Club Sportstiming

Application for premium members of Club Sportstiming

Project report

3rd semester

Alexander Nesheim, Casper K. Leth, Christopher C. Jensen, Dion S. Hansen, Mads C. B. Nielsen, Villiam Jacobsen

Group 314

Aalborg University

January 21, 2021

Number of words: XX

Number of words is without words on the front page, preface, contents, sources and bibliography.

AALBORG UNIVERSITET
STUDENTERRAPPORT

Second study year - Software
Selma Lagerløfs Vej 300
9220 Aalborg Øst

Title:

Club Sportstiming

Project:

P3 project

Project period:

September 2020 - December 2020

Projectgroup:

Sw314e20

Authors:

Alexander Nesheim
Casper Kjærhus Leth
Christopher Colberg Jensen
Dion Sean Hansen
Mads Christian Bruun Nielsen
Villiam Fredrik Jacobsen

Supervisor:

Andrej Kiviriga

Number of pages: 80
Finished: January 21, 2021

Abstract

The report revolves around solving a problem for a company. The company chosen is Sportstiming who wanted an application accessible to it's subscription server, Club Sportstiming. The app encompasses a chatroom, blogs, Q&A, polls, and training/diet programs. The requirements set by Sportstiming were prioritised using the MoSCoW method and hereafter presented in a table. The analysis of the report primarily consists of a problem domain analysis and an application domain analysis, which both support the building of a software solution by organising the procedure. The app contained all essential requirements and a few non-essential. This said there is a number of things that should have been done differently, but overall the project and delivery of the app is considered a success.

Preface

This project was written by group SW314 as part of the P3 Project during our 3rd semester of the B.Sc. in Software at Aalborg University.

The purpose of this project is to gain experience with object oriented analysis, design and programming, as well as the evaluation of user interfaces. Furthermore, the goal is to create a well structured application, the requirements of which originate from a collaborative partner. The courses will be utilised throughout the report to support the project. An additional requirement for the P3 project, is for the application to be written in C#.

We would like to thank our supervisor, Andrej Kiviriga, for his contribution to the project, and our contacts at Sportstiming.

Reading instructions

The report uses the vancouver method, therefore citations will be presented by [<Number>]. If specific pages in a book is referenced, the range of pages will be included. Elements written in *italic* is used when referencing a specific object presented in the report. The `typewriter` font is used for code specific elements.

Contents

Part I

Introduction

1 | Introduction

This report details a project made in collaboration with the company Sportstiming. The goal of the report is to create an application for Sportstiming's subscription service, called Club Sportstiming, that aims to reduce manual work, and improve the customer experience. The following sections will present Sportstiming, the challenges the project sets out to solve, a FACTOR criterion which will be the basis of the analysis, and a description of the strategy used to solve the problem.

1.1 About Sportstiming

Sportstiming is one of Europe's leading providers in timekeeping and registration for sport events, such as running, cycling, swimming and triathlon. Each year they handle several hundred thousands of participants [1].

Club Sportstiming is the latest addition to the services provided by Sportstiming. It is a subscription based service, where subscribers have direct contact with instructors and access to benefits. Instructors are licensed personal trainers that are employed by Sportstiming. Their job is to assist Club Sportstiming subscribers with their training through diet- and training-programs and workshops. Additionally, instructors also answer training related questions from subscribers and provide event specific training programs.

From here on, Club Sportstiming subscribers will be referred to as members. The distinction between instructors and members is that of employees and customers, but the two groups will collectively be referred to as users, since the system will be used by both groups, and the differing factor is their privileges.

Currently, Sportstiming facilitates the service through a private Facebook group, which includes workshop events, polls for member interest and more.

1.1.1 Challenges

Facilitating the service through Facebook means that Sportstiming have access to a lot of functionalities, but this comes at the cost of not being integrated with their existing system. The subscription service is managed via accounts on their website, therefore any changes to the accounts does not automatically propagate to their Facebook group, which leads to a lack of efficiency and tedious tasks for their employees. An example could be a member cancelling their subscription. This decision to cancel does not automatically exclude or otherwise remove them from the Facebook group, as the two systems are not connected. Instead an employee has to keep tabs on the membership list, and manually remove them from the Facebook group if a subscription is cancelled.

Currently Sportstiming uses posts to inform members about everything related to Club Sportstiming. This includes news relating to workshops, blog posts and so on. The individual posts are segmented into different topics to maintain the readability, but having news segmented across multiple posts could make it hard to be up-to-date for members. Furthermore, the inability to develop new features or even change existing ones combined with the inability to integrate it with their existing services such as blogs, programs, and most importantly their user system means that Sportstiming has an interest in developing their own solution.

The problem therefore resides in Sportstiming wanting to keep the functionality that comes with the Facebook group such as polls, posts and other tools, while at the same time automating the process of managing the members, by integrating it with their existing user system.

In summation, because of the mentioned limitations, Sportstiming has the desire of developing a new application, which has the desired functionalities of the Facebook group, while making it possible to add new features and integrate it with the services provided by their website.

1.1.2 Data access

As the project will handle delegate data such as email and password, a secure way of accessing such data is needed, therefore a web-service was made available by Sportstiming. Other than sensitive data, the web-service will be used to access data associated with features such as blogs and training programs.

1.2 Creating System Definition

Before analysing and developing an application, it is important to determine the functionalities and responsibilities of the system, also called a system definition. This is used to assert that the understanding of Sportstiming's challenges and ambitions for the application are correct. The system definition will also serve to direct the analysis and by extension the development, of what will eventually become the application.

There are numerous tools to aid in the creation of a system definition. One of them is the FACTOR criterion, which will be used to develop the system definition and verify it.

1.2.1 FACTOR

The FACTOR criterion will be made according to the definition derived from Object Oriented Analysis & Design [2]. The purpose of utilising the FACTOR criterion, is to complement a system definition which is accomplished trough an analysis of the criterion. This is done to ensure a well-defined system definition and furthermore formulate and verify the correctness of the system definition.

From the understanding gained from meetings with Sportstiming, the following FACTOR criterion was formulated:

Functionality: The system has to allow users to verify themselves by logging in. There has to be a chatroom, where users can message each other. The blog on the website should be integrated into the system. Users consist of instructors and members, which have different permissions. Instructors can create polls, and members can vote and create additional choices on the poll. Instructors can also create workshops within the system, and the members can register for these workshops. There has to be a Q&A forum, where members can ask questions, and only the instructors have permission to answer the questions. The various training and diet programs on the website should be available in the system. Lastly the system should be able to monitor the new content and changes for the purpose of notifying users, and displaying it.

Application domain: Instructors, Members, Web service (user login)

Conditions: Users have varying knowledge and IT-experience. Developed by students. C# based.

Technology: Written with C# using Xamarin, should run on Android and iOS

Objects: Question, Answer, Chatroom, Message, Blog post, Notification, Training program, Diet, Workshop, Poll, Member, Instructor

Responsibility: The system must provide a platform for Sportstiming's products, communication between users, as well as support the instructors in assisting the members.

1.2.2 System Definition

Using the FACTOR criterion the following system definition was created:

A mobile application used to consolidate the Club Sportstiming's content onto a single application. The application should provide the user with access to; a Q&A forum, a chatroom, training programs, diet programs, a blog, polls, a notification and workshops. The users are required to login before accessing any content within the application. Users consist of two distinct types, members and instructors. instructors have elevated permissions and can create/delete/change content within the application such as polls, workshops etc., whereas members can only interact with the polls, workshops etc.

1.3 Process Description

The report will be structured according to the waterfall method, whereas the work process will be iterative. An iterative approach was chosen because the project is a collaboration between the group and a company. Hence the requirements are elicited from interviews with the company and confirmed either by interpretation or discussion, but these requirements often change over time, as the parts of the project become more solidified and defined.

Following a strict waterfall approach would therefore mean that these possible improvements or misunderstandings would not be possible to implement or correct, which is undesirable if one's interest is the best possible end product. This is not to say a waterfall approach would be without merit, as this also pushes development forward by having to finish tasks before starting new ones, so one could refrain from being stuck in finished tasks, and work from a solid foundation. In contrast the iterative approach makes it possible to work on several tasks throughout the project, which could lead to tasks not being finished, but in this project it is seen as a negligible downside compared to the flexibility it provides.

The reason why the report will be structured according to the waterfall method, is because it provides a better overview of the process from analysis to conception and implementation. It also works to consolidate particular parts of the process to specific parts of the report. Furthermore the group has experience with the waterfall structure, but none for the iterative. This however presents a dilemma; should we experiment with a new structure? To determine this, a list of pros and cons for both structures were made, which ended up supporting the waterfall structure.

Delving into the work process, it will consist of multiple sprints, each sprint has a specific goal, an example would be developing on a specific subsystem while at the same time, documenting it in the report. Another aspect is the flexibility provided by being able to develop, document, test, and lastly evaluate subsystems in a single sprint. The sprints themselves will be confined to one week or two week intervals, according to the work load. The workload is determined at the start of each sprint and will consist of the previously mentioned goal, but it may also be combined with the urgent tasks. Being able to tackle urgent tasks on a sprint by sprint basis makes it possible to always allocate resources to the most urgent tasks of the project, and not be confined to a set schedule or sequence of tasks.

Part II

Analysis

2 | Requirements

During meetings with Sportstiming, a number of requirements were elicited, rephrased, and removed. It was an iterative process, as details in the project came to light and needed to be addressed and defined. As a result, requirements were changed and reevaluated a number of times during discussions with Sportstiming. Therefore it became important to structure the requirements, as it would greatly ease that process.

The requirements were structured using the MoSCoW method, which was chosen for its clear prioritising of requirements and timeboxing element [3].

The MoSCoW method draws on the reality that one can rarely fulfill all requirements in a constrained time period, which is the nature of this project. It therefore forces one to prioritise certain requirements over others. This means that requirements are put into one of four categories: *Must have*, *Should have*, *Could have* and *Won't have* [3].

Must have requirements are, as the name leads on, critical for the expected delivery of a system to be considered successful. If even one *Must have* requirement is unfulfilled, the delivery should be considered a failure. On the other hand *Won't have* indicates that the requirements will not be a part of this delivery [3].

Each requirement within the MoSCoW categories is ordered after priority. Meaning that the requirements at the top of the *Must have* category should be developed before the later requirements in the same category. This is not typical for the MoSCoW method, but it was determined to be helpful to further prioritise the requirements.

As mentioned earlier, the process of changing and reevaluating the MoSCoW list, involves presenting it to Sportstiming and discussing the requirements and their prioritisation. During one particular discussion, changes were made based on the difficulty of the requirement and prioritisation within a category.

The requirement *Diet program*, was originally placed in the *Could have* category, but was changed because the nature and difficulty of the requirement closely resembled that of *Training program*, which was situated in the *Must have* category.

MoSCoW
Must have
The system must contain one chatroom for all users
All users can send and receive messages
All users can see and download all training programs
All users can see and download all diet programs
Instructors can create polls
Members can vote on polls
The system must retrieve all blog posts from the Sportstiming website
Only members can ask questions
Instructors can answer questions
Users can see all public questions and answers
The notification page automatically displays all in-app updates
Should have
Any user can tag another user in the chatroom
A tagged user receives a notification
Instructors can delete messages
Instructors can pin messages
All users can add a poll choice to an existing poll
A question can only be commented on by instructors and the member asking the question
Could have
Instructors are recognizable by a badge
Instructors can create chat rooms
A pinned message can have a timer
Poll choices consist of the events being held at least a month in advance
A member can ask a private question, which cannot be seen by other members
Workshops can be created by instructors
Workshops have a set cap of attendees
Workshops have a location, date and time
Members can sign-up for workshops
Won't have
Members can create chat rooms
A section for specific event training programs should be implemented
Users can view videos of an exercise in the app
Users can comment on blog posts

Table 2.1: MoSCoW list

3 | Problem Domain Analysis

This chapter will focus on the problem domain. The problem domain concerns the objects the system has to monitor, administrate and/or control [2,p. 47]. The result of the problem domain analysis will be a model of the system, as the users of the system would see it. The model is structured using class diagrams and behavioural patterns in the form of state chart diagrams. Initially, we will examine class candidates by observing the objects related to the problem domain.

3.1 Classes

Classes are a way to categorise objects of the problem domain, that share structure, behavior, and attributes. This abstraction, of the problem domain objects into classes, supports the development of the problem domain model, as future users would see it. Candidates for potential classes, within the problem domain, were generated through brainstorms and in cooperation with the customer. The class candidates can be seen in table 3.1.

Message	Chatroom	Question	~~Answer~~
Poll	~~PDF~~	Member	Instructor
~~Notification~~	Q&A Thread	~~Notification Page~~	~~Notification Post~~
~~Workshop~~	~~Pinned~~	Blog Post	Training Program
Comment	~~Poll Vote~~	Poll Choice	Diet Program
~~Workshop Sign-Up~~			

Table 3.1: Class Candidates for Sportstiming system. Strike-through candidates are discarded

The class candidates were then evaluated upon, initially by examining whether the individual class candidates were a part of the problem domain. Examining whether class

candidates are part of the problem domain, involves discussing if the object in question is administered, controlled or monitored within the target system, if that is not the case, then the candidate was removed.

Furthermore, the class candidates are not a complete and final list of classes for the system. A class would be split into multiple classes, if the class encapsulated multiple related but different objects or functionalities.

***Member & Instructor*:**
The main difference between *Members* and *Instructors* is the level of authorization, and an *Instructor* should be able to create polls, whereas a *Member* can only vote on polls. Both *Member* and *Instructor* share attributes such as name, username, email etc., which is the reason why the *User* class was added to the class diagram, seen in section 3.3 on figure 3.1.

The *User* class is a generalization class from which the member and instructor class are specialisations. All users are either a member or an instructor.

***Comment & Answer*:**
A *Comment* is a comment or answer related to a particular *Question* within a *Q&A thread*. A *Comment* and an *Answer* object are fundamentally the same, and as such the *Answer* class has been excluded to remove redundancy.

***Chatroom & Message*:**
To administrate the chat functionalities desired by Sportstiming, the *Chatroom* class acts as a container for instances of the *Message* class within a particular *Chatroom*. Furthermore, the *Chatroom* class enables users to not only send, but also receive messages.

***Training Program, Diet program, Program & PDF*:**
The *Training* and *Diet program* classes encapsulate resources available as PDF files for download on Sportstiming's website. The programs are required to be accessible on the application, and as such have been modeled as *Training Program* and *Diet Program* classes, sharing attributes and methods derived from the *Program* class, as seen on the class diagram in section 3.3 figure 3.1. The *File* and *PDF* classes are redundant as of now, due to being part of the *Program* class rather than separate classes. It is not desired to manage the PDF files in the application, but rather to link to the resources themselves from within the application.

***Poll, Poll Choice & Poll Vote*:**
A poll consists of a number of poll choices, which the users can cast votes on. A poll,

complete with *Poll* and *Poll Choices*, encapsulate and facilitate the functionalities related to serving and managing a poll. A *Poll* is only to be created by *Instructors*, while both *Instructors* and *Members* should be able to cast votes for multiple *Poll choices* within a particular *Poll*. The *Poll vote* class has been omitted, as the class was found to be redundant, since it does not store any information. The information retained in a possible *Poll vote* class is simply the name and id of a *User*, which is already contained in the *User* class itself. The relation has instead been modeled directly between the *User* and *Poll Choice* classes, which can be seen on the class diagram in section 3.3 figure 3.1.

***Notification Post*, *Notification Page* & *Notification*:**
The *Notification Post* and *Notification Page* classes have been omitted due to being redundant, as the objects of the classes are not a part of the problem domain. Additionally, the *Notification* class does not store any information, and it has no instances as a notification is an event rather than an object.

***Workshop* & *Workshop Sign-Up*:**
The *Workshop* and *Workshop Sign-Up* classes has been omitted due to prioritization. In the MoSCoW list, seen in section 2 on table 2, Workshop is situated in the *Could have* category, hence they have been excluded from the problem domain.

***Pinned*:**
The *Pinned* class was meant to encapsulate a pinned message, however, there is no need for a *Pinned* class to model this interaction, as a pinned message is simply a *Message* object in a particular state. As such the *Pinned* class have been left out.

The excluded classes are not considered further, and will not be present in the event table or the class diagram presented in section 3.2.1 and 3.3 respectively.

3.2 Events

Events are used to characterize the classes and provide an understanding of the dynamic interactions between the events and classes in the problem domain model. To avoid ambiguous and vague events, an event is evaluated according to three criteria [2,p. 65]:

- Is the event instantaneous[1]?
- Is the event atomic[2]?
- Can the event be identified when it occurs?

[1] An event should be instantaneous in order to avoid ambiguity of when the event occurs

[2] An atomic event is the smallest, traceable event which is to be monitored in the system. As such events should be broken down until no other sub-events are found

The event candidates, are the result of brainstorming the possible events associated with the classes chosen in section 3.1.

~~Blog post Submitted~~	~~Blog post Deleted~~	~~Member Subscribed~~
Blog post Retrieved	Vote Retracted	Message Pinned
Message Unpinned	Comment Added	Comment Removed
Chatroom Created	Chatroom Deleted	Diet Program Retrieved
~~Member Created~~	~~Instructor Created~~	~~Instructor Deleted~~
~~Member Deleted~~	~~User Created~~	~~User Deleted~~
Comment Deleted	User Retrieved	~~User Removed~~
Q&A Thread Answered	Q&A Thread Closed	Comment Marked as Answer
~~User Authenticated~~	Question Asked	Question Answered
Message Sent	Message Received	Message Deleted
Poll Created	Vote Cast	Poll Choice Added
Poll Choice Removed	Poll Deleted	~~Notification Sent~~
Q&A Thread created	Q&A Thread deleted	Training Program Retrieved
~~Notification Received~~	~~Fetch Training Program~~	~~Sign-up for Workshop~~
~~Deregister for Workshop~~	~~User Unauthorized~~	User tagged
Comment rejected as Answer		

Table 3.2: Event Candidates for Sportstiming system

The event candidates were evaluated to identify which candidates to include or exclude. In the event candidates table 3.2 a ~~strike-through~~ entry corresponds to an excluded event, while any event that is not stroked-out are included.

Looking at the event candidate table 3.2, it is seen that *Blog post Submitted* and *Blog post Deleted* have been omitted, this is due to not being a part of the problem domain, as the creation and deletion of blog posts is not part of the problem domain, but instead administrated by Sportstiming. On the other hand, the *Blog post retrieved* event is included as it is an instantaneous, atomic event, that can easily be identified when it occur, and as such it does not violate any of the evaluation criteria. Similar reasoning applies to the events *Diet Program retrieved* and *Training Program retrieved.* Furthermore, *Instructor-* and *Member- Created* and *Deleted* have been omitted, as this is not part of the problem domain, although the events themselves are instantaneous, atomic and can be identified when they occur.

Any event that is deemed outside the problem domain or violating any of the criteria, have been omitted.

3.2.1 Event table

An event table supports the understanding and structure of the event and class relations, as events characterize classes by providing an understanding of the dynamics of the problem domain. The event table supports the process of evaluating the classes further, especially to examine which classes share relations, as well as identify missing classes.

An event table relates an event to a class by a quantifier. The quantifiers used are the multiplication sign *, and the addition sign +. The * symbol denotes a relation of zero or more occurrences of the event for the class object, while + denotes a relation of exactly one occurrence of the event for the class object.

The event table seen on figure 3.3, also shows a difference between the *instructor* and *member* classes, and how their functionality differ from each other. For example, where the instructor can delete and pin messages, the member can only send messages.

Furthermore, the information contained within the event table is used further in the analysis, to identify functions for the model 5.1.1 and function component 5.1.2, as well as the level of coupling and cohesion of the system in architecture 5.3.

	Chat-room	Mes-sage	Q&A-Thread	Ques-tion	Com-ment	Poll	Poll-Choice	Instruc-tor	Mem-ber	Blog-post	Train-ing Pro-gram	Diet-Pro-gram
Chatroom created	+							*				
Chatroom deleted	+							*				
Message sent	*	+						*	*			
Message received	*	+						*	*			
Message deleted	*	+						*				
Message pinned	*	*						*				
Message unpinned	*	*						*				
Question asked			+	+					*			
Q&A Thread deleted			+					*				
Q&A Thread Answered			+									
Q&A Thread Closed			+									
Question answered			+	+				*				
Question deleted			+	+				*				
Comment added			*		+			*	*			
Comment deleted			*		+			*				
Comment removed			*		+			*	*			
Comment Marked as Answer			*		*			*				
Comment rejected as Answer			*		*				*			
Poll created						+		+				
Poll deleted						+		+				
Poll Choice added						*	+	*	*			
Poll Choice removed						*	+	*	*			
Vote cast							*	+	+			
Vote retracted							*	+	+			

Table 3.3: Event table, the events are on the left and classes on the top.

3.3 Structure

The core structure of the system is established by the problem domain model, and corresponds to the class diagram on figure 3.1. The class diagram draws on the information of the relation between classes and events contained in the event table, that was presented in section 3.2.1 figure 3.3. The purpose of the class diagram is to present relations between classes and provide an overview of the system structure.

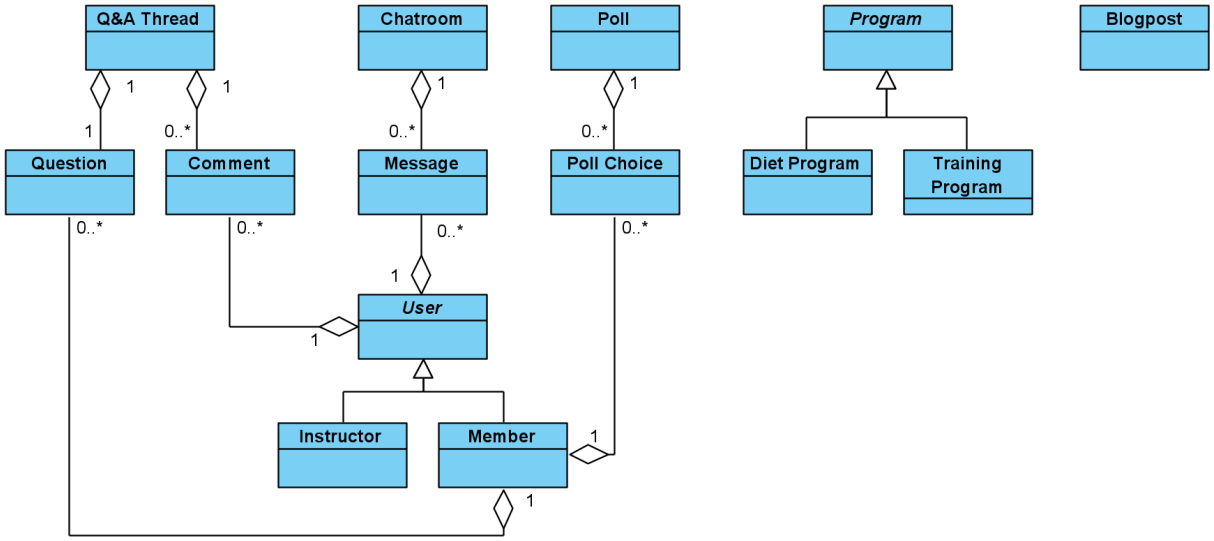


Figure 3.1: Class diagram for the system

The class diagram shows how different parts of the system, such as *Q&A*, *Poll* and *Chatroom*, have no relation to each other beside the *User*. This detail is vital for understanding the structure of the system, as the low amount of relations between parts enables the developing of functionalities individually, since there are little to no dependencies. Furthermore, the generalization classes[3], *User* and *Program*, have been introduced to the class diagram even though the classes are not part of the class candidates presented in section 3.1. This is because the *Instructor* and *Member* classes share properties, and the reasoning remains the same for the *Diet Program* and *Training Program* classes.

In section 3.1, it was mentioned that the *Answer* class had been omitted, and the reason was because an *Answer* is merely a specific type of *Comment*.

As seen in section 3.2.1 on the event table 3.3, the *Instructor* and *Member* classes have relations to almost all of the events. Furthermore, the aggregations between *Comment*

[3]A generalization class is a class that encapsulate shared properties of specialized classes[2,p. 74-76]

and *Message* into *User* is meant as aggregations into both of the specializations of the *User* class. This was done to reduce complexity of the class diagram.

Note that the *Instructor* class does not have any direct relations apart from the specialization from *User*, due to the system not monitoring or administrating the creation of a poll or chatroom.

Additionally, the generalization class *Program* and its specializations, *Diet Program* and *Training Program*, as well as the *Blog post* class are completely isolated from the rest of the class diagram. The reason being that these parts are a part of the problem domain, however there is no direct relation to the *User* or other parts of the system. This is due to the fact, that the system does not monitor when a user reads a blog post or opens a training program. These functionalities are more of a consideration in the application domain. The programs (training and diet) as well as blog posts are retrieved by utilizing a web-service made available by Sportstiming. The web-service is detailed further in section 7.2.3.

3.3.1 Clusters

As mentioned in section 3.3, the lack of dependencies between the different classes, as demonstrated by the class diagram, enable the simultaneous development of independent functionalities, and based on that there will be an introduction of clusters. The clusters represent the collection of classes, which are dependent on each other, which can also be thought of as complete functionality formed from the classes.

The system has six clusters: **Q&A**, **Chatroom**, **Polls**, **User**, **Programs** and **Blog post**.

The **Blog post** cluster simply contain the *Blog post* class, while the **Programs** cluster contain the *Program* class as well as its specializations. Similarly, the **Users** cluster contain the *User* class and its specializations.

The **Blog post** and **Programs** cluster only contain a single class, and the reason is because the blogs and programs are handled by Sportstiming, and so the single classes can still be thought of as a complete functionality and thereby a cluster.

The **Chatroom** cluster contains the aggregation of *Message* into *Chatroom*, while the **Polls** cluster contains the aggregation of *Poll Choice* into *Poll*. Finally, the **Q&A** cluster contains the aggregation of *Question* and *Comment* into *Q&A Thread*.

3.4 Behavior

The classes within the problem domain can either be dynamic or static, with static meaning that they do not change, and dynamic classes meaning they do, and this is modelled using state chart diagrams.

State chart diagrams depict the behavioral pattern of a class, where behavioral patterns refers to the possible changes to the state of a class. A state chart diagram illustrates the lifetime of class objects in the system. An object is created or instantiated, completes one or more state changes and finally dies. The state changes are presented as events with parameters, where the parameters correspond to the properties the system has to monitor or remember from the state change. As such only relevant properties should be included.

3.4.1 Poll

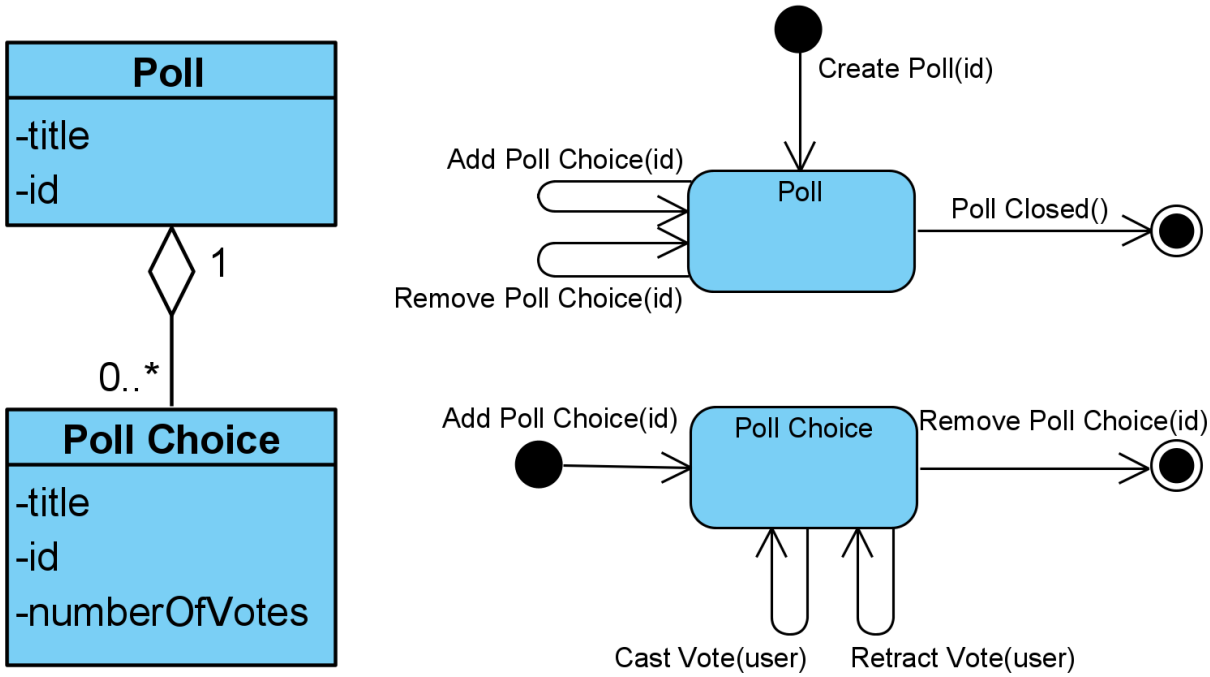


Figure 3.2: State chart diagram for poll.

The state chart diagram on figure 3.2, depicts the behavior of the **Polls** cluster within the problem domain model. A poll is only ever in one state, however it is possible to add *Poll Choice* objects to a *Poll* instance. The *CastVote* event is particularly interesting as one might be inclined to think that the event is related to a *Poll* instance rather than a *Poll Choice* instance. However, the problem domain model is modelled such that a *Poll*

consists of multiple *Poll Choice*, each of which track their individual amount of votes. This distinction ensure that the model resembles real-world polling, as a poll-respondent does not vote for a poll, but rather for a particular choice or option on the poll.

3.4.2 Q&A

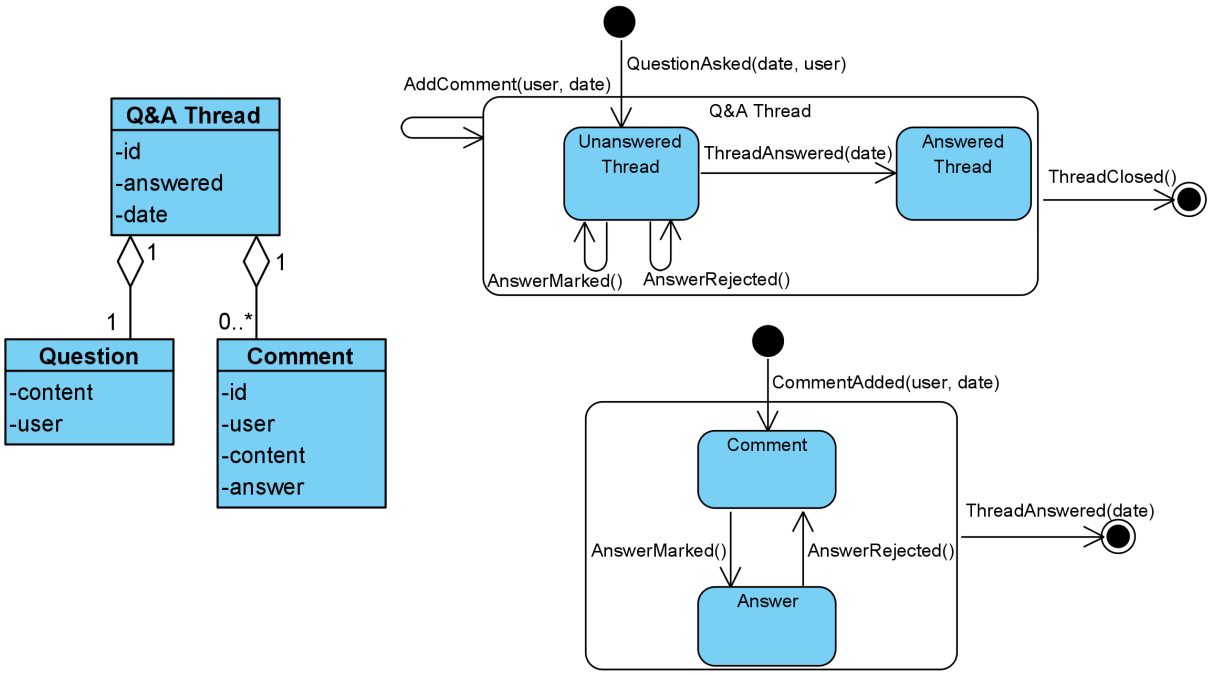


Figure 3.3: State chart diagram for Q&A.

The state chart diagram on figure 3.3, depicts the behavior of the **Q&A** cluster within the problem domain model. A *Q&A Thread* will during its lifetime go from being in the *Unanswered Thread* state to the *Answered Thread* state, before finally ending when the *Q&A Thread* is closed. It is important to note the events related to the answering of the question. In particular, notice that a *Comment* instance can change state between the *comment* and *answer* states by the *AnswerMarked* and *AnswerRejected* events. This behaviour is based on the decision to model the answer, to a question, as a *Comment* instance in a specific state. A *Q&A Thread* is answered when the *ThreadAnswered* event has occurred for a *Comment* instance on a *Q&A Thread* in the *Unanswered Thread* state. As such a *Q&A Thread* can, during its life time have multiple *AnswerMarked* and *AnswerRejected* events occur, corresponding to an instructor commenting on the thread with an answer, the instructor then mark the comment as an answer, if the member rejects the given answer, the thread continues to be in the *Unanswered Thread* state. If the member deems the given answer as satisfactory, the *ThreadAnswered* event will fire,

resulting in the thread going from *Unanswered Thread* to *Answered Thread* state. Finally, an *Instructor* can publish the *Q&A Thread* which fire the *ThreadClosed* event.

3.4.3 Chatroom

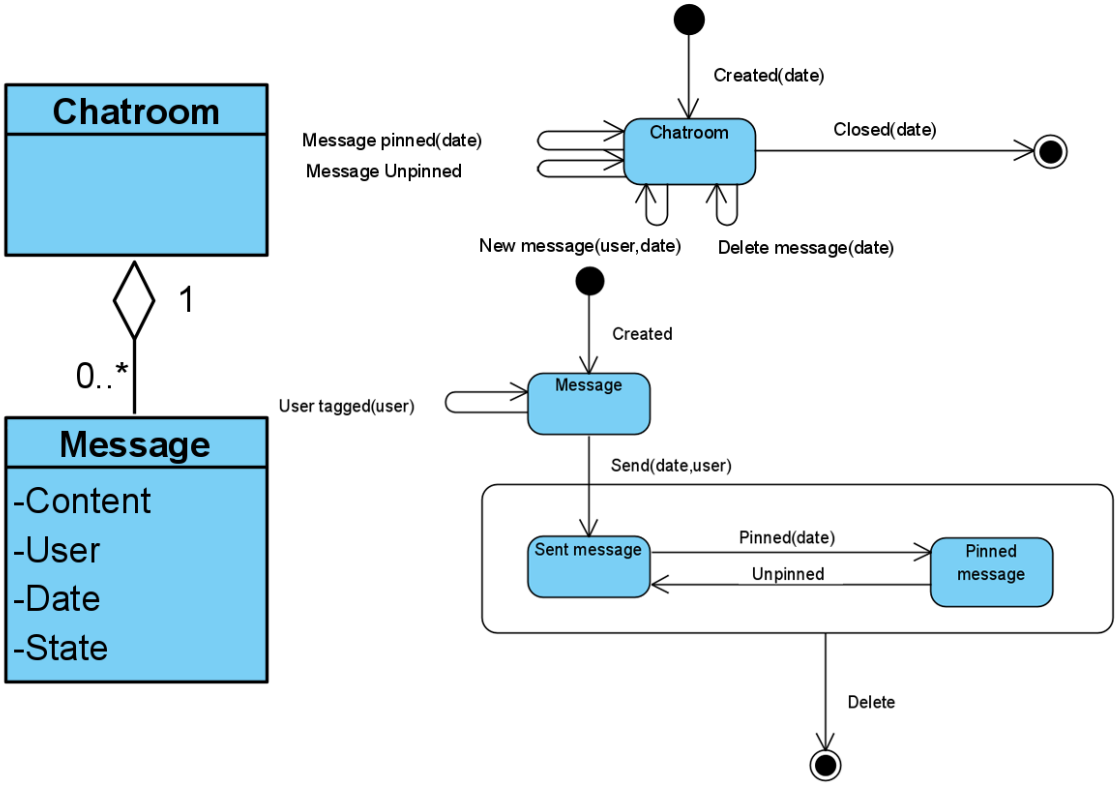


Figure 3.4: State chart diagram Chatroom.

The state chart diagram on figure 3.4, depicts the behavior of the **Chatroom** cluster within the problem domain model. The chatroom class acts as a container of messages, that can be deleted, created, added to and taken from, as shown in the upper state chart of figure 3.4.

The *Chatroom* class is only ever in one state, whereas the *Message* class has more, because of the pin and tag functionality, as shown in the lower state chart of figure 3.4. *Message* class can be sent, pinned, and unpinned, thereby the class has three possible states before the *Delete* event. When the *Send* event fires the *Message* class changes into the *Sent message* state, where only an instructor can change its state. When an instructor pins a message the *Pinned* event fires and changes the message state to the *Pinned message* state. When an instructor Unpins the *Unpinned* event fires and changes the event to the *Sent Message* state.

4 | Application Domain Analysis

This chapter will focus on the application domain. The application domain analysis focuses on how the target system will be used. As described in OOA&D, "the purpose in the purpose in analysing the use of a system, is to define requirements for the system's functions and interactions" [2,p.117].

4.1 Use Cases

Use cases are used as a specification, in natural language, of functionalities for a particular cluster of the system. Use cases guide the system development by specifying users' usage patterns in the system. As such use cases help define the application domain and act as cases, which by evaluation and analysis derive more detailed information related to functions and behavior within the system. Taking the users' interaction with the system into account using use cases also ensures a better understanding of the application domain as the users see it, ensuring a better system.

In this section some of the more interesting use cases related to the different clusters of the system will be presented, and some of the less interesting use cases have been omitted from the report. They can be found in the appendix C.1.

4.1.1 Chatroom

The chatroom cluster encapsulates messaging between users. Users are regarded as actors of the chatroom, corresponding to both Member and Instructor objects. The use cases for the chatroom, as well as their relation to the actors, are presented below in the actor table.

	Actors	
Use Cases	Members	Instructors
Open Chatroom	✓	✓
Send Message	✓	✓
Pin Message		✓
Unpin Message		✓
Open Pinned Messages	✓	✓
Delete Message		✓

Table 4.1: Actor table for chatroom

As evident from table 4.1, instructor objects have more use cases than member objects. This is due to their elevated authorization, as mentioned in section 3.1.

Delving into the use cases below, grants a more detailed definition of the functionalities and objects related to a particular use case for the system, by describing the use case scenario and clarifying which functions and objects are related.

Send Message
Use Case: Send Message is initiated by a user. After opening the chatroom the user can click the message input-field, type a text message and press the 'Send' button to send the message. **Object(s):** Instructor, Member, Message, Chatroom **Function(s):** SendMessage

Table 4.2: Send Message use case description

Pin Message
Use Case: Pin Message is initiated by an instructor. After opening the chatroom, an instructor can press/select a message and choose to pin the message by pressing the corresponding button in the context menu. Pinned messages will appear in the 'Pinned Messages' tab of the chatroom. A pinned message can be unpinned by selecting the corresponding message, and selecting the 'Unpin' option from the context-menu. **Object(s):** Instructor, Message, Chatroom **Function(s):** Pin Message, Unpin Message

Table 4.3: Pin Message use case description

Unpin Message
Use Case: Unpin Message is initiated by an instructor. After opening the chatroom, an instructor can press/select a pinned message, either in the chatroom itself or in the 'Pinned Messages' tab and choose to unpin the message. **Object(s)**: Instructor, Message, Chatroom **Function(s):** Unpin Message

Table 4.4: Unpin Message use case description

The use cases have now been presented and functions have been determined. The use-case functions will be presented below in figure 4.5, and given a complexity and function-type in line with OOA&D chapter 7 [2].

Chatroom		
Use case	**Complexity**	**Type**
Send message	simple	Update
Pin message	simple	Update
Unpin message	simple	Update

Table 4.5: Overview of use case complexity and type

As is evident in table 4.5, all functions relating to the chatroom use cases are categorized as simple, and are of the update function type and therefore will not be discussed further.

4.1.2 Poll

The **Polls** cluster will be expanded upon in this section, particularly in relation to the application domain. As such the actor table below provides an overview of the use cases as well as the actor's relation to each of the use cases. Only key use cases and functions will be expanded upon.

The actors correspond to the users of the application, which is both members and instructors.

	Actors	
Use Cases	Members	Instructors
Create Poll		✓
Delete Poll		✓
Cast Vote	✓	✓
Add Choice	✓	✓

Table 4.6: Actor table for Poll cluster

The creation of a poll is only available to the instructors, as this is only to be carried out by the instructors. However, both members and instructors should be able to cast votes and add choices to existing polls.

The use cases are presented below, and will be expanded further in regards to the complexity and type of functions.

Create Poll
Use Case: Create Poll is initiated by an instructor, which enters all the data needed for a Poll creation. This includes a title of the poll as well as zero or more poll choices. Additionally, the data is converted to a poll, and finally the poll is propagated to the application at which point the users can cast votes. **Object(s):** Instructor, Poll, Poll Choice **Function(s):** GetData, VerifyData, InitializePoll, UpdatePage

Table 4.7: Create Poll use case description

Delete Poll
Use Case: Delete Poll is initiated by an Instructor. An Instructor can delete any poll. No lingering data is kept after deletion. **Object(s):** Instructor, Poll **Function(s):** DeletePoll

Table 4.8: Delete Poll use case description

Cast and Retract Vote
Use Case: Cast Vote is initiated by a user selecting a particular poll and ticking off which of the poll choices they want to vote for. Additionally, a user can retract their vote by selecting a poll they previously voted on, and un-ticking the vote field for the particular poll choice they want to retract their vote from. Both casting and retracting a vote should update the poll such that users of the application can see the changes in votes. **Object(s):** Member, Instructor, Poll, Poll Choices **Function(s):** SelectPoll, CastVote, RetractVote, UpdatePoll

Table 4.9: Cast Vote use case description

Add Choice
Use Case: A user initiate Add Choice by selecting a particular poll and pressing the Add Choice button. The addition of a choice to a poll shall be propagated such that any user of the application can see this change. **Object(s):** Member, Instructor, Poll, Poll Choice **Function(s):** AddChoice, UpdatePoll

Table 4.10: Add Choice use case description

Each of the use cases have been verified by meetings with the customer. Additionally, the complexity varies between the use cases and their functions, hence the table below provide an overview of the complexity and type of function they exhibit.

	Poll	
Use case	**Complexity**	**Type**
Create Poll	Complex	Update
Cast Vote	Medium	Update
Add Choice	Simple	Update

Table 4.11: Overview of use case complexity and type

As can be seen in the table 4.11 the Create Poll use case is a complex update function, Cast Vote is a medium complex update function and Add Choice is a simple update function. Create Poll is deemed complex due to the multiple function calls and objects related to creating a poll, as well as propagating a created poll to the application. As such, further specification of Create Poll is presented below to clarify the functionalities related to the use case. The medium and simple functions are not deemed complex enough to warrant further specification at this time.

	Poll	
Use case	**Complexity**	**Type**
Create Poll	Complex	Update
- GetData	Simple	Update
- VerifyData	Simple	Read
- InitializePoll	Medium	Update
- UpdatePage	Simple	Update

Table 4.12: Partitioning and specification of Create Poll use case by sub use cases.

As seen in table 4.12, the complex use case, Create Poll, has been partitioned into four simple to medium complexity update and read functions. These use cases are well defined, precise, and provide a better specification for the use case Create Poll.

Additionally, it is easier to understand the process of creating a poll with the specification in mind as opposed to without it. The process of creating a poll is to get the data inputted by the user (GetData), read and verify the data (VerifyData), initialize a poll based on the data (InitializePoll) and finally update the system to reflect the created poll (UpdatePage).

Furthermore, to exemplify the context of the use case, Create Poll, while also detailing the use case itself, a state chart diagram can be seen on figure 4.1.

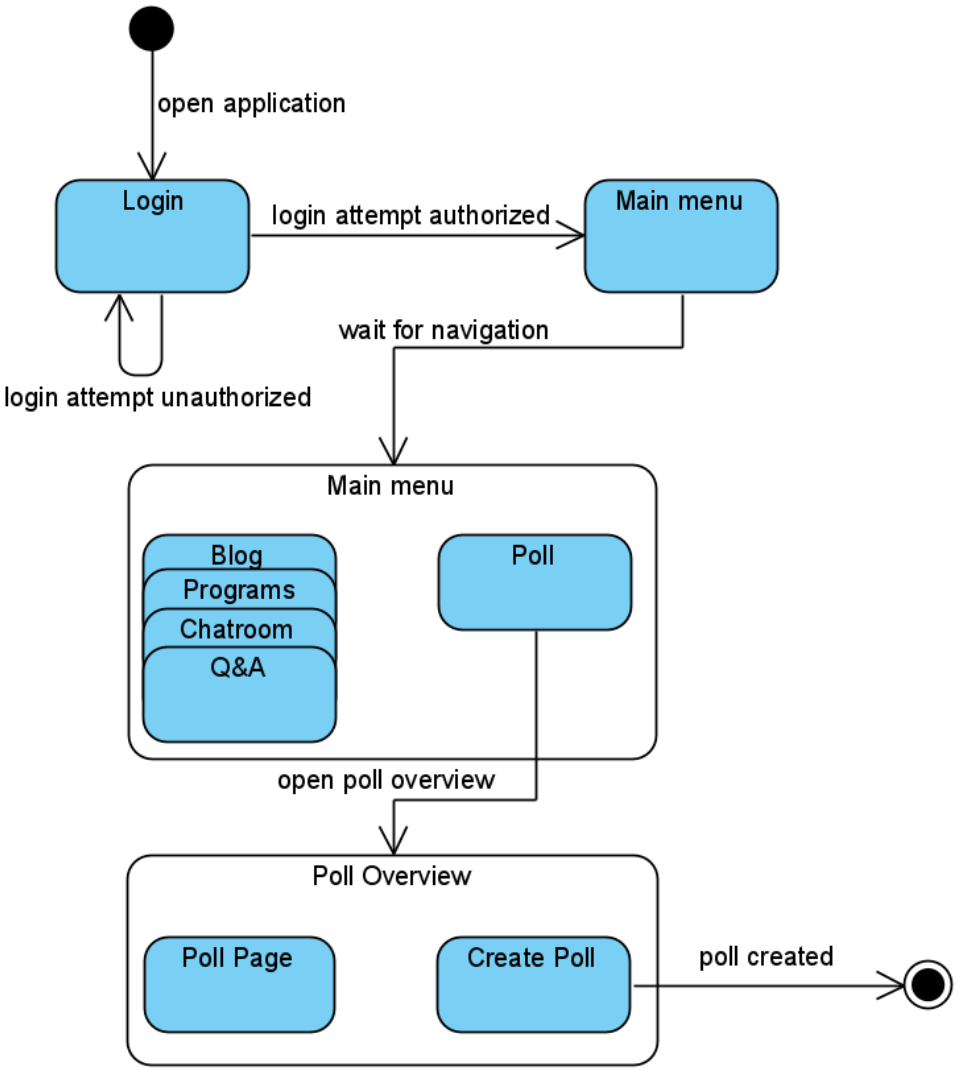


Figure 4.1: State chart diagram for creation of a poll.

5 | Architecture

To understand how a system works, it is important to get a grasp of its architecture. The structure of the system has so far only been represented by the class diagram and the behavioral patterns, presented in section 3.3 and section 3.4 respectively. Together they provide a core understanding of the system, and the next step is to consider the architecture of the system as a whole.

5.1 Components

The components of a system are a defining factor and a key element in a successful system. The components of a system consist of well defined parts that constitute a whole. A component can therefore be one class, a set of classes, or the whole system. In this project, components will be used to describe a set of classes, which define an isolated part of the system. From figure 3.1 in section 3.3, it is clear that there exists multiple subsystems whose only relation is through users.

This presents an obvious possibility to use the clusters, introduced in section 3.3.1, as components. The clusters will work as components because they each constitute a whole in themselves encapsulating a collection of functionality. However, it is important to note that even though the clusters are components, a component is not necessarily a cluster, as the term cluster has been appropriated to mean a certain thing in the context of this report, and does not necessarily hold the same meaning outside the scope of this report.

This leads to the following components being: *Q&A*, *Program*, *Chatroom*, *Blog*, *Poll* and *User*. These components will be further described by separating them into a model component (section 5.1.1) and a function component (section 5.1.2).

5.1.1 Model component

The purpose of creating a model component, is to expand the class diagram by adding attributes and implementing new classes if need be. To accomplish this, the attributes of the state chart diagrams described in section 3.4, are added as attributes in the respective classes

An example of this is the *Q&A* state charts. From the state chart diagram in figure 3.3, it is seen that both the date and the user should be remembered, consequently the *Q&A Thread* class had the date attributes added and the user aggregated, which can be seen in figure 5.1. Alongside these events it later proved necessary to have an id of of each thread, hence the *id* attribute. Furthermore, it is required from Sportstiming, that a Q&A thread is closed whenever the question is answered. Therefore the *Answered* attribute of the class is updated whenever the operation *AddComment* in the *Q&A* class is performed and the *Answer* attribute of the comment is set as true.

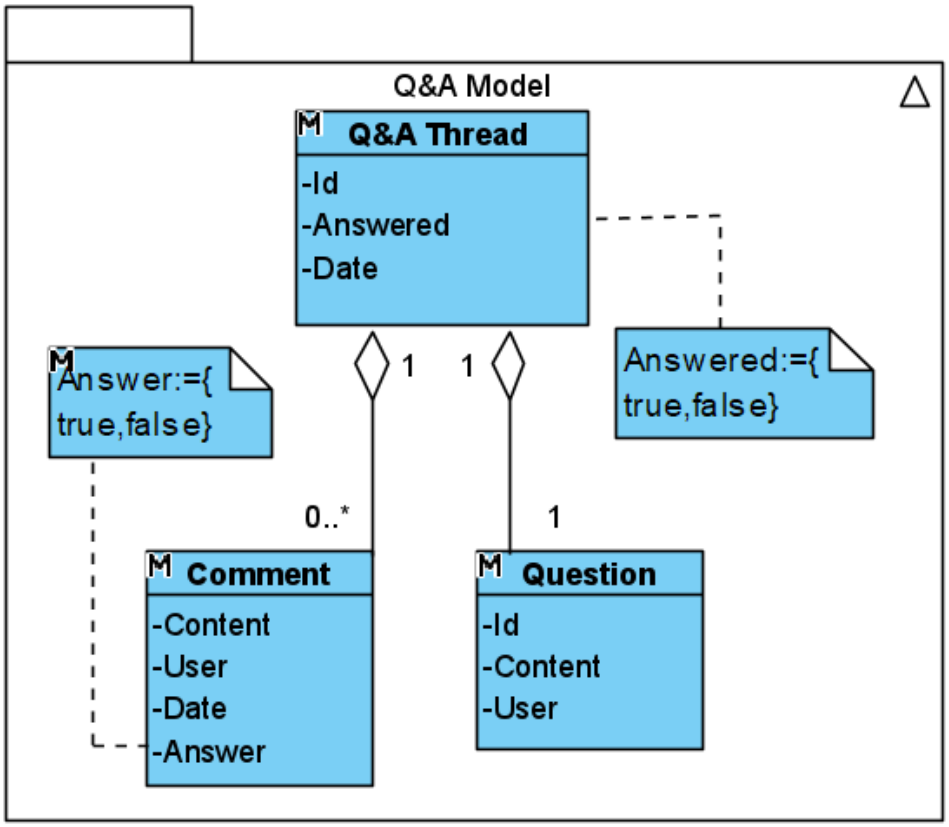


Figure 5.1: Q&A model component

As the argumentation for the rest of the components is trivial it will not be explained, but the complete result of the model component task can be seen on figure 5.3.

5.1.2 Function component

Expanding on the model component the function component is added. Creating the function component consists of adding methods to your model component providing a complete overview of the system. To add methods to the model component, the events from the state chart diagrams are added as methods in the correct classes. An example of this is depicted in figure 5.2, where the events *QuestionAsked* and *CommentAdded*, have been added as *AskQuestion()* and *AddComment()* respectively.

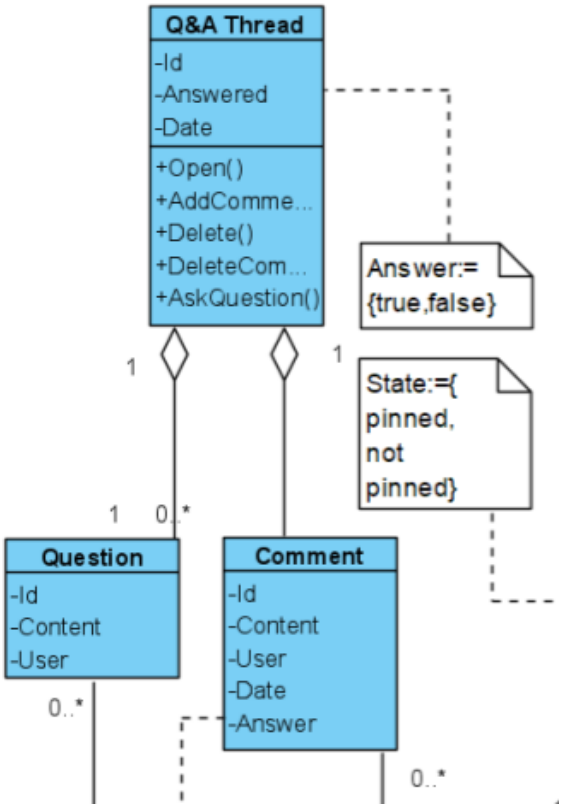


Figure 5.2: Q&A function component

As the the procedure for the rest of the components is trivial it will not be explained, but the complete function component diagram can be seen in figure 5.3.

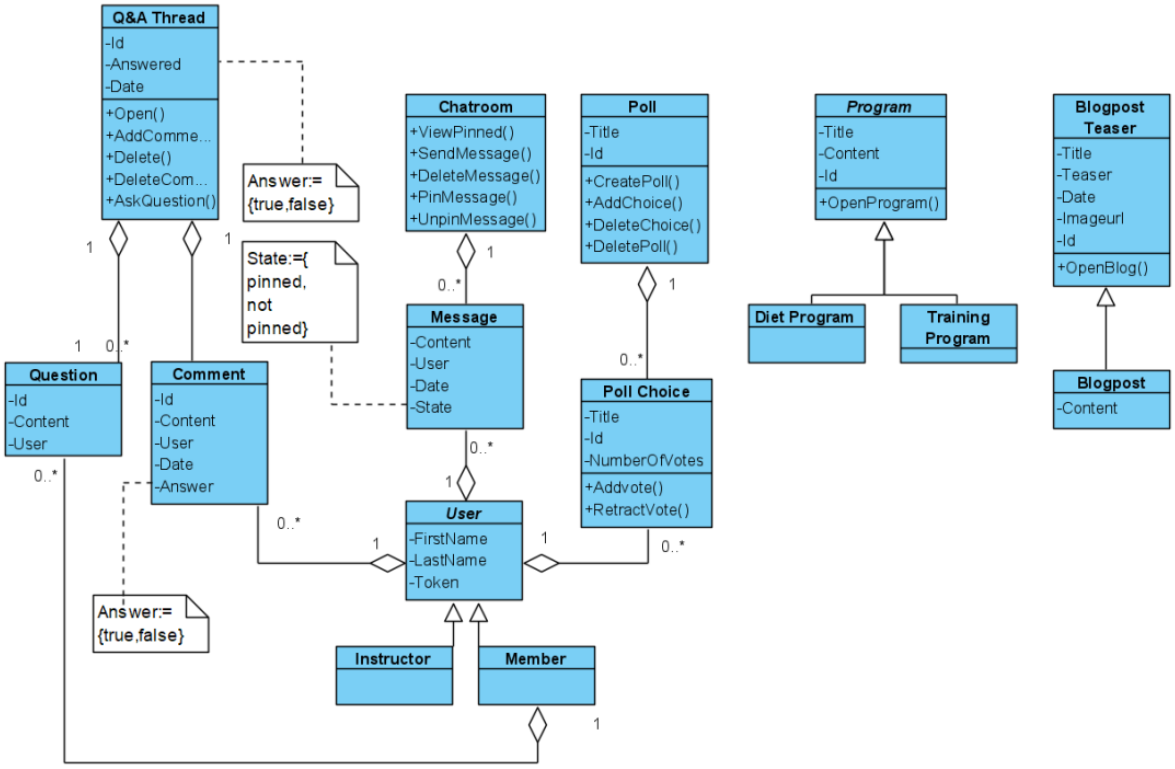


Figure 5.3: Complete function component

5.2 Pattern

The pattern of a system's architecture describes how components are connected, and what responsibilities they have. Examples of architecture patterns include the client-server pattern and the layered pattern. Though they are different patterns, they are not mutually exclusive and can be used in the same architecture.

The client-server pattern splits the system between two parts, one being the client and the other the server. This means a system using the client-server pattern can have more access points, as each users has their own access point in the form of a client, rather than connecting through the same access point.

The layered pattern describes a system that works in layers, which ensures encapsulation and low coupling if the layers can only interact with layers immediately adjacent to it. This is known as a closed-relaxed architecture. However, this means that a layer that might need a functionality multiple layers below have to go through multiple other layers to call it, instead of being able to call it directly. If layers are allowed to call components of any other layer it can lead to high coupling. This is known as open-relaxed architecture [2,p. 197].

The system has separation between components based on the different functionalities such as *Q&A*, *Chatroom*, *Training Programs* and *Diet Programs*. However, the functionalities of each component are also separated into different layers, as most of the functionality is split between the client and server. An example is creating a poll, which involves using the client to gather the necessary data before sending it to the server, where it is then made into a poll, added to a database, and made available to the client.

This of course leads to the client-server pattern, which makes it possible to contain and interact with the model at the server, while the UI and individual functionalities are contained on the client. This allows for the server to perform critical functionalities without it being available to the client, and therefore not easily tampered with. This is highly desirable, as the system handles users, API-keys, databases, and other sensitive data (see section 7.2.3). Furthermore, most of the system's functionality is common for all users and should be handled on the server, as opposed to on the client, to avoid unwanted interaction.

The client-server pattern can be specified further by determining where the UI, model component and function component are stored, as there are variations of the client-server architecture depending on where the different elements are stored. The different client-server variations can be seen on table 5.1. In the system all UI is stored on the client, but it is not as clear a distinction regarding functionality.

Some functionality regarding the UI, such as adding dynamic messages is contained on the client, and the client therefore contains both the UI and some functionality. The server side contains core functionalities that interact with the model, such as adding messages to the *Chatroom* and updating votes on a poll in *Polls*. Therefore the server side contains the model, but also some of the function component. This means that the system uses the distributed functionality architecture, which can be seen in table 5.1. Additionally the architecture for the system has been illustrated on figure 5.4.

Client	**Server**	**Architecture**
U	U + F + M	Distributed presentation
U	F + M	Local presentation
U + F	F + M	Distributed functionality
U + F	M	Centralized data
U + F + M	M	Distributed data

Table 5.1: Different forms of distribution in a client-server architecture [2,p. 202:figure 10.9]

By stating that the system uses the distributed functionality architecture, it implies that there is also some level of abstraction by layers, which leads to the system also using the layered architecture as seen in figure 5.4.

From this it is clear that the system uses both the client-server architecture and the layered architecture. Furthermore, it is visible from figure 5.4, that the system uses a closed-relaxed pattern, as it is only possible for a layer to interact with a layer immediately adjacent to it.

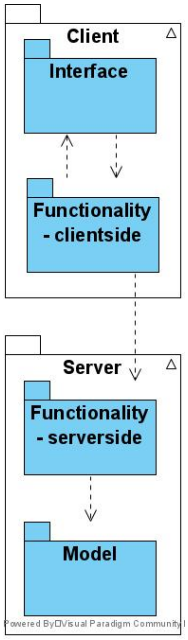


Figure 5.4: System architecture

5.3 Coupling and cohesion

To obtain a high level of flexibility and comprehensibility, the principles of *cohesion* and *coupling* are introduced. *Cohesion* describes to which degree a class or component constitutes a whole with essential relations between its parts [2,p. 275], and therefore it is wanted to have a high level of cohesion, whereas it is desirable to have a low level of coupling, as it describes to what degree a change in one class or component necessitates change in another class or component [2,p. 274].

By looking at the function component in section 5.1.2, it is clear that the system excels in achieving low coupling, and high cohesion. The low coupling stems from a clear separation in the component architecture as demonstrated by the function component diagram section 5.1.2. Looking at *Q&A*, *Chatroom* and *Poll*, it is clear that they do not employ functionalities from one another, but instead only use functionalities from within their own component.

The only coupling in the system is the *User*, since it employs functionalities from nearly every component, however it only employs the public functionalities provided by the components through the function component, but never interacts with private functionalities.

Therefore a change in the *Chatroom* functionalities will not necessitate a change in the *User* unless the change is performed to the public functionalities. Thereby the system ensures low coupling by only using outside coupling.

This asserts that the system has a high level of cohesion, while maintaining a low level of coupling.

Part III

Solution

6 | Design

A well thought out design is essential to ensure a user friendly and well functioning application. Therefore multiple meetings with Sportstiming was spent talking about design and evaluating prototypes. From the beginning Sportstiming had a clear vision of how they wanted the application to look, which made it easier to create prototypes and discuss variations.

6.1 Prototyping

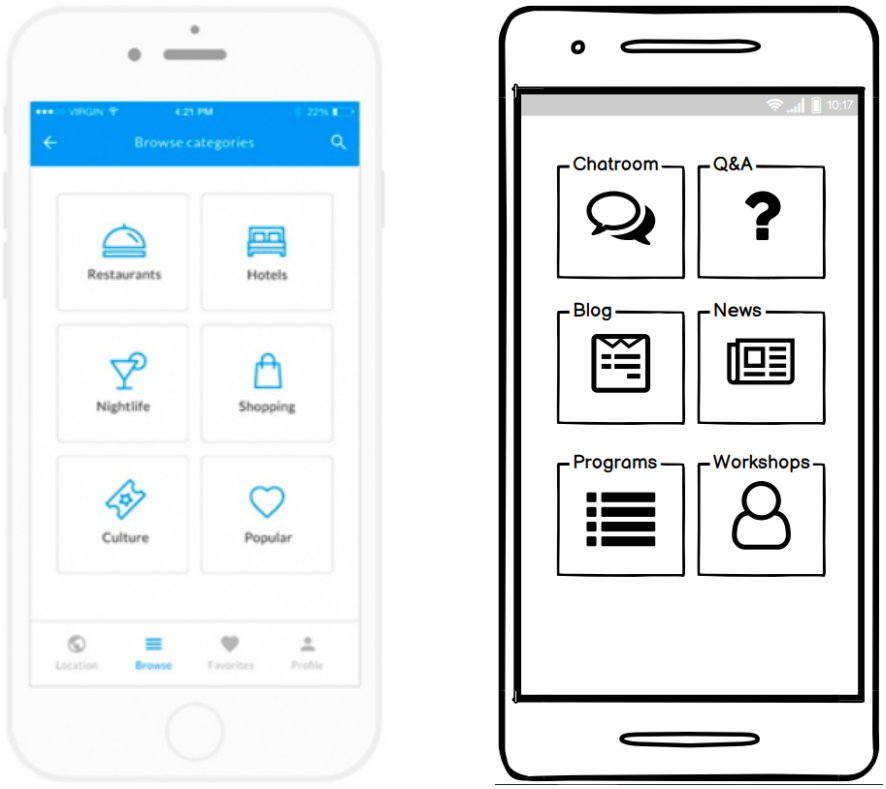
A prototype is a visual representation of a product. Prototypes help clarify, evaluate and verify design decisions, by examining and evaluating different prototypes. In particular prototypes are often used in collaboration with the customer to establish a common ground for discussion of functionality and design. There are a number of different types of prototypes which are sorted in two categories: High-Fidelity (Hi-Fi) and Low-Fidelity (Lo-Fi). A lo-fi prototype is quick and dirty sketch or layout of the design. This is used to visualize key elements and layout to aid development of more detailed prototypes. Usually presented as a sketch or a wireframe, but can even be sketched out on a napkin if need be. A hi-fi prototype on the other hand is a detailed prototype which should represent multiple features, interaction and full flow of the functionality in mind. It works as an acceptance test and should be agreed upon before implementation of the feature is carried out [4,p. 195].

During development of the application, hi-fi prototypes where developed using balsamiq wireframes as it allows for navigation between PDF pages simulating the actual interaction with the functionality within the application [5]. The prototypes presented in the following sections are all developed using balsamiq except the poll sketch (see figure 6.4) as it was sketched out in hand using lo-fi prototyping. The lo-fi prototype for the Poll component was used as an internal design document, which guided the developers to clarify UI elements and layout.

6.2 Design process

As the project is were set to follow the iterative structure, the preliminary plan was to have each prototype follow a certain iteration. This would not turn out to be case, as the project started to move away from the sprints the further it got. Therefore, instead of presenting one prototype per meeting, we had a meeting were the all the remaining prototypes were reviewed. As a result development of all features could begin simultaneously.

A common request for all of the prototypes was to use their company colors as a baseline. Furthermore they presented a sketch which they would like us to replicate and build upon. On figure 6.1 one can see their wishes and how the corresponding prototype ended.



(a) Wanted layout for the menu (b) Final prototype for main menu

Figure 6.1: Wanted design and resulting prototype [6]

6.3 Chatroom

As evident from the MoSCoW list in section 2, the chatroom contains functionality in both the Must have, Should have and Could have. This raised the issue of deciding what should be part of the prototypes, as it might not be fulfilled, however the prototypes ended up containing all features from the MoSCoW list as it was deemed a necessity to visualize

the final product. By looking at figure 6.2a one can see the chatroom as presented to the user where figure 6.2b shows the context menu which makes it possible to delete and pin messages. Furthermore it is visible that certain users, in this case *Kurt Normark*, has a badge next to their name. This badge represents a special status of the user e.g. an instructor. In figure 6.2b the message in focus has a pin on it symbolizing it is a pinned message. Furthermore to see the pinned messages one would press the *pinned* button in the top right corner which presents a context view of the pinned messages.

(a) Chatroom (b) Chatroom context menu

Figure 6.2: Chatroom

As it was determined from the beginning that the Must have features of all clusters were to be implemented before any Should haves could be implemented.

6.4 Q&A

One prototype that presented multiple options along the way and called for greater clarifications was the Q&A. First of all it had to be determined if all Q&A's should be

presented in one long thread or each question as a button containing a description. The buttons was then chosen and the next question arose: should the description include both the question and the answer or only the question. The group opted for both question and answer, but when evaluating with Sportstiming, they wanted only the question, as having the answers presented would make the description to long. This lead to a new obstacle, as it was no longer possible to determine whether or not a question was answered. It was chosen that the question should have a small check mark within it to recognize whether or not the question is answered. On figure 6.3 two prototypes is presented. One with both answer and question as a part of the description and the other with only the question being part of the description.

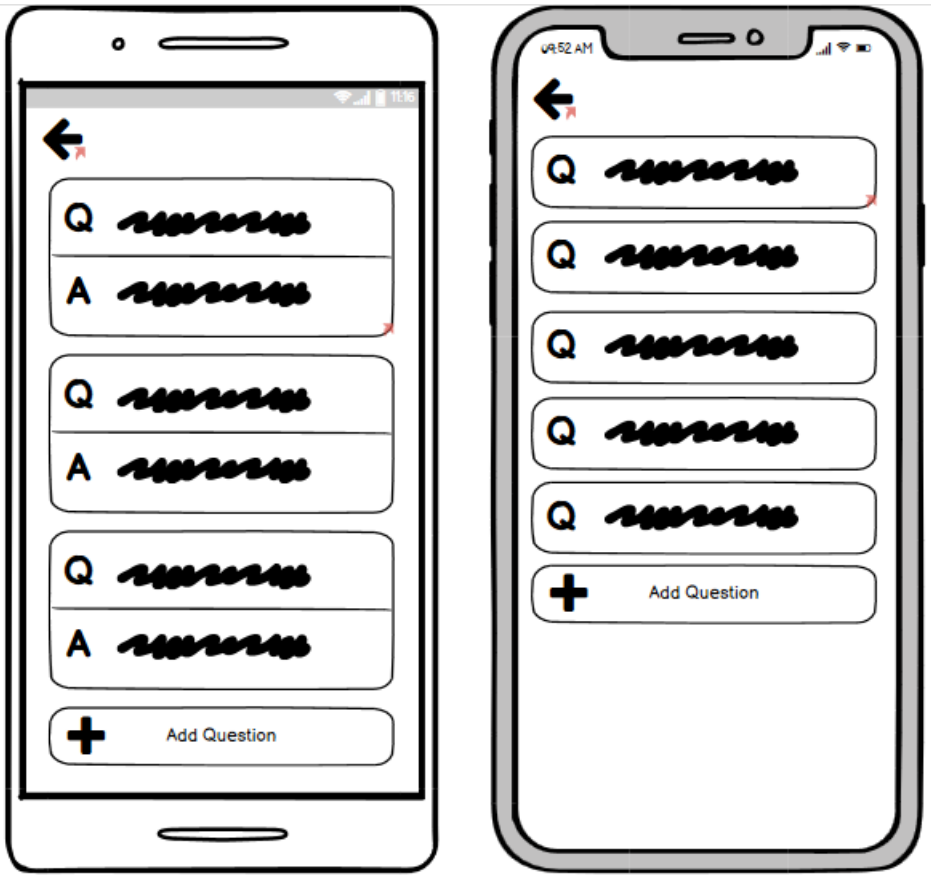


Figure 6.3: Q&A prototypes

6.5 Polls

When designing the polls section there was multiple different possibilities. All polls could be presented on the same page, a page could contain only the titles of the polls, or something completely different. As there was not one clear choice of the design, it would not make sense to create multiple hi-fi prototypes as it is more time consuming than

lo-fi prototypes. Instead a lo-fi protoype, more precise a sketch, were made to tryout the designs. The sketch of the final product can be seen on figure 6.4

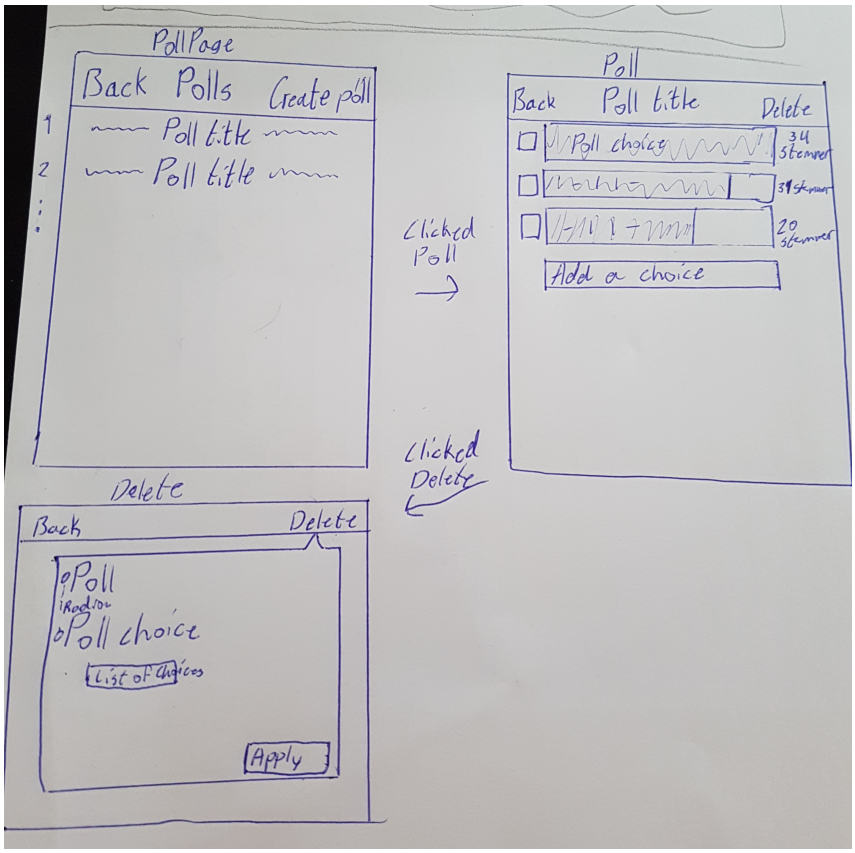


Figure 6.4: Sketch of the poll feature

7 | Implementation

This section is about how the solution were implemented, what tools were used and what solutions were needed to solve interesting problems encountered while developing the application.

Workflow

During development, it is often incredibly useful to see and interact with the application, which in case of mobile applications, means either emulating a device or running it on a physical device. Emulation was opted for, as it is quicker to deploy, and also makes it possible to test different operating systems. However, it is not possible to compile or deploy an iOS application without a macOS machine. Because of that restraint, the application will only be ran and tested on emulated Android devices.

7.1 Client

Following the client-server structure described in chapter 5 the client will be implemented as an app. Therefore the following sections will discuss frameworks and tools used to develop the app.

7.1.1 Xamarin

A central requirement, presented by Sportstiming, was that the product be implemented as a mobile application. Furthermore, as mentioned in the preface, a constraint for this semester's project is that the product be developed using C#, which results in the Xamarin framework being the only choice that coincides with the requirement and constraint, as no other mobile application framework uses C#.

Xamarin is built on top of .NET, and extends its capabilities with tools and libraries meant specifically for developing apps for a number of platforms, of which the most interesting for this project are Android and iOS. Xamarin is an abstraction layer that manages communication of shared code with underlying platform code, which makes it possible to share application code across platforms, while still maintaining native performance.

There are primarily two types of Xamarin, which is Xamarin.Forms and Xamarin.Native. The differences between the two is that the UI in Xamarin.Forms can be used on both Android and iOS, whereas Xamarin.Native also makes use of functionalities specific to each platform. Xamarin.Native is a common way to denote the otherwise distinct Xamarin.Android and Xamarin.iOS. The commonality is the approach to designing the UI, where Xamarin.Forms makes it possible to create UI for both platforms at the same time. However, it does this at the cost of not fully utilising each platforms differences. Android and iOS have similarities, but also differences, and because of that, focusing on the similarities as Xamarin.Forms does, means one cannot utilise their differences while sharing all the code.

Though, because of a lack of experience programming UI for Android and iOS apps, combined with the time constraints placed on the project, Xamarin.Forms was preferred.

7.1.2 XAML

In Xamarin, application UI can be created with either C# or XAML, also called Extensible Application Markup Language. As the name suggests, XAML is based on the more common XML language, and can be used to create numerous UI objects such as Buttons and Images.

Xamarin makes it possible to create the UI and logic entirely in C#, but it also allows for the UI to be created in XAML, and for C# to interact with XAML to make it dynamic. It is therefore often the case that a XAML page is linked to a C# file, as is the case with *ContentPage*s, which are used throughout the application.

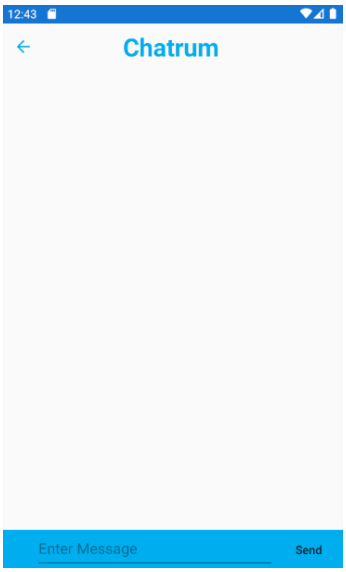
An example can be seen on figure 7.1a, where XAML is used to create the UI seen on figure 6.2a. As an example on 7.1a line 16, you can see an `Editor` being created, and multiple properties being set, such as `Placeholder`, which is set to "Enter Message". An `Editor` is a text field that a user can write in, and the result of the XAML code can be seen in the bottom of figure 6.2a, where it has a text field with the placeholder text, "Enter Message".

Figure 7.1a also shows a button being created on line 17, and at the end of the line you

can see the property `Clicked` being set to a string, and that string refers to a method by the same name in the linked C# file, which is then run when the button is clicked.

```
<StackLayout x:Name="MainStackLayout">
    <ScrollView VerticalOptions="FillAndExpand">
        <StackLayout x:Name="MessageContainer" Padding="5">
            <!--
                The messages go here :)
             -->
        </StackLayout>
    </ScrollView>
        <ScrollView VerticalOptions="End" BackgroundColor="#00aeef">
        <Grid ColumnSpacing="0" RowSpacing="0">
            <Grid.ColumnDefinitions>
                                <ColumnDefinition Width="1*"/>
                                <ColumnDefinition Width="7*"/>
                <ColumnDefinition Width="2*"/>
            </Grid.ColumnDefinitions>
                        <Editor x:Name="MessageInput" Grid.Column="1"
                        ↪  Placeholder="Enter Message"
                        ↪  Keyboard="Chat" BackgroundColor="#00aeef"
                        ↪  AutoSize="TextChanges"
                        ↪  HorizontalOptions="FillAndExpand"/>
                        <Button Text="Send" Grid.Column="2"
                        ↪  VerticalOptions="End"
                        ↪  BackgroundColor="#00aeef" CornerRadius="0"
                        ↪  Clicked="SendMessage"/>
        </Grid>
    </ScrollView>
</StackLayout>
```

(a) Chatroom UI in XAML



(b) Chatroom UI

Figure 7.1: Chatroom XAML and UI

7.2 Server

As described in chapter 5 there must exist a server containing an interface known to the client. A number of decisions must be made, when choosing a server and the reasoning behind should also be discussed. Therefore the following sections will explain the decisions made in regards of building the server with the first being why the server is made as a RESTful API.

7.2.1 REST

REST or 'Representational State Transfer' is an architectural design style for client-server communication. It imposes five different constraints that a server should adhere to. Those constraints are the requirements for a server to be "RESTful" [7].

The first constraint REST dictates is to separate user interface from data storage. Therefore it is not allowed to store any resources client-side. This separation allows the components to change independently of each other.

The second constraint of REST is for each request to a server to be stateless. Every request to the server is required to contain all information needed to complete the request and form a response. The server should therefore not use any stored context about the client.

The third constraint of REST is called "the cache constraint" which require that every response from the server, is marked, either explicitly or implicitly, as either cacheable or non-cacheable. This makes it so the client is able to temporarily store data from previous responses and could improve network time since some requests do not have to be made again. However, this would also mean that the client is not guaranteed to have the most up-to-date data and is therefore unreliable.

The fourth constraint is to have a "uniform interface". This means that every request takes the same form, and the data in the request is what changes the response. For this project, this uniform interface was chosen to be HTTP, which is a protocol that can be easily used with the uniform interface constraint to be used along with REST.

The Fifth constraint is to have a layered system, like the layered system architecture described in section 5.2 and will therefore not be explained again here [8].

The usage of REST gives a defined way to design and implement the client-server interface together with HTTP, where no client can influence the state of another client and/or server. It gives the opportunity to efficiently reuse data through caching and makes sure that client-server communication is well-defined. Which is why it was chosen as the architectural design for transferring data between client and server.

It should be noted that REST has an additional constraint, however, it is an optional constraint which will not apply to this project.

7.2.2 Socket

While HTTP is a great choice for infrequent requests, it becomes an obstacle when transmitting constant data, or wanting updates as fast as possible. HTTP is therefore not the most optimal solution for these situations.

Since the application has to implement a chatroom, indicated by the must-have requirement from section 2, it would be necessary to use some form of network communication to transfer messages.

A socket connection is highly favorable since a socket allows for full duplex data transfer, hence the client does not need to periodically keep sending HTTP requests causing more

load than necessary to the server. Instead, a connection will be established and kept open, until the client application terminates it.

7.2.3 The Web-service

From the beginning of the project it was made clear by Sportstiming that parts of the application in hand included some way of accessing their data. At first this seemed complicated, as we somehow should be able to access data only accessible to members, that were already logged in. However to avoid this Sportstiming set up a web-service for communication.

The web-service is a RESTful API which is accessible by sending a HTTP request and a bearer token. An example is the ability to determine if a user is a subscribing member or not, in which case they should not be allowed to use the app.

To verify a user, a HTTP POST request should be send to the web-service containing username and password, which would then respond with a JSON object as seen in figure 7.2. The key *authenticated* denotes whether or not the user was authenticated. The *member* key denotes whether or not they are a subscribed member. The *roles* describes what roles the user has. Roles are separated with a comma symbol. This also means that a response also could consist of both *member* and *instructor* or just one of them depending on their status since any instructor is also a member. Lastly, the *token* key is used throughout the application to access data from the web-service after initial login as the client is not permitted to store the username nor password.

```
1 {
2     "authenticated": "true/false",
3     "member": "true/false",
4     "roles": "instructor/member",
5     "token": "XX",
6     "firstname": "XX",
7     "lastname": "XX",
8 }
```

Figure 7.2: Web-service login response

7.3 Router

In order to handle multiple types of requests, a router class was implemented with the goal of routing different paths to the appropriate actions.

7.3.1 Defining a route

A route is a path and a HTTP request method, that produces a certain response from the server. An interface called IRoute shown in figure 7.3, was made to enable different implementations of routes, which are all still accepted by the router class, while encapsulating the response from the server.

```
1 public interface IRoute
2 {
3         public string Path { get; set; }
4         public HttpMethod Method { get; set; }
5         public ResponseAction Action { get; set; }
6 }
```

Figure 7.3: IRoute interface

The `HttpMethod` class was used instead of a string because the class defined what request methods are supported. This meant that an unknown request method cannot be used, which can save the program from potential errors.

To specify what action should be carried out in the Route, a delegate called `ResponseAction` was made, which is shown in figure. 7.4.

```
1 public delegate Task<byte[]> ResponseAction(HttpListenerContext context);
```

Figure 7.4: ResponseAction delegate

The `ResponseAction` delegate shown in figure 7.4, is defined with a return type of `Task<byte[]>` to give the server the ability to handle multiple requests at once, the byte array returned by the task is what the server sends back to the client.

7.3.2 The router class

The `HttpRouter` class is used to control and manage the routes, and is used by the server class to find the right response to a request. The router class is responsible for deciding what response should be sent, while the server class is responsible for receiving requests and sending responses.

The `HttpRouter` class contains a dictionary of implementations of `IRoute`, where the key is a string consisting of the path of the route, combined with the HTTP request method,

and the value is an implementation of `IRoute`. The request method is added to the key to enable one path being called with different request methods.

To get the correct task for a response, a method called `GetTask` was implemented which is shown in figure 7.5.

```
1 public Task<byte[]> GetTask(HttpListenerContext context)
2 {
3     IRoute route;
4     string key = context.Request.Url.AbsolutePath + context.Request.HttpMethod;
5     if (_routeDictionary.TryGetValue(key, out route))
6         return route.Action(context);
7     return _defaultAction(context);
8 }
```

Figure 7.5: GetTask method of the HttpRouter class

The `TryGetValue` method of the dictionary class is used to get the correct route on line 5 in figure 7.5, the method returns a boolean that indicates if an IRoute implementation with a matching key was found. If a route is found, the result of the `ResponseAction` is returned. If the route is not found, the result of the default action, `_defaultAction`, will be returned on line 7.

```
1 public void Add(IRoute item)
2 {
3     string key = item.Path + item.Method;
4     _routeDictionary.Add(key, item);
5 }
```

Figure 7.6: Add method of the HttpRouter class

To add routes to the router, an `Add` method (shown in figure 7.6) was implemented to simplify usage of the router by encapsulating the technicalities of the dictionary.

7.4 Chatroom

As mentioned in section 7.1.2, the user interface is defined by the XAML language. Therefore, the static part of the chatroom was made in XAML. Whereas the functionalities of messaging and displaying messages themselves, was written in C#.

7.4.1 Messaging functionality

Once the chatroom has been opened by the user, a socket connection to the server is established by calling the method `OpenSocketAndListen` inside a try-catch block. This method is responsible for opening a socket and calls the method `ListenForMessage` to handle incoming data if the try-catch does not encounter an exception. The messages are then transmitted through the socket, which is responsible for handling incoming messages. The method `ListenForMessage` is seen in figure 7.7.

```
1  private void ListenForMessage(SocketAsyncEventArgs package)
2  {
3
4      if (package == null)
5      {
6          //Package is the object that handles the shipping and recieving of messages
7          package = new SocketAsyncEventArgs();
8
9          // Runs this anonymous function whenever a message has been recieved
10         package.Completed += new EventHandler<SocketAsyncEventArgs>((object sender,
           ↪  SocketAsyncEventArgs e) =>
11         {
12             string jsonMessage = Encoding.Unicode.GetString(package.Buffer);
13             Message msg = JsonConvert.DeserializeObject<Message>(jsonMessage);
14
15             //Since UI updates can only be done on main thread, we queue a function to run on it that
               ↪  updates the chatroom.
16             Device.InvokeOnMainThreadAsync(() => _chatroomPage.PushMessage(msg));
17
18             // Listen for message again so a new message can be recieved
19             ListenForMessage(e);
20         });
21     }
22     else
23     {
24         // Reset the package so its ready for another go
25         package.AcceptSocket = null;
26     }
27
28     byte[] buffer = new byte[2048];
29     package.SetBuffer(buffer, 0, buffer.Length);
30
31     _Socket.ReceiveAsync(package);
32 }
```

Figure 7.7: ListenForMessage method in Chatroom.cs

`ListenForMessage` (figure 7.7) is structured around the `SocketAsyncEventArgs`, from *System.Net.Sockets*[9], which is used in conjunction with the socket to receive data. If `ListenForMessage` is called with `null` (line 4), the `SocketAsyncEventArgs` will get initialised and its `Completed` property (line 10) will be given an anonymous method to call when the socket receives a message, as seen in figure 7.7. However, if `ListenForMessage` is called with an existing `SocketAsyncEventArgs`, it will be reset so it is ready to receive another message, as seen in lines 22-26.

```
1  public class Message
2  {
3      private User _user;
4      private string _message;
5
6      public User User { get => _user; set => _user = value; }
7      public string MessageContent { get => _message; set => _message = value; }
8
9      public Message()
10     {
11
12     }
13 }
```

Figure 7.8: Message model

The anonymous method, that is created in figure 7.7 line 10, is the method that gets called once the socket has fully received a network message. It transforms the network message into a chat-message object, defined in figure 7.8, and uses `InvokeOnMainThreadAsync` to call `PushMessage` on the `ChatroomPage` instance which updates the UI with the new message.

7.5 Blog

A key feature in the app is the ability to read all blogs that have been uploaded to Sportstiming's website. This means that the app needs a way to fetch all blogs from the website and display the corresponding HTML document within the app.

To obtain this, two steps are needed. First of all, a GET request containing a user token is sent to the web-service (see section 7.2.3), which then returns a JSON object containing an array of blog teasers. The blog teasers contain everything seen in figure 7.9 other than the `text` key. Therefore a second GET request is required, which fetches a single blog requested from the client. The JSON response object is seen in figure 7.9. This response will then be used to represent the blog using the XAML WebViewer functionality.

```
1  {
2      "id": 289,
3      "date": "2020-11-10",
4      "title": "Derfor burde du ikke gå ned på løbeudstyr",
5      "teaser": "Uanset hvilket niveau du er på, så burde du ikke...",
6      "imageUrl": "/images/uploaded/news/gaaikkenedpaaloeb.jpg",
7      "text": "HTML"
8  }
```

Figure 7.9: Specific blog response

Within the process is an important discussion regarding the architecture and the responsibilities that follow. As stated in section 5.2 the system uses the distributed functionality pattern where functionality is both contained on the server and the client. To comply with this the client only manipulates the response received from the server and displays it in the app, whereas the server handles database manipulation and checks for new blogs. By doing this, the responsibilities are mainly contained on the server and only necessary functions like parsing JSON to C# objects is performed on the client.

Whenever the client updates the blog page, the server will check if it has a newer blog than the client by sending a request to the web-service and extracting the ID of the newest blog, hereafter it compares it with the ID of the newest blog on the client. If they match it will respond with a message saying that no updates were found. If they are not similar the server will send every blog until it reaches the ID of the client. The interaction can be seen in the sequence diagram in figure 7.10.

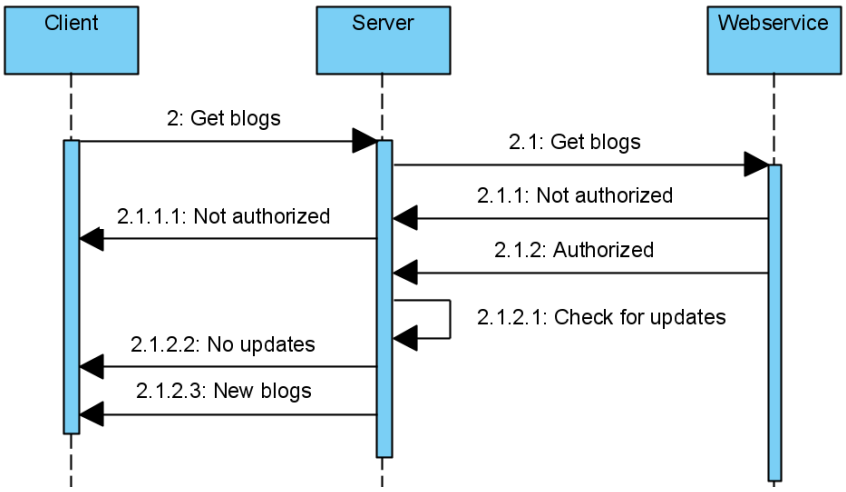


Figure 7.10: Sequence diagram for communication

7.6 Programs

In order to display the programs, it is necessary to first download them, as they are stored by Sportstiming as PDFs. Therefore a GET request with the user's token is sent to the web-service (see section 7.2.3), and a JSON object containing a list of download links to the programs is returned.

In the application the programs are listed into different categories depending on the sort of program. This is done to better present the programs, and give the user a good overview of the programs.

When a specific program is clicked, the client uses the URL for the specific program, to download the program, but before it can be used the URL has to be trimmed. When the URL is received from Sportstiming it contains the user's token as a query parameter, and that renders the link unusable, which is the reason the URL has to be trimmed.

On figure 7.11, the link is effectively trimmed on line 3-6, and passed to the `OpenAsync` method, from the Xamarin.Essentials' `Launcher` class, on line 8, which then opens the link in the browser where the PDF is automatically downloaded and displayed.

```
1 tapProgramItem.Tapped += (object sender, EventArgs eventArgs) =>
2 {
3     string input = program.Url;
4     int index = input.IndexOf("?");
5     if (index > 0)
6         input = input.Substring(0, index);
7
8     Launcher.OpenAsync(input);
9 };
```

Figure 7.11: The OpenAsync used to open the PDF's

Since the URL received from Sportstiming contains the user's token, as a query parameter, the URL is unusable. Therefore the token has to be removed before use, which is easily done by simply removing any query parameters from the URL as seen in figure 7.11 on line 3-6.

Opening the PDF with the provided link, opens it in the phone's browser. If the PDF is of significant size, the user will be prompted to download the PDF, after which they can manually open it. Unfortunately, opening local PDFs with `OpenAsync`, can prove problematic since Android and iOS handle local files differently, which would require more development time than was deemed available for this feature.

This has the unfortunate effect that the user has to either manually open the PDF or re-download it each time they want to view the PDF.

7.7 Polls

The application's poll feature is made up of three *Content Pages*, namely `OverviewPollPage`, `CreatePollPage`, and `PollPage`. Together these three pages facilitate the functionality put forth in the requirements in section 2.

First the deviations that the implementation has from the model will be explained, thereafter the pages that make up the poll feature will explained.

7.7.1 Model

The model describing the **Polls** cluster has been formed and detailed throughout the report, but during the implementation, parts of the model have been changed to better accompany the implementation and intended functionalities. This includes the introduction of classes, which are only used to transfer information between the client and server. They are called `PollVote` and `ChangePollItem`. `PollVote` is not to be confused with the class candidate of the same name, which was discarded in section 3.1. As the class candidate was conceived under a different class structure, and the new `PollVote` together with `ChangePollItem` were introduced as a means of transferring information between the client and server.

The reason classes were used to transfer information from the client to the server, is that it allows for packing of information in an object, and moving it between the client and server as a JSON object. This is opposed to sending it in plain text in a HTTP body, and extracting the data from the request or response.

The `PollVote` class has information regarding the creation of a new vote or the deletion of a vote, also referred to as casting or retracting a vote. This includes the poll ID, the choice ID, the ID of the user, and finally the flag that determines if a vote is being cast or retracted. The poll and choice ID are used to the correct choice, and the user ID is used to identify the user with the purpose being to keep the votes consistent. Such that a user can only vote on a choice once, and retract a vote only, if they have previously voted on a choice.

It is a similar situation with the `ChangePollItem` class, which is used for removing a poll and adding a choice to a poll.

7.7.2 Poll Overview

The retrieval and displaying of polls, is handled by `OverviewPollPage` and can be seen on figure 7.12.

On line 1, the method `ReceivePolls` sends a **GET** request for the polls to the server, and the response contain the polls in JSON, they are then parsed and returned to the property `_polls`. In the foreach loop on line 3 each poll, received from the server, is used

to initialise a corresponding `PollPage`, which are accessible by pressing the button for the respective `PollPage`.

Additionally, `OverviewPollPage` has a button that directs users to a `CreatePollPage`, where it is possible to create new polls.

```
1 _polls = await ReceivePolls();
2
3 foreach (Poll poll in _polls)
4 {
5     InitializePollPage(poll);
6 }
```

Figure 7.12: OnAppearing in OverviewPollPage.xaml.cs

7.7.3 Creating Polls

The creation of polls is handled by `CreatePollPage`. It presents the user with two text fields and a button to add more text fields. Placeholder text is written in each text field, such that a user can distinguish them. The placeholder of the first text field is "Titel", and the second has the placeholder "Valgmulighed". Any subsequent text fields added with the button all have the placeholder "Valgmulighed", as a poll can only have one title, but multiple choices.

Once the user has finished writing the poll and choice titles, they can press the check mark button at the top right. The button `Clicked` event then runs the `CreateAndSendPoll` method also seen on figure 7.13.

On line 3, the poll is created using the text inputted to the text field, and on line 5 the choices added are extracted in a similar fashion and added to the poll. The poll is then parsed to JSON and sent to the server along with the user's token using the `SendPoll` method.

```
1 private async void CreateAndSendPoll(object sender, EventArgs e)
2 {
3     Poll poll = new Poll(this.FindByName<Entry>("Title").Text);
4
5     ExtractChoices(poll);
6
7     SendPoll(poll);
8 }
```

Figure 7.13: CreateAndSendPoll in CreatePollPage.xaml.cs

The server then verifies whether the poll is valid, and that the user is an instructor using Sportstiming's web-service, since creating a poll is only permitted as an instructor.

If however any of those checks should fail, the server will send the client an appropriate status code, which in turn will dictate the action performed by the client. An example would be in the case of an invalid poll, where the client would then displays an alert that indicates the error was a result of an invalid poll.

7.7.4 Poll Voting

Handling the deletion, voting, and changes to an individual poll is `PollPage`. It initialises the poll choices, which consist of a checkbox for voting, a choice title, and a number representing the number of votes. It also provides an empty text field that makes it possible for users to add additional choices to the poll. Lastly it provides the functionality to delete the poll.

The UI for choices is created by the method `ChoiceUI`, it sets up the UI, namely the checkbox, title and number of votes for a choice, but most interesting is the checkbox. When the checkbox is clicked an event called `CheckedChanged` is fired, and by subscribing to this event, it is possible to perform actions according to the state of the checkbox, which is what can be seen on figure 7.14.

The if-else on line 4, determines, using the `IsChecked` property, whether the checkbox is checked or not, and whether a user has previously checked the box, i.e. if they are in the list of user hashes. This is done to provide consistency in the application, since users who have previously voted for the choice would expect to be informed accordingly. In the case that the checkbox is checked and the user has not previously voted for the particular choice, `SendChoiceVote` sends the information necessary for a vote to the server on line 7, and line 10-12 execute the necessary code to inform the user that the vote was successful. Specifically the addition of the user to the list of user hashes in order to prevent them from voting again, incrementing the vote count by one to reflect the new vote, and notifying the user of their successful vote by text.

On the other hand, if the box is unchecked or the user has voted previously, the opposite actions are performed. Namely `SendChoiceVote` is sent with a flag representing the retraction of a vote, and the client side removes the user from the list of user hashes, decrements the vote count, and notifies the user by text of their successful vote retraction.

```
1  checkBox.CheckedChanged += (sender, e) =>
2  {
3      // Add/Remove vote via parameter
4      if (checkBox.IsChecked && !choice.CheckForVote(User.CurrentUser.userHash))
5      {
6          // Server side
7          SendChoiceVote(new PollVote(poll.ID, choice.ID, User.CurrentUser.userHash, true));
8
9          // Client side
10         choice.AddUserVote(User.CurrentUser.userHash);
11         numVoteLabel.Text = choice.GetNumberOfVotes().ToString();
12         notifyVoteLabel.Text = "Stemme gemt!";
13     }
14     else
15     {
16         // Server side
17         SendChoiceVote(new PollVote(poll.ID, choice.ID, User.CurrentUser.userHash, false));
18
19         // Client side
20         choice.RemoveUserVote(User.CurrentUser.userHash);
21         numVoteLabel.Text = choice.GetNumberOfVotes().ToString();
22         notifyVoteLabel.Text = "Stemme fjernet!";
23     }
24 };
```

Figure 7.14: Client: Voting - ChoiceUI in PollPage.xaml.cs

`SendChoiceVote` sends the vote information to the server, and as seen on figure 7.14 it consists of the poll's ID, the choice's ID, the user's hash, and lastly whether a vote is being cast or retracted. On figure 7.15, one can see the servers response to the request from `SendChoiceVote`.

The vote object is first deserialised from JSON, after that it is determined whether the vote is being cast or retracted on line 7, and following the result the vote is added or removed accordingly on line 9 and line 13 respectively. The poll method `FindChoiceAndVote` uses the `Find` method in an identical way, searching through the choices for matching IDs on the list of poll choices, and according to the value of `vote.CastVote`, it then adds or removes the vote.

```
PollVote vote = JsonConvert.DeserializeObject<PollVote>(json);

byte[] data;

try
{
        if (vote.CastVote)
        {
                Polls.Find(p => p.ID == vote.PollID).FindChoiceAndVote(vote);
        }
        else
        {
                Polls.Find(p => p.ID == vote.PollID).FindChoiceAndVote(vote);
        }
}
catch (ArgumentNullException)
{
        data = Encoding.Unicode.GetBytes("The vote has failed, either because of missing poll or
        ↪  choice.");
        ServerUtility.SetContextBAD(context, data);
        return data;
}

data = Encoding.Unicode.GetBytes("Received vote correctly.");
ServerUtility.SetContextOK(context, data);
return data;
```

Figure 7.15: Server: Voting - NewPollChoiceReponse in NewPollVote.cs

The text field that facilitates the adding of additional choices can be seen on figure 7.16, and it uses an event handler on the `Entry` event called `Completed`. The event is fired when a user has finished entering text into the text field, and similarly to earlier event handlers, the one seen on figure 7.16 also has a server and client part. The server part sends the information required to create a poll choice to the server, and the client part updates the UI, and adds the newly added choice to the list of choices using the method `ChoiceUI`.

```
pollChoiceEntry.Completed += (sender2, e2) =>
{
    // Server side
    ChangePollItem item = new ChangePollItem()
    {
        ChoiceTitel = pollChoiceEntry.Text,
        PollID = poll.ID
    };
    SendAddPollChoice(item);

    // Client side
    poll.AddPollChoice(pollChoiceEntry.Text);
    ChoiceUI(poll, poll.PollChoices.Last(), choiceGrid, row, notifyVoteLabel);
    row++;

    pollChoiceEntry.Text = string.Empty;
};
```

Figure 7.16: Add choice in PollPage.xaml.cs

7.8 Log in

When the login process is first initiated, the user will be prompted for an email and a password. Once the user taps the button marked "log ind", the client sends the login information to the server, and the server passes it to the web-service. As described in section 7.2.3 the web-service responds with an object containing information about whether the login was successful or not. The server then sends the object from the web-service back to the client, and the login process is completed. This process can be seen in the sequence diagram on figure 7.17.

If the login was successful, the received login object contains a token, that can be used to authorise the user in later requests without using the aforementioned login credentials again as it is not permitted to store the username nor password, as mentioned in section 7.2.3. The token is cached on the client along with the rest of user's data received from the web-service.

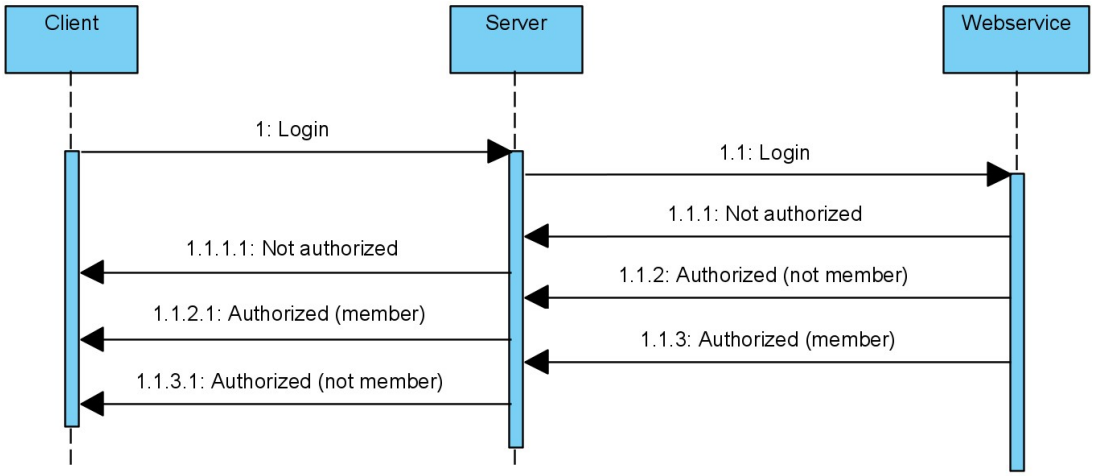


Figure 7.17: Login Sequence

Once the login process has been successfully completed at least once, the application will automatically log-in for the user. This is due to the token from the last successful login attempt being stored. When the application starts, it checks whether a token is stored or not, which can be seen in figure 7.19 line 3-4. If it does not have a token, it continues with the aforementioned login process, whereas if a token is stored, it initiates the automatic login process which is depicted in the sequence diagram on 7.18.

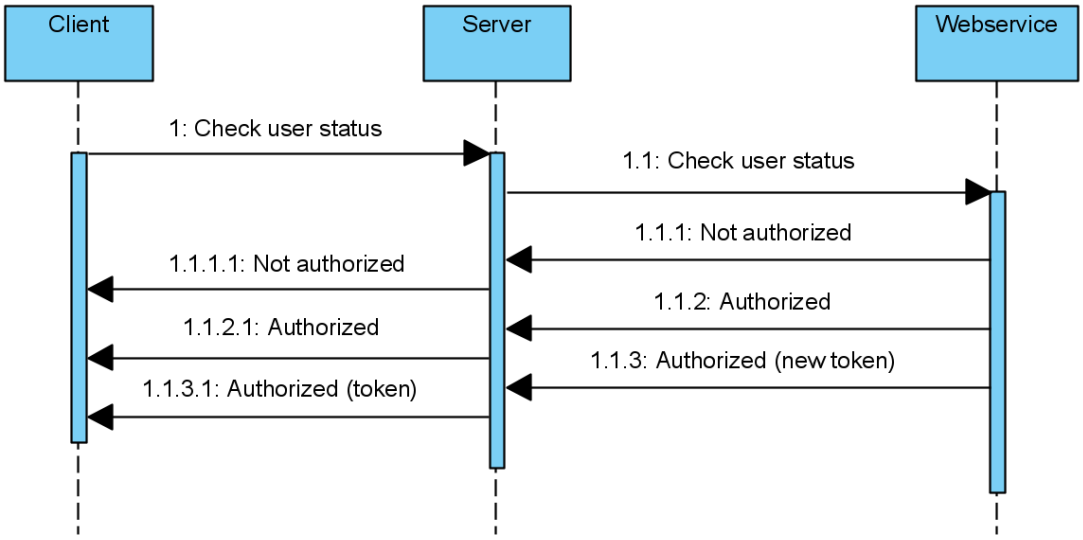


Figure 7.18: Automatic Login Sequence

Once the automatic login process has been initiated it sends the stored token to the server which sends it to the web-service. The web-service responds with a login object akin to the object received when the user first logged in. The server then sends the login object back to the client, the client then evaluates upon whether the login request was successful or not. In some cases, the login object received might contain a new token, as seen in 1.1.3 of figure 7.18, in which case the stored token will be updated.

If the login request was unsuccessful, the client deletes the stored user information and continues with the normal login process. This is the case seen in lines 30-33 in figure 7.19. However, if the login request is successful, the app successfully navigates to the main menu, as seen on lines 15-29.

```
protected override void OnStart()
{
    bool hasUser = UserDataBase.Table<User>().CountAsync().Result > 0;
    if (hasUser)
    {
        try
        {
            User currentUser = UserDataBase.Table<User>().FirstOrDefaultAsync().Result;

            StringContent content = new StringContent(currentUser.Token);
            HttpResponseMessage response = ServerConnection.PostAsync("isTokenValid", content).Result;
            LoginResponse loginObj = JsonConvert.DeserializeObject<LoginResponse>(
                Encoding.Unicode.GetString(response.Content.ReadAsByteArrayAsync().Result));

            if (loginObj.IsAuthenticated && loginObj.IsMember)
            {
                //In case Sportstimer returns with a new token
                if (loginObj.Token != currentUser.Token)
                {
                    currentUser.Token = loginObj.Token;
                    UserDataBase.DeleteAllAsync<User>().Wait();
                    UserDataBase.InsertAsync(currentUser, typeof(User));
                }

                User.CurrentUser = currentUser;

                /* The following part navigates the user to the MainMenu */
                MainPage.Navigation.PushAsync(new MainMenu());
            }
            else
            {
                // Saved user is no longer member
                UserDataBase.DeleteAllAsync<User>().Wait();
            }

        }
        catch (Exception e)
        {
            //Failed relogin somehow.. Continue as not logged in, write to console.
            UserDataBase.DeleteAllAsync<User>().Wait();
            Console.WriteLine(e);
        }
    }
```

Figure 7.19: Automatic Login code

7.9 Q&A

In this section the parts within the implementation of Q&A will be explained, first the controller and other classes on the server, then the pages made on the client.

7.9.1 Model

The model of the Q&A functionality has been implemented on the server, the function component shown in 5.1.2 figure 5.3, has been the starting point for the implementation.

The implementation has deviated from the state chart shown in section 3.4 figure 3.3. The ability to reject and mark answers has not been implemented. In the implementation the answer can only be overwritten. That means if a user is not satisfied with the answer, the only option is to post a comment on the thread, then an instructor could overwrite

the answer or post another comment. The functionality was not implemented because it was not prioritized as a *Must have* or *Should have*.

The Controller

The `QAndAController` is a class that controls and contains the Q&A functionality on the server.

To Store the Q&A threads a private linked list `_threads` was added to the controller, because the list will not be accessed by indexing, and the low cost of adding and removing items anywhere within the list.

To access the list of threads, a property `Threads` of the type `IEnumberable` was made. The property returns the linked list, as an `IEnumberable`, because only the controller should add or remove routes.

The controller has a method for getting a thread by an ID of the thread from the list called `Get`. To add threads to the list an `Add` method was implemented, and to add comments to a thread in the list an `AddCommentToThread` method was implemented.

Q&A Thread

The program contains a class called `QAndAThread` which is used to store information regarding a particular Q&A thread. Question, answer and comments, are stored inside the `QAndAThread` and the questions author is stored as `User` property. To Identify a thread, an `Id` property is given by the controller. The question class, seen in section 5.1.2 figure 5.3 of the function component for Q&A, was not implemented, but instead a string and user instance, was stored as properties of the `QAndAThread` class. The reason behind this deviance was that a question does not appear without a thread.

```
1  private async Task<byte[]> PostQuestionResponse(HttpListenerContext context)
2  {
3      try
4      {
5          //getting the body
6          byte[] buffer = new byte[context.Request.ContentLength64];
7          await context.Request.InputStream.ReadAsync(buffer);
8
9          QAndAThread thread =
           ↪  JsonSerializer.Deserialize<QAndAThread>(Encoding.Unicode.GetString(buffer));
10
11         if (string.IsNullOrEmpty(thread.Question))
12             throw new Exception("Empty Question received");
13
14         string token = _authenticator.VerifyUser(thread.User);
15
16         if (token == null)
17             throw new Exception("User not verified");
18
19         _qAndA.Add(thread);
20
21         //response
22         byte[] data = Encoding.Unicode.GetBytes(token);
23         ServerUtility.SetContextOK(context, data);
24         return data;
25     }
26     catch (Exception e)
27     {
28         Console.WriteLine(e);
29         byte[] data = Encoding.Unicode.GetBytes("Could not post question");
30         ServerUtility.SetContextBAD(context, data);
31         return data;
32     }
33 }
```

Figure 7.20: PostQuestionResponse of the PostQuestionRoute class

The `PostQuestionRoute` is the route responsible for accepting questions from the client, the client sends a new thread to the server in the body of a request, then the author of the thread is verified using the `Authenticator` class shown on line 14-17 in figure 7.20. On line 19 the responsibility of adding the thread is given to the controller, which is referenced as a field `_qAndA` in the `PostQuestionRoute` class.

Comment

A comment contains a string with the content and the user who sent it. A comment can be an answer and therefore has a boolean property `IsAnswer`, which indicates if the comment is meant as an answer or not. Only the instructor can send answers, thereby only comments sent by the instructor could have `IsAnswer` set to true. To ensure that answers from members are not accepted, a check was implemented to the `PostCommentRoute` class which is the route responsible for adding comments.

The route is similar to the `PostQuestionRoute`, because both routes verifies the users and then calls a method of the controller to add content to the controller. In the case of

`PostCommentRoute` the method called from the controller is the `AddCommentToThread` method seen in figure 7.21.

```
1  public void AddCommentToThread(Comment comment)
2  {
3      QAndAThread thread = Get(comment.Id);
4
5      if (thread != null && UserAllowedToComment(thread.User, comment.User))
6      {
7          thread.AddComment(comment);
8      }
9      else
10         throw new Exception($"a comment on {comment.Id} failed");
11 }
```

Figure 7.21: AddCommentToThread method of the QAndAController class

The addition of the comment is done through both the `QAndAController` and the `QAndAThread`. The `AddCommentToThread` tries to find a thread, if a thread is found it will call the `AddComment` method of the specific `QAndAThread` instance on line 7 of figure 7.21. The `AddComment` method then adds the comment to the list of comments or sets it as the answer depending on the `IsAnswer` property.

7.9.2 Pages

For the user to view, make, and interact with Q&A, two pages were made, a `QAndAPage` that shows teasers of the Q&A threads and a `QAndAViewerPage` which one are redirected too, when clicking a teaser. Here the thread is shown with question, answer and comments.

Because a list of threads could be expensive to send to the client, a teaser class was made. The `QAndATeaser` class contains the question asked, the date it was asked, if the question is answered, and an ID used by the server to get the thread by the `Get` method of the `QAndAController`.

Instructor check

The instructor should not be able to post a question, therefore when opening the `QAndAPage`, a check verifying that the user is not an instructor has been implemented. The instructor cannot see a button for actions that should not be used by an instructor, as the buttons visibility is dependent on the user's role.

```
1  public QAndAPage()
2  {
3      InitializeComponent();
4
5      if (!User.CurrentUser.Roles.Contains("instructor"))
6      {
7          AskButton.Clicked += ShowOverlay;
8
9          PostQuestionButton.Clicked += ShowOverlay;
10         PostQuestionButton.Clicked += SendQuestion;
11     }
12     else AskButton.IsVisible = false;
13 }
```

Figure 7.22: Constructor of the QAndAPage class

The check happens in the constructor of the `QAndAPage`, If the user is not an instructor, but just a member, event handlers are subscribed to the `Clicked` events of the `AskButton` and `PostQuestionButton` on line 5-11 in figure 7.22. The `AskButton` hides or shows the text editor for writing a question, while the `PostQuestionButton` sends the question, and is shown or hidden by the `ShowOverlay` event handler.

Displaying Q&As

On appearing, the `QAndAPage` will fetch the teasers and shows them by adding a `Frame` instance to the page. This is done through the `GetQuestions` method shown in figure 7.23. The method is called both after sending a new question and when the page appears.

```
1  public async Task GetQuestions()
2  {
3      QAndATeaser[] teasers = await FetchQuestions();
4
5      if (TeaserStack.Children.Count > 0)
6          TeaserStack.Children.Clear();
7
8      foreach (QAndATeaser teaser in teasers)
9      {
10         Frame bufferFrame = teaser.InitializeUI(null);
11         AddGestureRecognizer(bufferFrame, teaser);
12         await Device.InvokeOnMainThreadAsync(() => TeaserStack.Children.Add(bufferFrame));
13     }
14 }
```

Figure 7.23: GetQuestions method of the QAndAPage class

`GetQuestions` uses the `FetchQuestions` method, that sends a request to the server, the server responds with a list of teasers that the method returns. `GetQuestions` clears the list of children from the stack, to ensure that no duplicates appear on line 5-6 of figure 7.23.

```
1     private void AddGestureRecognizer(Frame frame, QAndATeaser teaser)
2     {
3         TapGestureRecognizer threadOpen = new TapGestureRecognizer();
4         threadOpen.Tapped += (obj, e) =>
5         {
6             QAndAViewerPage viewerPage = new QAndAViewerPage(teaser.Id);
7             Navigation.PushAsync(viewerPage);
8         };
9         frame.GestureRecognizers.Add(threadOpen);
10    }
11 }
```

Figure 7.24: AddGestureRecognizer method of the QAndAPage class

On line 11 of the figure 7.23 the `AddGestureRecognizer` is called. Each `Frame` instance from the teasers, get a `TapGestureRecognizer` class shown in figure 7.24. The `TapGestureRecognizer` made by Xamarin, handles the event of a user touching the UI element and enables UI elements to act like buttons and call methods on taps from the user. When the frame of the teasers is tapped a new `QAndAViewerPage` is instantiated as shown on line 4-8 of figure 7.24.

Making UI elements intractable instead of adding dedicated buttons, can use less space meaning more teasers can be shown in the limited space on the screen of the phone, without making elements smaller or removing the space between them. A problem with this approach may be that the UI elements do not look intractable to the user, which should be tested in usability testing. The problem can be alleviated by trying to get attention by making the text bold or using a different color, in hope of getting the user to tap it.

The `QAndAViewerPage` fetches the thread and then initializes the UI for the given thread and adds it to the layout of the page.

```
1  private async Task ShowThread(int id)
2  {
3      try
4      {
5          QAndAThread thread = await GetThread(id);
6
7          Frame threadFrame = thread.InitializeUI(null);
8          await Device.InvokeOnMainThreadAsync(() =>
9          {
10             StackXaml.Children.Add(threadFrame);
11             ConfigureCommentEditor(thread.User);
12         });
13     }
14     catch
15     {
16         await DisplayAlert("FEJL", "Kunne ikke finde siden", "OK");
17         await Navigation.PopAsync();
18     }
19 }
```

Figure 7.25: ShowThread method of the QAndAViewerPage class

The `ShowThread` method is called by the constructor of the `QAndAViewerPage` and `Update` method, which is called when the users adds a comment to the thread. The `ShowThread` method uses the `GetThread` method that fetches the thread with the given ID from the server on line 5 of figure 7.25. When the thread is shown a method called `ConfigureCommentEditor` is called with the threads author line 11. The `ConfigureCommentEditor` method will show and hide elements of the comment editor, according to what roles the user has and if the user is the author of the question of the thread.

8 | Test

A fundamental part of software development is testing. Testing ensures that the individual parts and integration of those parts work reliably, which greatly improves development processes by ensuring that ongoing changes to a function does not break the system unnoticed. Furthermore testing provides proof for a working program and should always be considered vital. Therefore this chapter will focus on unit testing and usability testing.

Besides unit, integration and usability tests, numerous different test methods were considered. Therefore this chapter will end with a section explaining what other types of test should be examined.

8.1 Unit test

To assist unit testing, the test framework NUnit is used. This is done as it is well documented and the Xamarin.UItest frameworks is built upon the NUnit test framework. When using NUnit the test should be defined by a `[Test]` attribute. Multiple different attributes exists, but only the `[Test]` is used in this project as the group does not have enough experience to utilize the various attributes. As the group uses Visual Studio for development, NUnit provides an interface showcasing all test and whether they have passed or not.

8.1.1 Blog

Within the blog functionality, several computational functions exist on the server. To ensure correct computation the functions should be tested to cover every edge case which is done by unit testing. An example is seen in figure 8.1 where it tests if *GetNewBlogs(blogTeaserList, clientNewestBlog)* returns the correct amount of new blogs. In this case one new blog exist on the server and therefore the count of the *newList* should be one.

```
1         [Test]
2         public void GetNewBlogs_OneNewBlogOnServer_Returns_1_NewBlog()
3                 {
4             // Arrange
5             List<BlogTeaser> blogTeaserList = new List<BlogTeaser>();
6             blogTeaserList.Add(new BlogTeaser()
7             {
8                 ID = 2,
9                 Teaser = "teaser",
10                Title = "title",
11                    Date = "date",
12                ImageUrl = "imageUrl"
13            });
14            int clientNewestBlog = 1;
15
16            int expected = 1;
17            // Act
18            List<BlogTeaser> newList = functions.GetNewBlogs(blogTeaserList,
              ↪  clientNewestBlog);
19
20            // Assert
21            Assert.AreEqual(expected, newList.Count);
22        }
```

Figure 8.1: Test of function GetNewBlogs

As the above test does not guarantee that the contents of the new blog is correct, another unit test was introduced to verify the contents of the blog. This test is seen in figure 8.2.

```
[Test]
public void GetNewBlogs_OneNewBlogOnServer_HasCorrectContent()
{
    // Arrange
    List<BlogTeaser> blogTeaserList = new List<BlogTeaser>();
    blogTeaserList.Add(new BlogTeaser()
    {
        ID = 2,
        Teaser = "teaser",
        Title = "title",
        Date = "Date",
        ImageUrl = "imageUrl"
    });
    int clientNewestBlogID = 1;

    int expectedID = 2;
    string expectedTeaser = "teaser";
    string expectedTitle = "title";
    string expectedDate = "Date";
        string expectedImageURL = "imageUrl";
        // Act
    List<BlogTeaser> newList = functions.GetNewBlogs(blogTeaserList,
    ↪  clientNewestBlogID);
    // Assert
    Assert.AreEqual(expectedID, newList[0].ID);
    Assert.AreEqual(expectedTitle, newList[0].Title);
    Assert.AreEqual(expectedTeaser, newList[0].Teaser);
    Assert.AreEqual(expectedDate, newList[0].Date);
    Assert.AreEqual(expectedImageURL, newList[0].ImageUrl);
}
```

Figure 8.2: Test content of new blog

8.2 Integration testing

Integration testing revolves around connecting components. The purpose of completing integration testing is to isolate and correct errors related to interfaces between individual components of the program. Before integration testing is done, the individual components should be unit tested to ensure that they work as an isolated unit.

As of this project the top-down approach is used for integration testing. The reasoning behind this, is that the rough outline of the UI and their navigational functionalities were the first to be implemented whereas underlying features like sending a message and interpreting server responses followed afterwards. This also meant that it was necessary to use stubs in some cases of testing e.g. when sending a message to the chatroom. Beforehand it was known how the response from the server would look, and how the message would be modeled within a class, but the server functionality were not yet

implemented. Hence it was necessary to use a stub to mock the corresponding message object received from the server.

Additionally, as the number of integration test was limited, the functionalities have been tested by manual testing during development. Here the integration of the different functionalities were observed and verified to work as expected.

8.3 Usability

8.3.1 Preliminary to performing the test

A usability test involves the customers and users of the system. It tests whether or not the program is intuitive and works as intended in a real life scenario. Usability tests revolves around user interaction, and a questionnaire examining the user experience. To ensure a fulfilling usability test the success criteria should be established beforehand, which for this project is:

- Time, which is compared to a baseline found defined by the average time for developers to complete the tasks
- Number of missteps
- If the tasks is completed

Additionally, to support the measurements the testee is asked to fill out a System Usability Scale (SUS) questionnaire [4,p. 112-116]. The tasks that the testee are to complete is presented in table 8.1. When all tests are completed, the combined SUS score and the previously mentioned measurements will be evaluated upon to establish whether or not the program passes the usability test.

Tasks		Guide
Member		
№1	Login	-
№2	Ask a question in the Q&A section	After you have logged in, navigate to the menu you believe to be the Q&A and then add a question
№3	Send a message	Navigate back to the main menu and find the menu you believe to be the chatroom and try to send a message
№4	Open a blog post	Navigate to the blog menu and choose a blog you would like to read
№5	Vote on a poll	Navigate to where you think you'll find polls and vote on one or multiple answers
№6	Find a running program	Navigate to the section you believe would hold the running programs, and open one of your own choice
Instructor		
№7	Answer a Q&A	Navigate to the Q&A section and answer it
№8	Comment a Q&A	Navigate to Q&A and leave a comment
№9	Create a poll	Navigate to the poll section and create a new poll

Table 8.1: Usability tasks

8.3.2 Results

The test measured time, missteps and whether or not the test case was completed. As for time, the *baseline* is defined as the average time of the developers per task. The *baseline* is used to compare with the testees results as to get an indication of where there might exist an issue. This measurement should help highlight where one might have to examine the users interaction to a greater extend. Furthermore, the testees behavior and feedback is taken into consideration and should be weighted higher than the actual time of the test.

The results is sorted into three tables, the first being table 8.3 which represents the results of the group members. All results are measured in seconds and the number in the top row corresponds to the task of same number in table 8.1. Furthermore it should be noted that the vertical lines splits the task, so that the first group of columns is member tasks, the second group of columns is the instructor tasks and the final group is various different

results. In table 8.4 the result of the test persons is presented. The third table, table 8.5, is the baseline used for time comparison.

Examining the results of the test, two tasks stands out as not acceptable. The two tasks are *№2 ask question* and *№5 vote on poll*. The first failed task *ask question* was a result of testee 1 thinking that a question, in the app, were the button one should press to ask a question. When asked why, they explained that as it did not say "test question" anywhere they got confused and mixed it with the actual "add question" button, hence the time. As for the second task *vote on poll*, testee 2 were confused as to whether or not their vote was accepted as no feedback were given. Furthermore, testee 1 also found the lack of feedback when voting on a poll confusing. Another issue found within the poll feature, was for the instructor task *№9 create a poll* completed by testee 1. Here they found it counter intuitive that the app did not automatically focus on the input field. Moreover both testees did not recognize that each poll title worked as a button in itself as there where no borders or other visual aspects pointing towards it being a button. Following the observations, multiple visual updates where implemented to the poll component, such as feedback on voting consisting of a small text message indicating that the vote was successful, and borders around all individual elements. As for the SUS scale the result can be seen in table 8.2

Testee	**SUS Score**
Testee 1	80
Testee 2	83

Table 8.2: SUS Scores

The SUS score ranges between 0 and 100, where 0 is worst and 100 is the best. It is said that the average SUS score is 68. As our results is 15 above average the test is said to be successful, further backing the that the program passes the usability test [10].

An important observation from the usability test is that the feature containing the most usability issues, being polls, is the only feature without a hi-fi prototype and also the only feature with a prototype that were not presented to Sportstiming hence no feedback were given on it.

	№1	№2	№3	№4	№5	№6	№7	№8	№9	Time	Missteps	Completed
Group												
Alexander	15	10	8	16	9	21	7	12	13	111	-	-
Casper	38	34	13	17	17	13	19	17	15	183	-	-
Christopher	25	15	29	16	37	19	16	8	20	185	-	-
Dion	15	21	20	11	8	23	10	10	17	135	-	-
Mads	28	25	12	12	13	14	18	15	39	176	-	-
Villiam	11	7	7	10	12	15	16	8	20	185	-	-

Table 8.3: Usability test results in the project group. This is used to form the baseline. Times is measured in seconds and the number refers to the task in table 8.1

	№1	№2	№3	№4	№5	№6	№7	№8	№9	Time	Missteps	Completed
Testee												
Testee 1	34	50	7	15	21	23	11	11	27	199	2	yes
Testee 2	20	25	17	13	32	15					4	yes

Table 8.4: Usability test results. Time is measured in seconds and the number refers to the task in table 8.1

	№1	№2	№3	№4	№5	№6	№7	№8	№9	Time	Missteps	Completed
Baseline	**22**	**19**	**15**	**14**	**16**	**17**	**14**	**12**	**22**	**146**	**-**	**-**

Table 8.5: Baseline for the usability test. Time is measured in seconds and the number refers to the task in table 8.1

8.4 Other test methods

When developing software one encounters numerous ways of testing their software, and therefore the developer should carefully consider what testing methods to use. As of this project; unit, usability, integration, acceptance and end-to-end testing is performed, but other test methods should also be considered. The test methods that were considered will be discussed in the following sections.

8.4.1 Mutation testing

Mutation testing is meta testing and revolves around testing your own test cases, that is testing if your test actually identifies the correct errors. This is done by mutating some of the program to simulate an error and hereafter the developer should check if the tests identified and isolated the error.

8.4.2 Performance testing

Performance testing is a generalisation which consist of different kinds of tests. One of them is stress testing where the system is put under realistic and unrealistic load cases which will help the developer to see where the program will break under immense stress. As Sportstiming did not set up any demands regarding performance of the application and gave no insights as to under what load the application will be, performance testing is not done.

8.4.3 Compatibility testing

Compatibility testing is used to test how your system will perform under different environments e.g. an operating system. However, as it was stated in the *workflow* section in chapter 7, the application would only be run and tested on android devices as no one had access to test on an iOS system.

Part IV

Evaluation

9 | Discussion

Throughout the process of creating the application, multiple challenges arose and discussion about how to overcome said challenges were had. The more important discussions will be presented in this section. The following sections will discuss whether or not the application was successful, how the development of the product and project unfolded and how the strategy of the project changed. Additionally, problems related to tests will be discussed.

9.1 Fulfillment of requirements

In section 2, it was stated that all *Must have* requirements must be completed to consider the application delivery successful. The following sections will discuss whether the requirements in the *Must have* category are fulfilled by the application.

Chatroom

The application implements a chatroom where a user can send and receive messages. The application, however only contains one chatroom and not the ability to create additional chat rooms. The customer stated that the application should have one common chatroom for all users and that every user should be able to send and receive messages. Therefore the *Must have* requirements, from the MoSCoW list for the chatroom are fulfilled.

Programs

Sportstiming's main requirement to the programs, were that the user should be able to download and see all programs in-app. Sportstiming also had an additional requirement, that the programs should be opened in-app to keep users on the app.

However, this was not possible as the programs were not made accessible from the web-service until late in the process. Due to time constraints, it was decided that the PDFs should be opened with the device's default PDF viewer. This means that, whenever the user wants to view a training program they are redirected from the app, which is undesirable as Sportstiming wants their customers to stay on their service. Despite not being able to open the PDFs in the app, the users are still not required to login again, which negates the inconvenience experienced. Furthermore, the primary requirement from Sportstiming was that the user should be able to download and see programs, hence the main requirement is still deemed fulfilled as the primary demand was fulfilled.

Polls

The requirements for polls, was that instructors should be able to create polls, and that members should be able to vote on the created polls. When a user opens the Poll section of the app, a check to see if they are a instructor or just a member. if they are an instructor, a button to add polls is available, allowing the instructor to create a poll. If, however a member opens the section, they do not have this button and is thereby not able to create new polls. Both members and instructors can see the available polls, and both can open a poll, and see the choices available and choose the choice they want to vote on. With the functionality implemented, the *Must have* requirements have been fulfilled.

Q&A

The requirements for Q&A, is that members should be able to ask questions, instructors should be able to answer them, and lastly users should be able to see all public questions and answers. When a user opens the Q&A section, a check to determine whether or not the user is an instructor is performed. If they are an instructor, they will be presented with a list of question that users have asked, but they will not be able to ask their own questions, as it is counter intuitive. The instructors can comment on each question and they can make an answer to the question. Users that open the Q&A section will be presented with questions other users have asked, and with the option to create a new question. Users can make comment on their own question, but they can not answer it. Both users and instructors can see the comments and the answer to a question. With the functionality implemented, the *Must have* requirements have been implemented, as well as the *Should have* requirements, therefore the Q&A section can be considered to fulfill the requirements for a successful delivery of the application.

9.2 Usability

As Covid-19 caused disruption in our day-to-day lives, it also affected the way a usability test could be performed. This is due to the current lockdown, hence forcing us find new ways to test our application. One such way is through online platforms like Microsoft Teams, which was chosen for performing a usability test with Sportstiming, however yet another obstacle occurred, as the university does not allow for outsiders to take control of a university account's shared screen, resulting in Sportstiming not being able to try out the app. Therefore no usability test, but only acceptance testing, could be performed with the company contact. Furthermore, as a result of both Covid-19, the fact that Club Sportstiming is a new initiative from the company, and coupled with the problems regarding Microsoft Teams, it was not possible to test on the actual customer base, resulting in testing within the group-members' close contacts. As this is the closest we can get to a representative result, it is accepted as a correct result, but should still be looked upon with some skepticism.

With the execution of the test being less than ideal with few test subjects and unreliable data, one might raise the question of how this was not given more thought. The answer would be inexperience as the group thought of it as a lesser task, when in reality it is a vital task, and should be given more time, when preparing it.

9.3 Process

To alleviate the high amount of uncertainty, frequent meetings with Sportstiming were planned. The meetings often revolved around prototypes and implementation technicalities, which helped clarify design and implementation choices. Furthermore the group changed the construction of sprints, as working on one cluster was too small a task for six people. Instead the group worked on different tasks, depending on the urgency of the task. These frequent customer meetings became a vital part of development and greatly improved our understanding of customer cooperation and the importance of it.

9.4 Scoping

From the beginning of the project we had a clear vision of how the project would end, what requirements would be completed and how difficult it would be. That perception, however, has changed as the project progressed. First of all, none of the group members

have any experience with app development, which proved a bigger challenge than was assumed, resulting in the expectations of our capabilities from Sportstiming being greater than what it should have been. This had the effect that while all the *Must Haves* and a couple of the *Should Haves* were implemented, no *Could Haves* were implemented.

9.5 Test

Test is a crucial part of ensuring a complete program, as without test there is no way of proving that your program works. Therefore it is a crucial error that the application does not implement more tests. As of unit test, only a small amount of the application has been tested. With integration testing there have not been any automatic tests but only manual tests, which does not provide the same amount of insight as a proper code test.

The unfulfilling amount of testing is a result of lack of knowledge, as the group did prioritise for a lot of testing, but as the project progressed there were not enough time for testing. In future projects the group will strive to achieve better prioritization of testing and implementation.

10 | Conclusion

The project set out to create an application for Sportstiming, which is to be used as a premium member feature. The collaboration between the group and Sportstiming was based on shared understanding of the desired product. This understanding was elicited by numerous meetings and discussions with Sportstiming, which culminated in the MoSCoW list in section 2. The success criteria for the product development was to fulfill the *Must have* requirements detailed in the MoSCoW list.

The project set out to generate and systematically evaluate classes and events, resulting in the problem domain model and several use cases. These results would later be used to create the model and function components establishing a greater understanding of the system. Lastly the analysis reflected the architectural patterns of the system and why these patterns were chosen.

The MoSCoW list identified the responsibilities and structure of the application, and to confirm the understanding of the requirements, hi- and lo-fi prototypes were used. The two in conjunction, meant that it was possible to proceed in the project with a higher degree of certainty that Sportstiming's requirements were correctly understood and that no misunderstandings had taken place.

In section 9.1, the MoSCoW list is contrasted with the application to determine the accuracy of the implementation, and the conclusion derived, was that all the *Must have* requirements were fulfilled. This was also corroborated by Sportstiming expressing that the application had indeed fulfilled all of the *Must have* requirements. It is therefore concluded that the delivery of the application is a success.

However, there are still parts of the project that were accomplished to a lesser extent, particularly the lack of testing as described in section 9.5. There are demonstrations of unit testing and integration testing to a degree, but not to the degree that was sought in the beginning of the project. It is therefore an ideal place to improve on the project.

The project did, however, see quality assurance in the form of usability tests and the

improvements implemented based on the evaluation of said tests.

Considering the successful delivery of the application, combined with the experience gained while working with Object Oriented Analysis & Design. The group is content with the result of the project as a whole, while acknowledging the outlined shortcomings.

11 | Future work

The app was developed for Sportstiming as part of their new premium member service. Therefore a lot of opportunities lies ahead for further development. First of all, the requirements from the MoSCoW list (section 2) that were not fulfilled during this project is an obvious choice for further development. Furthermore a lot of in-app functionalities like payment, creating your own programs and even integration with Sportstimings primary app is a possibility.

A functionality which could prove beneficial, is a system to add, remove or change routes during run-time. This would allow for less downtime if Sportstiming want to add additional routes, as a server restart would not be needed new routes to be usable.

Also, many features within the app is largely untested as stated in section 9.5. Therefore, implementing tests for existing functionalities is one of the most important future implementations, in order to have an application that is guaranteed to work.

Bibliography

[1] Sportstiming. *Om Sportstiming*. https://www.sportstiming.dk/about, 2020. Visited 08-10-2020.

[2] Lars Mathiassen, Andreas Munk-Madsen, Peter Axel Nielsen, and Jan Stage. *Object Oriented Analysis & Design*. Metodica ApS, 2 edition, 2018. ISBN 978-87-970693-0-1.

[3] *MoSCoW method*. https://en.wikipedia.org/wiki/MoSCoW_method. Visited 16-12-2020.

[4] David Benyon. *Designing User Experience - A guide to HCI, UX and interaction design*. Pearson, 4 edition, 2019. ISBN 978-1-292-15551-7.

[5] Balsamiq. *Balsamiq*. https://balsamiq.com/wireframes/, 2020. Visited 16-12-2020.

[6] ByPeople. *Cityguide: Mobile App Template*. https://www.bypeople.com/mobile-app-template/, 2020. Visited 03-11-2020.

[7] howtodoinjava. *What is REST*. https://restfulapi.net/, 2020. Visited 14-12-2020.

[8] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, UNIVERSITY OF CALIFORNIA, Irvine, CA, 2000.

[9] Microsoft. *.NET Documentation: SocketAsyncEventArgs Class*. https://docs.microsoft.com/en-us/dotnet/api/system.net.sockets.socketasynceventargs?view=net-5.0#examples, 2010. Visited 19-11-2020.

[10] usability.gov. *System Usability Scale SUS*. https://www.usability.gov/how-to-and-tools/methods/system-usability-scale.html, 2020. Visited 17-12-2020.

Appendix

A | Requirements - First meeting

A.1 Initial Requirements

Trough meetings and discussions with Sportstiming a number of requirements has arisen. To give a greater overview of the problem and possible solutions the requirements is separated into features and components. For a greater understanding a distinction between member and instructor is made, however both members and instructors are contained in the common definition *user*

A.1.1 Chatroom

First of all a common chatroom for all users is a key component in this system and therefore we will describe requirements regarding this component first:

- The system must contain one common chatroom for all users
- Any user can tag another user in the chatroom
- If a user is tagged they will receive a notification

Furthermore some nice to have features exist regarding the chatroom:

- Instructors should be noted by a badge
- Instructors should be able to create individual chatrooms
- Instructors should be able to delete messages
- Instructors should be able to pin messages
- A pinned message should have a timer for deletion
- It should be possible to create subchatrooms

A.1.2 Q&A

Another key component of the system is a Q&A for which a number of requirements are outlined:

- Only members can ask a question
- Only instructors can answer questions
- A question can only be commented on by instructors and the member asking the question
- Members can see all public questions and answers

Regarding Q&A one nice to have feature co-exist alongside the last requirement:

- A member can ask a private question, which cannot be seen by others

A.1.3 Polls

To give the instructors a better understanding of what the members want, they would like to have polls. The requirements for the poll component is as follows:

- Instructors should be able to create polls
- Members should be able to vote on poll choices
- Poll choices should consist of workshops being held at least a month in advance.
- All users should be able to add a poll choice after the poll has been created

A.1.4 Blog post

Sportstiming would like to send out news and articles trough a blog post, that will exist as an individual section. To fulfill this a requirement is established:

- The system must get all blog post from the website and display them in the app

One nice to have feature follows along:

- It should be possible to comment on blog posts

A.1.5 Notification page

Besides a blog post a notification page is required. However these features differs significantly in that a blog post is an article, training program or news in the industry

posted to a blog, but the notification page is simply a notification stating that a new blog post have been submitted, a training program have been created or another update within the app have happened. For this a different requirement is needed:

- The news section should automatically update with in-app news

A.1.6 Training program

To help members prepare for an upcoming event, Sportstiming offers a number of training programs, which all should be accessible from within the app, which creates the following requirements:

- All members can see and download all training programs
- There must be a section for training programs

A set of nice to have features follows:

- A section for specific event training programs is wanted
- It should be possible to view videos of an exercise in the app

A.1.7 Workshop

Lastly, Sportstiming would like the possibility to arrange and add a workshop as preparation for an event. However this component is nice to have. This presents the requirements:

- Workshops can be arranged by instructors
- Workshops has a set cap of attendees
- Workshops has a location
- Workshops has a date
- Members can sign-up for workshops

B | Usability test scheme

B.1 Usability test scheme

See electronic appendix 1

C | Use cases

C.1 Use cases

C.1.1 Q&A

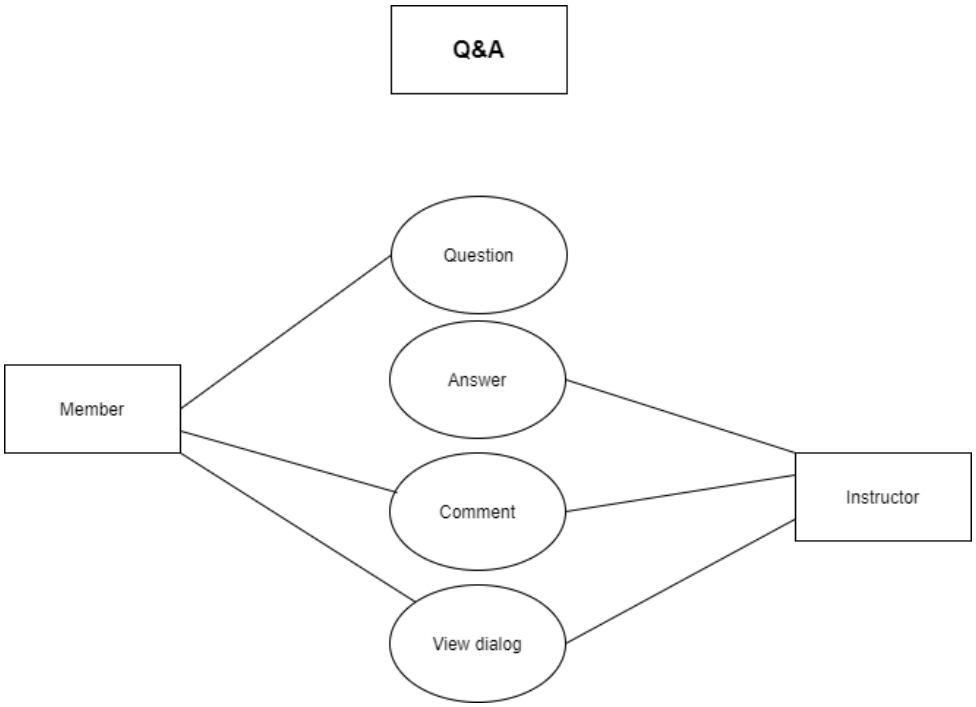


Figure C.1: Use-case diagram for Q&A

Question
Use Case: Question is initiated by the *asking member* writing a question in a text field and posting it as either public or private. If it is a private question it is only visible to the *asking member* and the *instructors*, but if it is public it is visible to all members.
Object(s): Asking member, Public question, Private question.
Function(s):

Table C.1: Asking a question

Answer
Use Case: The answer is conducted by an *instructor*. The instructor clicks on the text field and writes out their answer. Hereafter they check the "Answer" box and submit their answer by clicking the "Send" button or pressing enter.
Object(s):
Function(s):

Table C.2: Answering a question

Comment
Use Case: Commenting is done by either a *asking member* or an *instructor*. The *instructor* or the *asking member* writes out their comment in the text field and submit their comment by clicking the "Send" button.
Object(s):
Function(s):

Table C.3: Commenting on a question

View private dialogue
Use Case: View private dialogue is initiated by an asking member or an instructor clicking on a private question. This will display the answer and comments related to the question.
Object(s):
Function(s):

Table C.4: Viewing a Q&A dialogue

View public dialogue
Use Case: View public dialogue is initiated by a member or an instructor clicking on a public question. This will display the answer and comments related to the question.
Object(s):
Function(s):

Table C.5: Viewing a Q&A dialogue

C.1.2 Blog

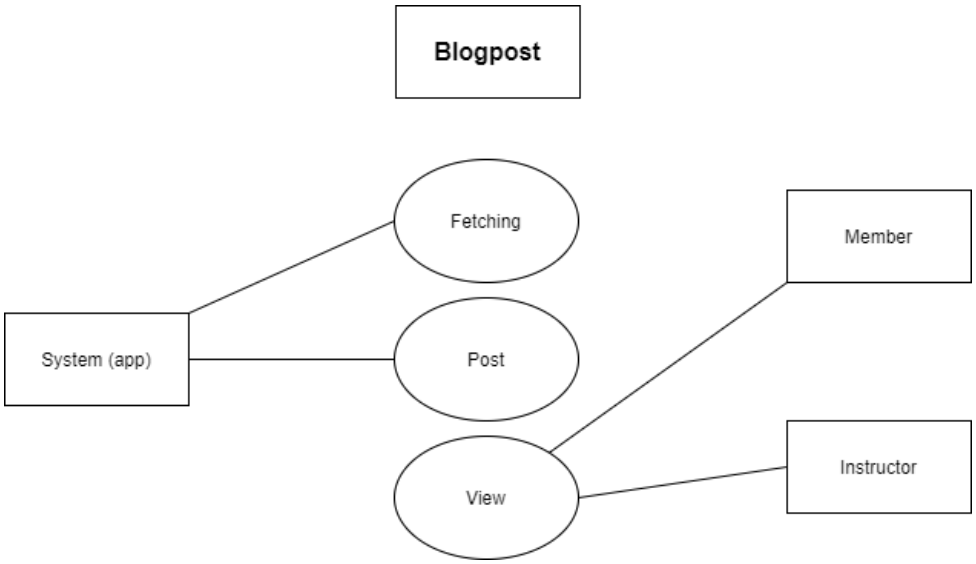


Figure C.2: Use-case diagram for blog post

Fetching
Use Case: Fetching is done by the back-end system. Firstly the system sends a request to a web service. The web service then responds with a blog post taken from the Sportstiming website.
Object(s):
Function(s):

Table C.6: System fetching blog posts from a web service

Post
Use Case: The system takes a blog post retrieved from the web service and displays it in the app. **Object(s):** **Function(s):**

Table C.7: System posting a blog post in the app

View post
Use Case: The *user* selects one of the posts by pressing it. Hereafter the full blog post will be shown to the *user*. **Object(s):** **Function(s):**

Table C.8: Viewing desired blog element