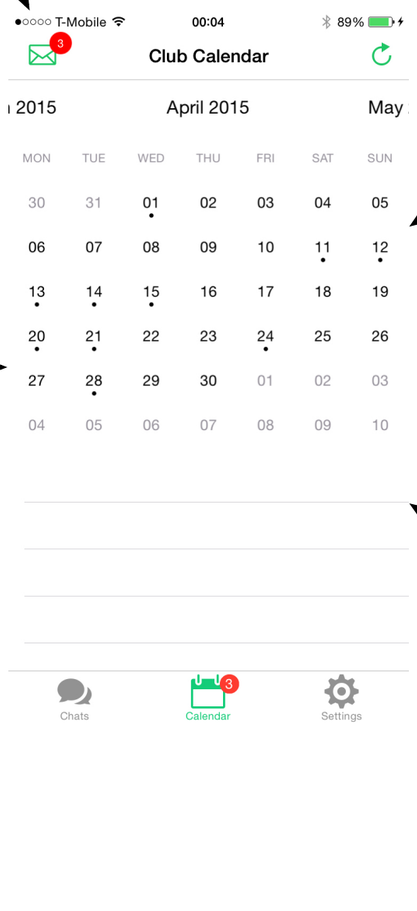


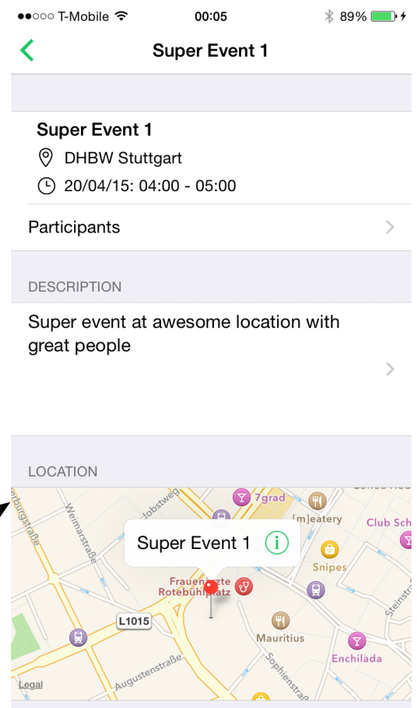
Show chat view

Login

Logout



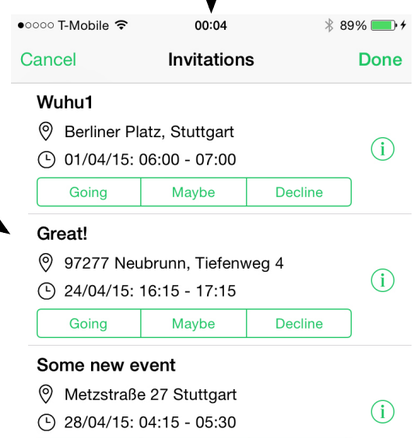




Show event

Show event details

Show open invitations



Show messages settings

Show calendar settings