Andreas Berre Eriksen, Lars Emil Eriksen
Jens Emil Gydesen, Mikkel Alexander Madsen
Nichlas Bo Nielsen & Ulf Gaarde Simonsen
D301E12@cs.aau.dk

December 20th - 2012

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Computer Science
Cassiopeia
Selma Lagerlöfs Vej 300
Phone: +45 99 40 99 40
Fax: +45 99 40 97 98
http://www.cs.aau.dk

Title:
BogoBeats

Topic:
Developing applications - from users to data, algorithms and tests - and back again

Project period:
P3, Autumn semester 2012
3. September - 20. December

Project group:
D301E12
d301e12@cs.aau.dk

Members:
Andreas Berre Eriksen
Lars Emil Eriksen
Jens Emil Gydesen
Mikkel Alexander Madsen
Nichlas Bo Nielsen
Ulf Gaarde Simonsen

Supervisor:
Xike Xie

Total page number: 127
Ended on: 20. December 2012

Abstract:

In this project we worked on creating a social music service where users could vote for songs they'd like to hear, making the music at parties democratically chosen, where we also recommend songs for a party by finding similar parties and recommending songs from those parties. We used an iterative work process for this P3 project, creating multiple prototypes. We developed this system with the help of two informants, giving us ideas for the system and by testing the prototypes we presented. The system uses a; website written in HTML, using PHP and JavaScript for functions, and a framework called Bootstrap for the graphical part; a MySQL database for storing data about songs and users and an Android application written in Java, for voting and administering music.

Preface

This P3 project is developed by 6 computer science students at 3rd semester at Aalborg University. The project started at September 3, 2012, and ended at December 20, 2012. The topic for the project was "Developing applications - from users to data, algorithms and tests - and back again" with the goal that we students learn to develop and implement a large system, and ensure the quality of the system using informants and system tests. The conditions for this project was as follows:

- Problem to be addressed by the project must be thoroughly formulated.
- Requirements should be investigated in a realistic setting involving 2 or more informants none of which are students or lecturers. The purpose is to investigate a context for a new application with which the students are unfamiliar.
- Requirements models, designs, and program must be object-oriented.
- Requirements should be explored through interviews with potential users and through prototypes evaluated by potential users. Requirements should thus be documented with both object-oriented models and with prototypes.
- Designs should include both software architecture and interaction designs.
- Program should be with all necessary parts and relevant data.
- Tests should cover the program to a reasonable and explainable degree.
- Interaction design should be explored through prototypes and usability should be evaluated systematically.

We decided to work with a social music service after talking with several of the available supervisors.

Our report consists of 2 parts; a development report (Part I) and an academic report (Part II). The development report contains the documentation of the project, while the academic report contains a description of our experiences working with this project. During the development report, we will refer to cited sources, which will be marked using square brackets. The bibliography can be found at the end of the development report, followed by the academic report and at the end the appendix. Our appendix will primarily consist of images and code used for testing, as well as some of the test results. The source code for the system can be found online at Bogosongs.com/SourceCode.zip or at http://goo.gl/8hxh5, with a reading guide, all compressed to a Zip-file. The project includes a website: Bogosongs.com, used by the system. A glossary with words that might not be well known or have special meaning in our project can be found after this section. The report is made using LaTeX. All the links for the sources used were working at the project deadline December 20 - 2012.

At last we would like to thank our supervisor Xike Xie for supervising us, and our two informants Rikke Boel and Ida Gaarde Simonsen for helping us during this project.

Glossary

The following is a list of words used in the report, that might not be well known, or have special meaning in this context.

App/Application An application (or app for short) is a piece of software run by a smartphone (although it may also be used for a program on other platforms)

Webservice A webservice is an application hosted by a remote system, often executable through a web browser

Bookmarklet A bookmarklet is a small application stored by a bookmark in a web browser

Platform A piece of software in which applications can be executed, typically an operating system like Windows, Mac OS, Android, iOS, etc.

Signatures

Andreas Berre Eriksen
aeriks11@student.aau.dk

Lars Emil Eriksen
leriks11@student.aau.dk

Jens Emil Gydesen
jgydes11@student.aau.dk

Mikkel Alexander Madsen
mama11@student.aau.dk

Nichlas Bo Nielsen
nnie11@student.aau.dk

Ulf Gaarde Simonsen
usimon11@student.aau.dk

Contents

List of Figures

List of Tables

Part I

Development Report

Chapter 1

Introduction

In this chapter we describe our initializing problem in Section 1.1. Since there are already some existing services dealing with our problem, we will analyze these to see what they do well, and where they can be improved, which we will do in Section 1.2. We will then in Section 1.3, collect our knowledge and define a research question. Section 1.4 describes our work method during this project, and Section 1.5 gives a quick overview of what will be in the following chapters.

1.1 Initializing problem

Music is a huge part of many people's lives. There are very few people, if any, in the world who doesn't know any music that they like. The amount of different music genres is enormous, and how each genre sounds can differ dramatically. Because of this, people's tastes in music are often very different and this can cause problems. From personal observations, we've noticed that when a lot of people with different music tastes are gathered, it can often be very difficult to decide what kind of music should be played. This can lead to the music changing a lot, even before songs are finished. This is because some people do not care if others put a song on the playlist, when they want to hear the music they like. It's safe to assume that it will happen more often when people are under the influence of alcohol. It can also lead to songs playing, which only a few people wants to hear. It can also be annoying for the host of the party to be responsible for the music, and have to spend a lot of time to make sure that there is always music playing, and that it is something the majority of guests would like to hear. To find a solution to these problems we first need to specify a target group who could benefit from a solution.

The target group for our solution will primarily be people between the age of 15 and 40, because this is the group which is most often found at parties (we assume). We will use informants who are in our target group to help us develop a proper solution. We decided not to use quantitative questionnaires as we preferred having detailed answers from a few people, rather than short answers from a lot of people.

We will describe each informant by their demographics and by their needs. We used their needs to help us determine which part of the problem is the most problematic from our informants' point of view, and hence our target group. As our target group is rather broad, we decided to go with one from each end, that is a teenager/young adult, and an adult. We interviewed each informant several times, each time testing a new version of our solution.

The following information about the informants was gathered when we had our first usability test (describes in Section 6.1).

Ida Gaarde Simonsen

- Name: Ida Gaarde Simonsen
- Age: 17
- Party frequency: 1-2 times a month
- Party size(participants): 30 to several hundred , typically more than 30

Our first informant is Ida. Ida is a teenage girl, attending a Danish high school. She attends parties a couple of times a month, most of them are high school parties. These parties are usually larger than 30 people, and sometimes up to a few hundred people. The music is played by a computer using Spotify (an online music streaming service), where they usually make a playlist beforehand, and then plays this offline playlist at the party. She rarely experiences people changing song before the current song is over. She doesn't see it as a big problem. She liked the idea of not having to get up (and go to the computer) to change song (to remotely control the music).

Rikke Boel

- Name: Rikke Boel
- Age: 37
- Party frequency: Maximum once a month
- Party size(participants): 12-15

Our second informant is Rikke. Rikke is an adult, working as a volunteer at Skråen, which is a music association that arranges concerts, shows and events. She doesn't go to parties as frequently as most young people, but she still attends parties rather often. At the parties she attends there's usually 12-15 people attending. At these parties they often use an iPod or an iPad connected to speakers to play the music, and sometimes they use playlists on PCs. She often experiences people changing music while the current song is playing, and finds it annoying, especially when it's a song she wants to hear that is being skipped. An interesting note from this first interview, is that she said if the music stops, the party stops. In order for a party to take place, a constant flow of music is needed.

1.2 Existing Services

In this section, we review existing services and analyze their functionality as a social playlist service. Each service will be briefly described and analyzed in a pros/cons table. Based on these pros/cons tables, we highlight our proposals to a better system.

1.2.1 TuneTug

TuneTug as described from the website:

> "TuneTug is a new service that lets everyone be the DJ at parties and events. TuneTug allows your guests to "Tug" up and down songs with their mobile phone that they want to hear. To set up TuneTug you need an internet connected iPhone, iPod or iPad - WiFi is suggested but not required. Users on most any type of mobile device can vote and see the playlist live. TuneTug Voting has been tested on recent smartphone devices including: iPhone, Android and Blackberry." [18]

TuneTug as an application is only available on the iOS platform, which limits its user-base. Music can only be played in the application with the built-in music player, or through Spotify with a premium account for 99DKK a mouth [17]. The playlist is either run through Spotify, iTunes with a local library, or by combining these two into a new playlist. TuneTug also has a webservice which can be run on any smartphone with a web browser.

You can log in through Facebook, or create a quick name. You can "Tug down" or "Tug up" a song to push this song on the dynamic playlist. Tugs neutralize each other, meaning that 1 "Tug up" + 1 "Tug down" = 0. The one with highest rating is the next song to be played. The UI is basically a playlist, where you can see the song played at the top, and two buttons "Tug up" and "Tug down", as seen by Figure 1.1 and Figure 1.2.

In the iOS application the following can be configured:

- Party name
- Tugs per user (standard is 10), these are votes
- Update party location via GPS
- Limit song plays
- Clear votes
- Participants must register (yes or no)
- Anyone nearby can join party (yes or no)

We will now shortly analyze the pros and cons of TuneTug. An overview of the pros and cons can be seen in Table 1.1.

You're able to vote through the web application, which allows all users to vote, and not just those with iOS devices. The host can control the privacy of the party. He can make the party public, giving easy access to everyone nearby, or lock it with an ID, which the guests needs to get before they can vote. The host can also choose whether or not the guests have to register first. If the host has a premium Spotify account, he can stream the music through that, giving the guests a lot of music to choose from. This does, however, cost 99DKK a month to have (per December 2012 [17]), otherwise he will have to own the music himself, and stream it through his iTunes library. The host will also have to own an iOS device himself to use the native application.

Pros	Cons
Web application for voting	Platform dependent
Parties can be public or locked with an ID	Have to own the music or pay
No registering required	
Easily set up	
Spotify connectible	

Table 1.1: TuneTug pros & cons

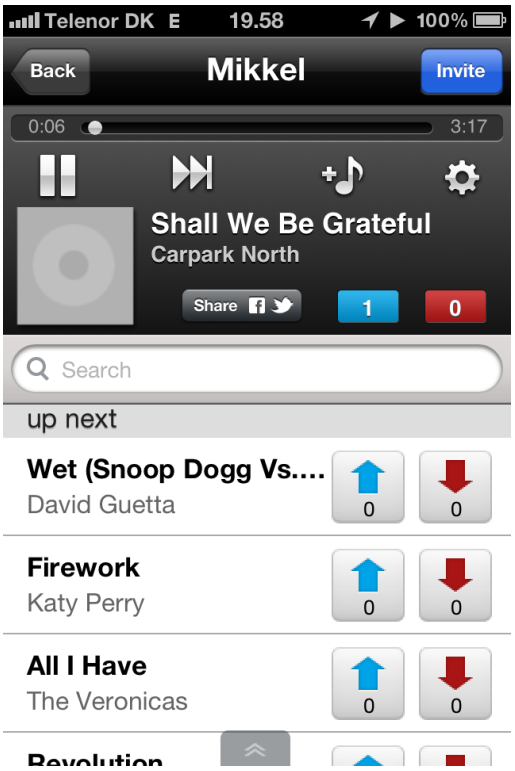


Figure 1.1: TuneTug native application

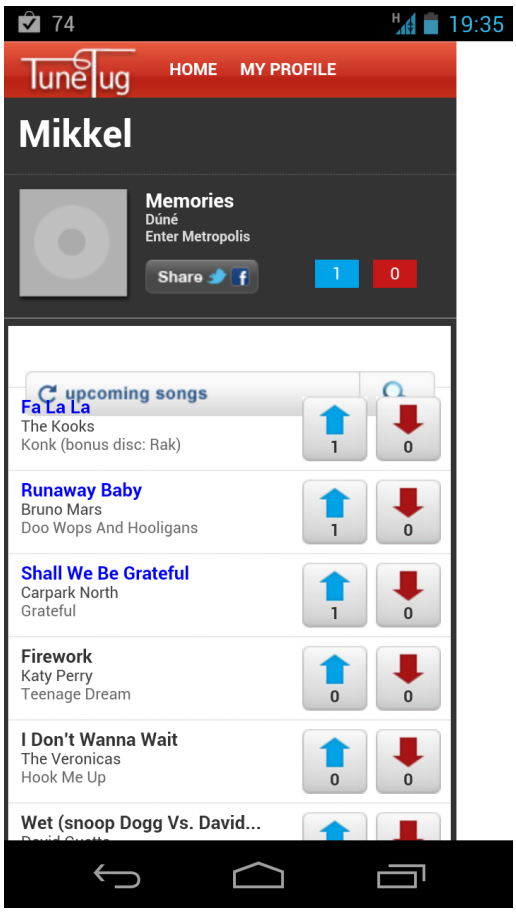


Figure 1.2: TuneTug web application

1.2.2 djtxt

djtxt about their service:

> "Throwing a party? With djtxt you can set up a crowdsourced party playlist in one click. Your party guests send in song requests by text message, and djtxt plays their tunes. You'll see who added each song, and you can look at a recap of the playlist after the party." [9]

djtxt is a webservice where the music can be controlled through either Twitter, email or SMS. Instead of voting, songs are played if requested by an attendee. djtxt uses Grooveshark, a webservice used to stream music from a large collection of music. To start streaming music from Grooveshark, the host has to use a unique bookmarklet for his party (created by djtxt), which creates an overlay on Grooveshark's website. The users can then send messages to djtxt, which then handles song requests and adds the requested songs to a Grooveshark playlist. If the playlist is empty at some point, djtxt automatically adds a random song, to make sure that there's always music playing.

The pros and cons of djtxt will now be analyzed, and an overview can be seen in Table 1.2.

Since djtxt can be controlled through SMS, email and Twitter messages, it's very platform independent. However, to use SMS, the party must be a premium party, which costs $2 an hour, and does only work in the United States. Since the service uses Grooveshark to play the music, they get access to the complete Grooveshark library, which holds a lot of music. However, the service will also suffer from whatever problems Grooveshark

might have. Grooveshark have been under legal problems, and has even been blocked by many ISPs (Internet Service Provider), for example in Denmark [16]. This means that if Grooveshark has been blocked, djtxt will not work.

When you add a song through djtxt, it will simply add the first one on the list of songs it finds by searching Grooveshark. This may lead to unwanted song requests, as you cannot pick the right song from the search.

Pros	Cons
Platform independent	Requested songs are always played
Requested songs are always played	Basically Grooveshark
All of Grooveshark's music can be played	Songs requests can be misinterpreted by service
Everyone can participate	
Easy to use	

Table 1.2: djtxt pros & cons

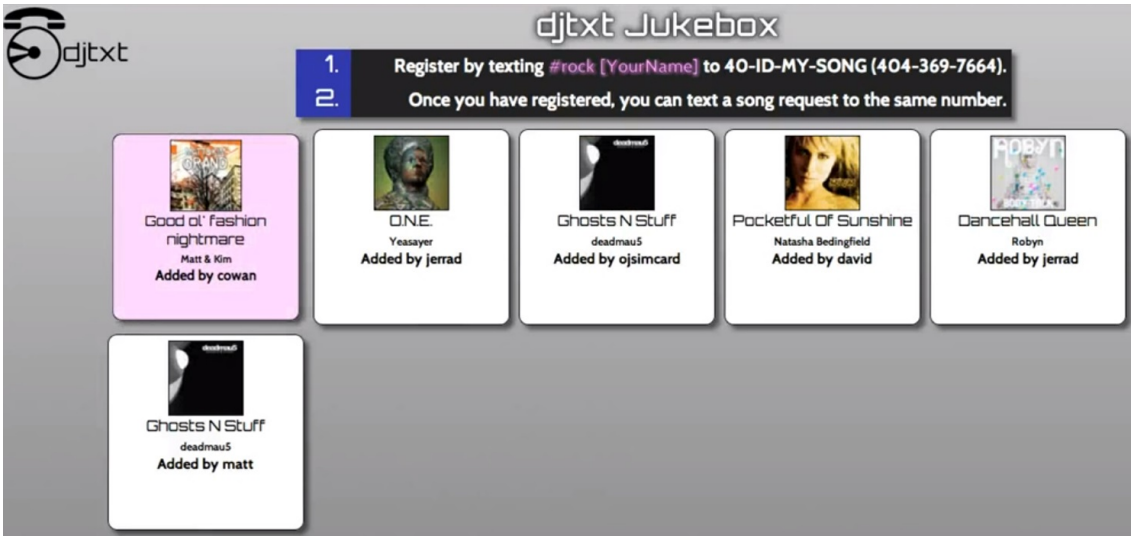


Figure 1.3: djtxt Grooveshark overlay

1.2.3 Our Proposals For a Better System

We will now use the knowledge gathered from the existing services to see what we can use from their systems, to create a better system. We really liked the idea of voting from TuneTug, although we were not sure if we wanted to have a vote down button. We didn't like the idea that the host had to have the music or pay before he could host a party. For this we wanted to go with djtxt's solution, and use an external music service like Grooveshark. We also liked the idea of a platform independent system like djtxt, which is something we wanted to do. One thing neither of the existing services had, was a suggestion or recommendation system for the users to find and explore new music. We feel that such a system would help users find music and give a better music experience at parties.

To sum up, we wanted to use a voting system that is platform independent, where the music is streamed online for free, and with this a feature to recommend music to the users.

1.3 Research Question

After analyzing the problem in context and reviewing existing services together with their solutions, we will define a research question. We had some interviews with out informants, which showed that there was an interest in such a system to be developed. The specific problems we noticed from the interviews were:

- Music was changed in the middle of a song
- The music was not always in the favor of the majority
- If the music stopped, the party stopped

From the collected information we defined the research question:

> *How can we develop a system which can help people at private parties listen to the music that they want to hear, without interruptions?*

Our novel features can be summed up to the following objectives:

- A democratic playlist, where music is chosen by the party attendees
- A free music playing service, from an external provider with a vast library
- A music recommendation service

1.4 Work Process

To answer our research question, we needed to find a work method. Since we were working with our informants, we could use their help on how to develop a proper system, according to their needs. To get the best out of this, we decided to use an iterative work method, working closely with our informants. The idea behind this method is to develop a system by going through cycles consisting of different steps as illustrated by Figure 1.4. As seen by Figure 1.4, the process consists of several iterative steps. Using this method, we first started with the initial planning, where we planned milestones and what had to be done in this project. We then started planning the first prototype test, and getting some basic information from our informants to set up some requirements for the system. We then made a prototype, tested the prototype on the informants, evaluated on what they said about it, and then began planning the next prototype which would be changed according to what the informants had said about the previous prototype. Near the end of the project, we had our last prototype test, implemented the last changes and deployed the final system.

Using this method makes it harder to plan the entire process of the development, as there can be many unforeseen changes you have to make, which makes it harder to know exactly what you're going to make, and when you're going to make it.

For this method to work, we had to make prototypes and test these on our informants. This meant that we had to spend time developing several prototypes, and spend time testing and evaluating on them. That did, however, also mean that we could better develop our system to our informants' needs, as they could tell us what exactly they would like to be changed in the prototypes, potentially giving us a better product in the end. During the development of our system, we made several prototypes, and tested each of those on our informants, which can be seen in Chapter 6.

This iterative process was new to us, which gave us all new experience. Our thoughts and what we learned from it can be seen in Part II. We chose to use this method because we first of all wanted to make sure our system would become a system that our informants would use, and to try a new development method.

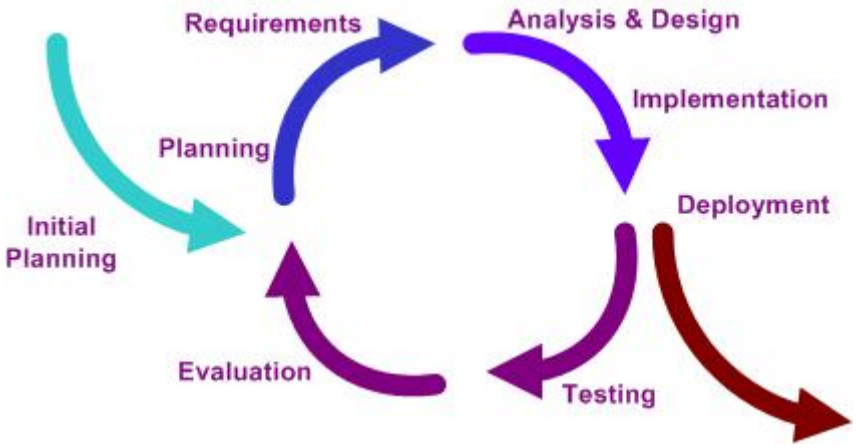


Figure 1.4: The iterative process [8]

1.5 Report Guideline

In this section a quick overview of the coming chapters are to be presented. The following chapters will not describe the prototypes, nor the process of making them. The following chapters will, however, describe our final system, which we implemented as our final product.

In Chapter 2 the development process of the system will be outlined. We will go through the graphical and algorithmic design process of the system in Chapter 3. Chapter 4 is a short chapter about system choices, such as platforms. The implementation of the system is described in Chapter 5. Chapter 6 is about the tests made during the project, both system and prototype testing. Lastly, Chapter 7 is the conclusion of the report.

Chapter 2

The System

In this chapter we will describe our system through various activities. We will start with defining our system in Section 2.1, to get a quick overview of what functions the system has, who is involved, the conditions for the system, what technology have been used to develop and to use the system, and also what the purpose of the system is. In Section 2.2 we describe our specification for our system in more detail. We analyze the problem-domain in Section 2.3, to better understand what information the system will have to deal with. We then analyze the application-domain in Section 2.4 to define requirements for the system's functions and requirements. Section 2.5 describes the architectural design of the system. In Section 2.6 we connect the information gathered, to form a final model of our system.

2.1 System Definition

The system is a democratic dynamically generated social playlist, where each attendee can vote for their favorite music, and save each song to a favorite list for easy access later. This is all done through a mobile application, while a website will administrate the parties, and an external music service will play the music through their website. The system is to be used by users with very different experience in using mobile applications and websites.

2.2 System Specifications

In this section we define the specifications for our system, and elaborate as to why these were chosen. Our specifications are defined as *hard constraints* and *soft constraints*. *Hard constraints* are constraints that must be complied, as the system would not function without them. The *soft constrains* are luxuries, however they still matter a great deal for the product's functionality. This is a subjective matter, however this is what we believe is important. The constraints are primarily based on input from our informants, but also from our own vision.

Hard constraints:

- Music should always play, unless it is paused intentionally
- Party attendees shall not be able to change songs
- Have a library of music
- Have a playlist for music

- Can add music to playlist
- Being able to vote up music
- Being able to play music
- The highest voted music is played

The idea of our research is to develop a system which keeps the music playing without stopping, implying that the majority will get to listen to the kind of music they like. This means that if the playlist runs out of songs, the system should add songs to the playlist by itself that fits the party's music taste. We also would like the music playing to always be determined by the system, thus attendees shall not be able to change the music. Having a library of music allows users of the system to find music they like, same principle as going to a library and picking out the book you want. Having a playlist is also important as the playlist's function is to serve as a list of music that will be played in some near future, perhaps not in the order of which the music numbers are listed, but playing the highest voted song first. Thus the functionality of playing, voting and having the highest voted music first in the playlist is our priority, giving it a democratic feel.

Soft constraints:

- User uniqueness functionality
- A favorites list
- A list of previously played songs
- Connection with social media
- Party check-in
- Administrator permissions e.g. play/stop/pause/skip music
- Platform independent
- Option to have maximum songs on the playlist

User uniqueness functionality is very useful. It provides us with the ability to keep track of each user and have our system act differently depending on the user. The idea of a favorite list provides an arbitrary user the ability to quickly add his favorite songs to a playlist, while the idea of the previously played songs list is a way to make sure a given user can see the songs which have been played at a party, and possibly favorite them. Social media connectivity is a neat feature, given how popular social services are in this era, e.g. Facebook, Twitter, Spotify etc. The system could for example be integrated with Facebook's events, giving a Facebook user the ability to post his party information on Facebook. We think it is necessary for the administrator of the party to be able to skip music. After all it is his party, and stopping or changing the music *may* be needed. We want our system to be platform independent, which means that it should run on more platforms than just one. Having a platform independent system would mean that we could reach a larger user-base, allowing everyone at a party to use the system. We also think the host should have the option to control the amount of songs on the party's playlist. If everyone could add as many songs as they like, going through the playlist to look for songs you want to vote up, could take too much time.

2.3 Problem-Domain Analysis

In this section we analyze the problem-domain, to look for requirements for our system. The problem-domain is that part of a context that is administered, controlled or monitored by a system. We start by identifying classes in Section 2.3.1 which will result in an event table for our problem domain. In Section 2.3.2 we describe the relations between the classes in our system, in a class diagram. We then analyze the behavior and attributes of objects and classes in Section 2.3.3.

2.3.1 Classes

To model our problem-domain, we start by identifying classes and events in our system. We define a class to be a collection of objects. These objects share attributes, behavior and structure. An event is an incident involving one or more of these objects. We put our classes and events in an *event table*, to get an overview of the classes and their events. The result of this can be seen in Table 2.1. An * means that the event can happen multiple times, whereas a + means that the event can only happen once for each object of that class.

We found classes for the problem-domain by brainstorming. After evaluating these classes, we decided on the following classes, by analyzing their importance and relevance for the problem domain. We will briefly describe each class and event, to give a better understanding of how we found those classes and events.

Classes:

Host
: The host is the person who hosts the party. He has different privileges compared to the party's attendees, such as controlling the electronic devices playing the music, when the party starts and when the party ends.

Party
: The party is more abstract, but can serve as a collection of information, including who's hosting, attending, what music is available, location and so on.

Attendee
: The attendee is a person invited by the host to join a party. The attendee doesn't get to control as much as the host during the party, but can still add songs to a playlist, wish/vote for songs, start/stop the music and more.

Playlist
: The playlist is a collection of songs, which is created at the start of the party, and changes dynamically when songs are added. Whenever the music is paused/stopped, the playlist will also be paused/stopped.

Song
: A Song object is part of a playlist, and can be played at the party. People at the party can stop the music, and hence stop the song from being played.

Vote
: To get the right music at the party, attendees (and the host) can wish/vote for which songs should be played, so that everyone can enjoy the music.

As seen by Table 2.1, we have selected events that a system would have to monitor. These are found the same way as we found the classes.

	Classes					
Events	Host	Party	Attendee	Playlist	Song	Vote
Party created	+	+				
Party hosted	+	+				
Party started		+				
Attended party		*	+			
Left party		*	+			
Song added	*		*	*	+	
Song voted	*		*		*	+
Song played				*	+	
Music started	*		*	*	*	
Music stopped	*		*	*	*	
Playlist created	*		*	+		
Playlist deleted	*		*	+		
Host ended party	+	+	+			
Party ended	+	+	+			

Table 2.1: Event table

Events:

Party created
Whenever a host creates a party, a new party object is made. The host is only active as long as the party is active, and will be inactive whenever the party ends.

Party hosted
An event that occurs when the party starts which involves the host.

Party starts
This event occurs when a host object starts hosting a party. A party object can only be started once.

Party attended
Attendees can join a party created by a host. The attendee will only be an attendee as long as he is attending the party.

Party left
Happens when an attendee leaves the party, destroys the Attendee object since he is no longer an attendee at the party.

Song added
Both the host and the attendee can add songs to a playlist. This creates a Song object.

Song voted
Both the host and the attendee can vote for songs.

Song played
Whenever a song is done playing on the playlist, a new song will be played from the playlist, destroying the Song object.

Music started
This event occurs when the music starts playing. It is triggered by a person pressing "Play" on the music player.

Music stopped
Happens when someone pressed "Stop" or "Pause" on the music player during the party.

Playlist created
A person at the party can create a playlist for the party.

Playlist Deleted
A playlist can be deleted by a person at the party.

Host ended party
This event happens if the host wants to stop the party. There will no longer be any attendees when the party stops. This event triggers the "Party ended".

Part ended
This occurs whenever the party is ended. It can be triggered either by the host stopping the party, or when the party stops because of some other reason, i.e. if no one is at the party.

2.3.2 Structure

We will now take the classes from Section 2.3.1, and add structural relations between them. We will make an *object-oriented structure*, and describe the relation between the classes with three different structures: *Generalization*, *aggregation* or *association*.

Generalization The generalization structure describes a relation between two or more specialized classes, and a super class. The specialized classes (subclasses) inherit properties from the superclass. We show a generalization relation by a $\triangle$, and we express the relation as a "is-a" relation.

Aggregation The aggregation structure is a relation between two or more objects. We show multiplicity by showing a number next to the object. A number shows the amount of objects each object is related to. An * means that we have "many" objects related to it. "..." means that we have *x to y* objects. For example "0..*" describes zero to many. The aggregation relation expresses a "has-a", "is-part-of" or "is-owned-by" relation, and we show this relation with a $\Diamond$.

Association The association structure describes a relation between objects, much like aggregation, but does not define any properties of the objects. We express the association relation by "knows" or "associated-with", and we show this by a simple line. As with the aggregation, we can also have multiplicity between the related objects.

The result of these structural relations between the classes, can be seen in Figure 2.1.

As seen in Figure 2.1, the classes are identical to the ones from Table 2.1. At the top we have the Party class, which holds all the information about who is at the party and what songs are in the playlist, and how many votes these songs have. Every party can have several attendees, but at least one which is the host. A host, however, can be hosting several parties. Since the people attending the party can create and delete playlists, a party can have zero or many playlists active, each containing zero to many songs. Each song has a least one vote, since the person who put the song on the playlist, must at least have voted for it.

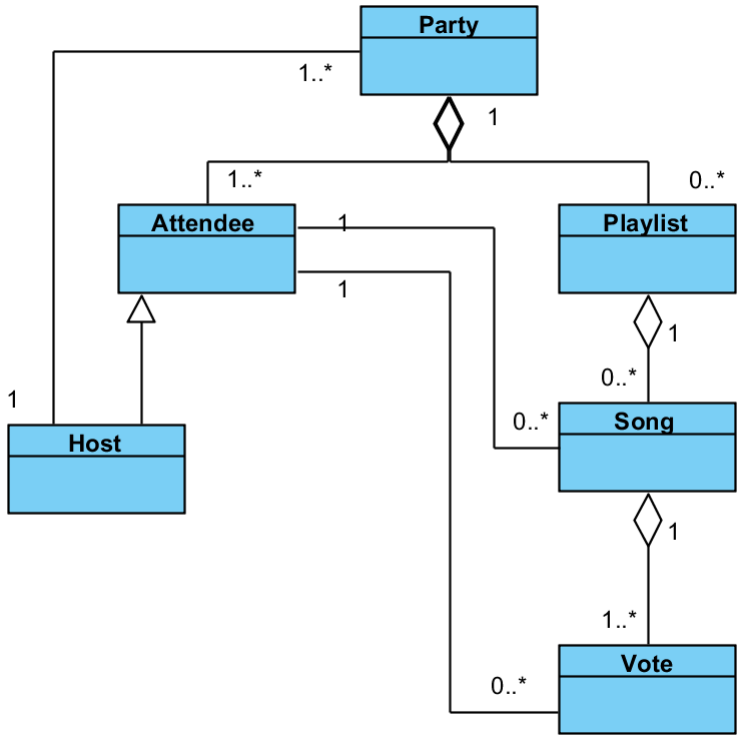


Figure 2.1: Class Diagram

2.3.3 Behavior

We will now describe the behavior of each class we found in Section 2.3.1. To describe the behavior of key classes in more detail, we make a behavior pattern analysis of each key class. We do this with statechart diagrams.

The first class to be analyzed is the Party class as seen by Figure 2.2.

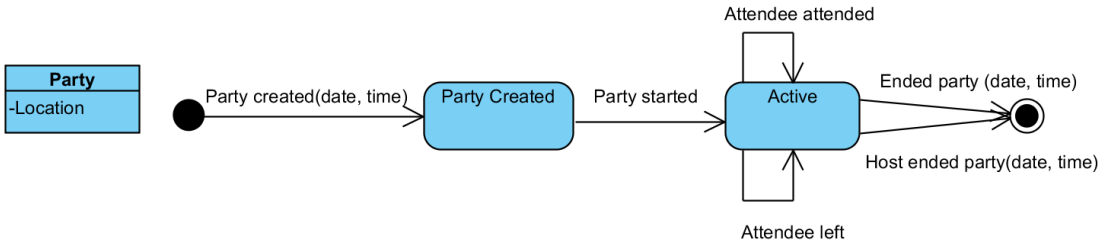


Figure 2.2: Behavior of Party Class

The Party class only has one notable attribute, which is the location of the party. When a party is announced, a new Party object is created, with a date and time for when it starts. Announcing a party does not necessarily mean that it starts right away. When a party is started, attendees can join and leave as they like. The party is active until the host ends it, or when it ends by something else, such as if everyone, including the host, left the party.

The Host class's behavior is illustrated by Figure 2.3.

Since people will always have names, so does the host. A host is not a host until he announces a party, and hence creating a Party object. However this will not make

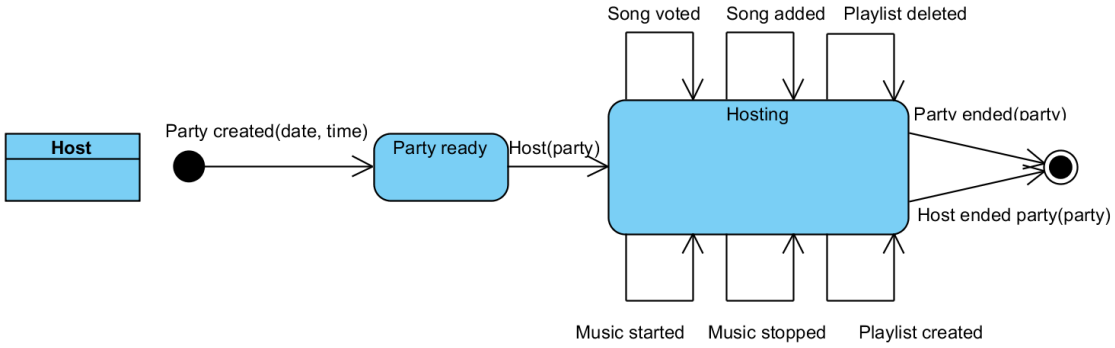


Figure 2.3: Behavior of Party Class

the host active, until he actually starts hosting the party, which happens when the party starts. During the party he is able to vote for songs, add songs, start/stop the music and create/delete playlists. He can choose to end the party at some point, which will destroy the Host object, since there's no longer any party. As mentioned before, the party can also end by something else.

Since the host isn't the only person at the party, we will also analyze the behavior of the Attendee class, as seen by Figure 2.4.

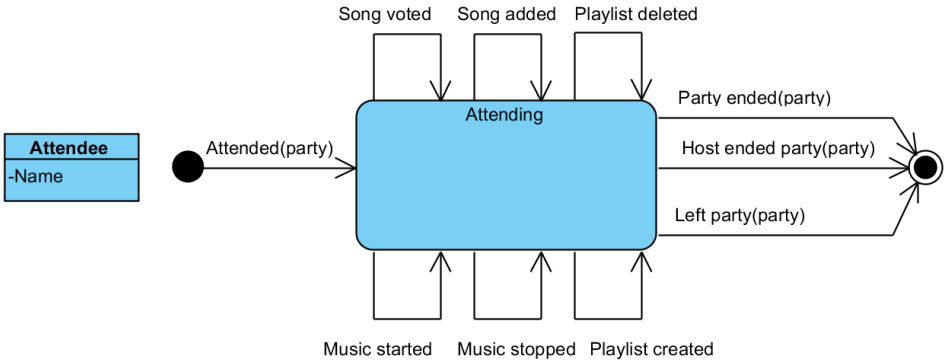


Figure 2.4: Behavior of Attendee Class

The Attendee class is almost identical to the Host class, except that an attendee can choose to leave the party, so we won't go into more detail with this class's behavior.

The next class to be analyzed is the Playlist class as seen in Figure 2.5.

The Playlist class's behavior is rather simple. It's active when it's created by an attendee or host, and destroyed when it's deleted. When its active, songs can be added to it, and it changes song whenever a song has been played. The playlist will be paused, but active whenever the music is stopped, and it will be playing whenever the music is started.

The last two classes, Song and Vote, are simple as seen by Figure 2.6 for the Song class, and Figure 2.7 for the Vote class.

A Song object is active whenever it has been added, and can then be stopped/started like the playlist, and it is then destroyed whenever it's done playing. A Vote object is active whenever a song has been voted, and gets destroyed when removed. A Song object can have multiple Vote objects active.

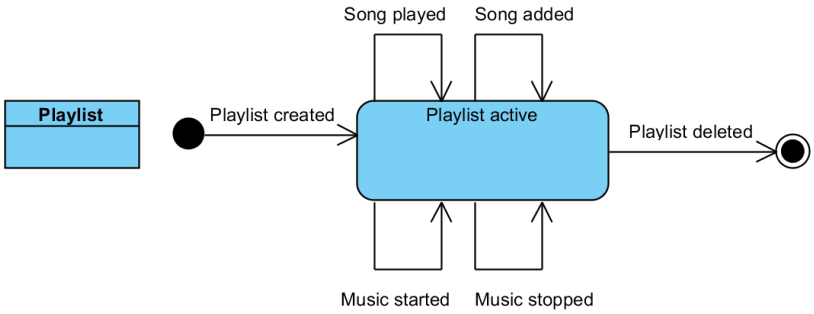


Figure 2.5: Behavior of Playlist Class

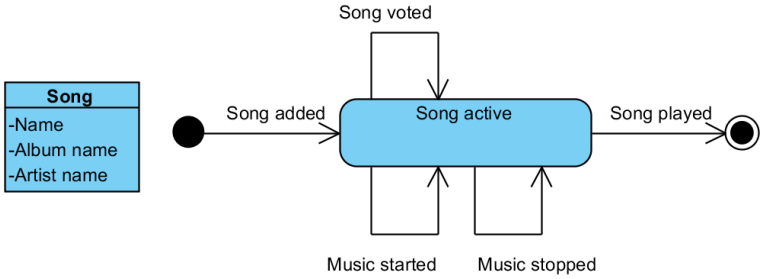


Figure 2.6: Behavior of Song Class

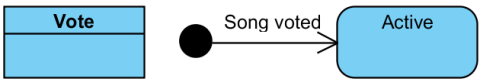


Figure 2.7: Behavior of Vote Class

2.4 Application-Domain Analysis

In this section we will analyze the application domain to determine the requirements for use of the system. The application domain is an organization that administrates, monitors, or controls a problem domain. We do this by identifying actors and use cases in Section 2.4.1. Afterwards we analyze the functions of the system in Section 2.4.2.

2.4.1 Usage

To determine how actors will interact with our system, we need to identify the use cases and actors in our system. We have done this with an actor table, as seen in Table 2.2. A ✓means that an actor interacts with the use case.

As seen by Table 2.2, our system only has two actors. We have a user of the system, which is the person who will use the system, and an external music service, which is a system that interacts with our system. To explain our actors in better detail, we'll describe each actor, and each use case.

	Actors	
Use Cases	User	External music service
Create Account	✓	
Create Party	✓	
Add Party	✓	
End Party	✓	✓
Delete party	✓	
Login/Logout	✓	
Start/Stop Music	✓	✓
Skip Song	✓	✓
Vote/Remove Vote	✓	
Music Search	✓	✓
Add Music To Playlist	✓	✓
Add Favorite Song	✓	

Table 2.2: Actor table

Actors

We will now describe our users briefly by Figure 2.8 for User, and Figure 2.9 for the External music service. Each description consists of three parts; goal, characteristics and examples. Goal describes the actors role in the system. Characteristics describes important aspects of the actor, i.e. if the actor is a person or another system. The examples describes different examples of each actor.

User

Goal: A person using the system. The user can create and administrate parties, and add or administrate the music in the system.
Characteristics: A user can be characterized as a host or an attendee. The host can do more than the attendee in the system, such as ending the party. Users have different experience with using systems, and like different genres of music.
Examples: User A likes to host parties, and uses the system often to create a party and invite his friends. During the party, A lets the attendees control the music, as he does not care much about what is being played.

User B is an attendee at a party. B is really into music, and wants to hear the music he likes at parties. B uses the system to vote for and add songs he likes, and is an very active user of the system.

Figure 2.8: Actor specifications for “User”

External music service

Goal: An external system to play the music from its library
Characteristics: The external music service has a player that can play music, and can be controlled by our system. The external music service has a large library of music, which can be accessed by our system.
Examples: External music service A has a large library of music which can be played by A's internal player. A can interact with other system using a public API.

Figure 2.9: Actor specifications for "External music service"

Use Cases

In this section we will describe our uses cases from Table 2.2. We will group some of the use cases, and show the result in diagrams. We will also describe each grouping of use cases in a use case specification after each figure.

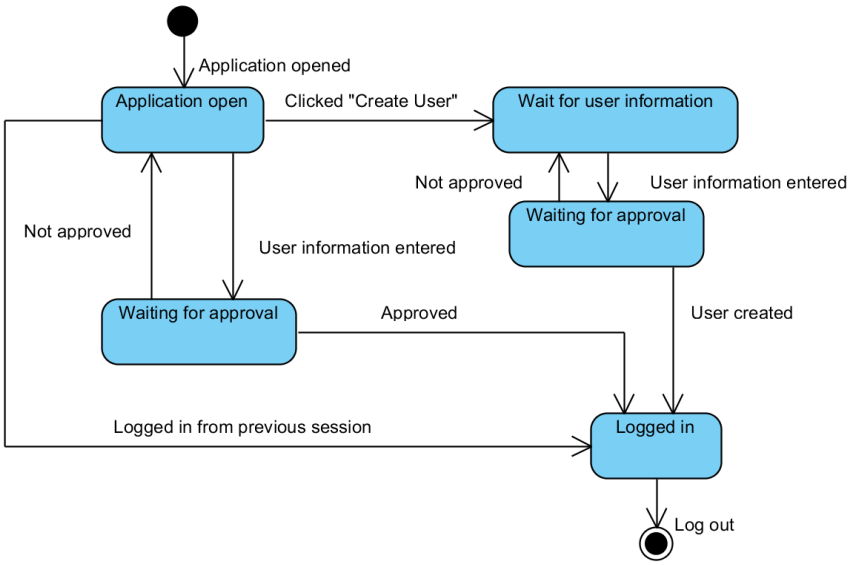


Figure 2.10: Use case for logging in and creating user

Logging in and creating user

Use Case: This use case is initiated when the application is opened by the user. The user can choose whether he wants to login (requires an already made user account), or create a new user account. If he chooses to login, he needs to input user information, which is username and password. If he chooses to create a user, he will have to input user information, which is username, password and email. If he creates a user, he'll be logged in right after.
Objects: Host, Attendee
Functions: Create user, Log in

Figure 2.11: Use case specification for logging in and creating user

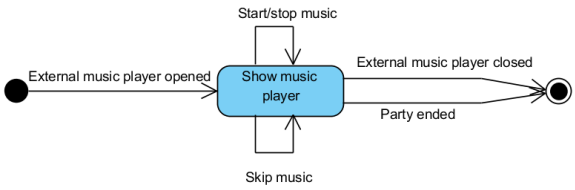


Figure 2.12: Use case for playing music

Playing music

Use Case: This use case is initiated by the user, and it interacts with the external music service. When the external music player is opened, a user can start playing the music by pressing a play button.
Objects: Host, Attendee, Playlist, Song
Functions: Start music, Stop music, Skip music

Figure 2.13: Use case specification for playing music

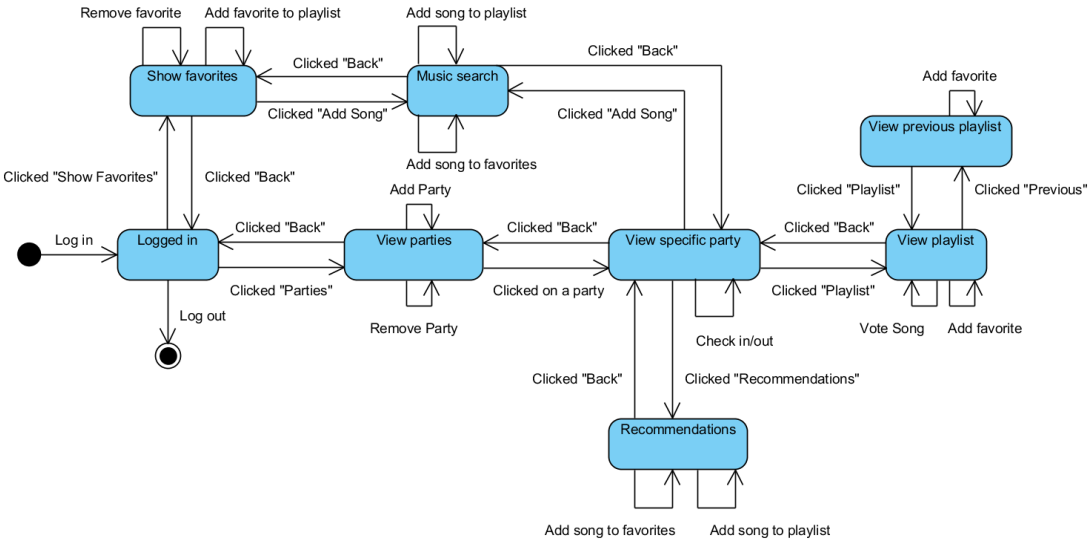


Figure 2.14: Use case for administrating music

Administrating music

Use Case: This use case handles everything that has to do with administrating what songs will be played, including what's added, and is initiated by a user logging in. While he is logged in he can see his favorites, search for music, view his added parties, view a specific party in detail (and check in to this party), get recommendations for the party and add or remove favorite songs as well as add songs to the playlist. He can navigate through the different screens as shown by Figure 2.14.
Objects: Host, Attendee, Party, Playlist, Song
Functions: Log in, Check in, Add party, View parties, View Favorites, View playlist, Add song, Add favorite, Remove favorite, Get recommendations

Figure 2.15: Use case specification for administrating music

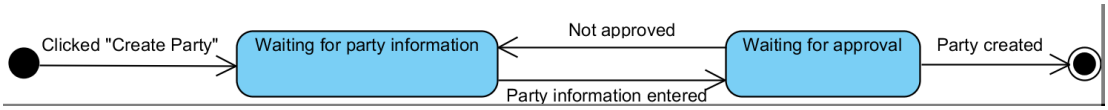


Figure 2.16: Use case for creating parties

Creating parties

Use Case: This use case shows how a party is created by the user. The user can press a button to create a party, and enter the information required, such as location, description and time.
Objects: Host, Party
Function: Create Party

Figure 2.17: Use case specification for creating parties

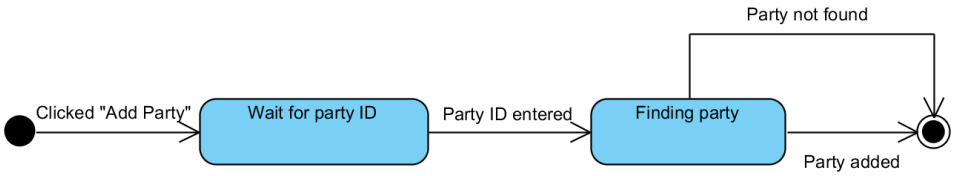


Figure 2.18: Use case for adding parties

Adding parties

Use Case: In this use case, the user can add parties. This is done by clicking a button, which lets the user enter a Party ID, and adds the party to the user's list of parties if it's found.
Objects: Attendee, Party
Function: Add party

Figure 2.19: Use case specification for adding parties

2.4.2 Functions

In Section 2.4.1 we focused on *how* the system would be used. We will now focus on *what* the system is going to do. We do this by identifying the functions we want the system to do, and then analyze what type each function is. In our system we deal with 4 different function types.

Update Update functions are activated by events we found in the problem-domain analysis (Section 2.3), and will result in changing the model's states.

Signal Signal functions are activated by changes in the model's states (from update functions), and will result in a reaction in the context. The signal function is connected to a critical state of the model.

Read Read functions are activated whenever there's an actor who needs information from the system, and the result will be displaying the relevant information to the actor.

Compute Compute functions are activated when the actor needs information that must be computed before it can be shown.

We will find functions from our event table in Section 2.3.1 and our use cases from Section 2.4.1, and then show a list of our system's functions in Table 2.3. We deal with 4 different levels of complexity; simple, medium, complex and very complex. As seen by

Function name	Complexity	Function type
Create user	Simple	Update
Log in	Simple	Update
Create party	Simple	Update
Check in	Simple	Update
Check out	Simple	Update
Add party	Simple	Update
View parties	Simple	Read
View favorites	Simple	Read
View playlist	Simple	Read
Add song	Medium	Update
Vote for song	Simple	Update
Remove vote	Simple	Update
Add favorite	Simple	Update
Remove favorite	Simple	Update
Start Music	Simple	Update
Stop Music	Simple	Update
Skip Music	Simple	Update
Party ending	Medium	Signal
End party	Simple	Update
Update playlist	Medium	Compute
Get recommendations	Complex	Computer

Table 2.3: Function list

Table 2.3, our system has a lot of simple functions. We will not go into details with these, as they are trivial. Instead we will focus on elaborate functions that has a complexity of Medium or higher.

Add song Although it may seem like adding a song should be a simple function, we have to get the song before we can add it. To do this we first need to establish a connection to an external library. When we have found the song on the external library, we have to download the data, and parse the data on to our playlist. The complexity comes from getting the data, and handling it in a good way.

Party ending This isn't a function that is initiated manually, but is triggered when a party hits its end time. When this happens, the party have to signal the host that it has stopped, and signal the external music player to stop playing music. It will also have to stop taking requests, and not allow attendees to add or vote, and check out any attendees who are checked in.

Update playlist When we want to update our playlist, we first have to sort it by the amount of votes each song has. Then we can continue on to showing the updated playlist to the user.

Get recommendations We want to be able to recommend songs to the users at parties. To do this, we need to look at the songs already played at the party, and look for similar playlists from other parties, to find songs from those playlists we could recommend for this party. We will go into more detail with this function in Section 3.4.

2.5 Architectural Design

In this section we describe the architectural design of our system. The architectural design focuses on structuring the system using components and defining system requirements in the form of criteria.

2.5.1 Criteria

This section is about the criteria our system needs to fulfill. The purpose of making these criteria is to set the design priorities for our system. The design priorities are an important part of achieving a good design. The result will be a collection of prioritized criteria as seen by Table 2.4. We define our criteria from the McCall quality factors [15].

Usable:
The purpose of our system is to give the user a larger influence on what music is played at the parties they are attending, in an easy way. This means we need to focus a lot on the usability of our system. The user should be able to use the system without thinking a lot about how the system works, as the user might be under the influence of alcohol.

Secure:
The system does not contain any person sensitive information, other than a password made when making an account. The password needs to be encrypted in case a user uses a password he/she uses for other things, but other than the password, the system does not contain any sensitive information, so security is not a important part of the system.

Efficient:
The system have a lot of communication with an remote database/server which handles things like music searching/sorting. This has to be efficient so the user doesn't have to wait for a long time to see the playlist at a party, or find a song to add to the playlist. Efficiency is an important part of the system.

Criterion	Very important	Important	Less important	Irrelevant	Easily fulfiled
Usable	X				
Secure			X		
Efficient		X			
Correct	X				
Reliable		X			
Maintainable		X			
Testable					X
Flexible	X				
Comprehensive		X			
Reuseable			X		
Portable	X				
Interoperable	X				

Table 2.4: Criteria table

Correct:

In Section 2.2 we set up constraints for our system. Since we have requirements that must be fulfilled in order for our system to work, we see this as a very important part of our system. Our system must comply with the hard constraints we set, and should comply with the soft constraints.

Reliable:

Reliability is of course important for any system. However, since we're not dealing with information that changes very important aspects, if our system does make a miscalculation or a mistake, it's not a big problem.

Maintainable:

Since we're using an external music player, we might have to maintain our system, if the external system changes, it should be easy to maintain our system. We can do this by collecting the parts dealing with the external music player in one place, so it's fast and easy to correct it.

Testable:

We will not focus at all on making the system testable, since we will have to test everything during the development of the system we believe this is easily fulfilled by the development itself. We should easily be able to see whether the system does what we want by saving or logging data, and look in the saved or logged data.

Flexible:

The system's functions should be grouped, so that we can change the interface without changing the functions, and the other way around. We should also be able to change the function of one component without affecting the other components.

Comprehensible:

The system should be kept as simple as possible, so that a user without any experience easily can start using the system during a party, even when under the influence of alcohol, and easily find his way to the wanted functions. To help achieve this, we split the functionality up into an application and a webservice. Because of this, we can keep both things simple, and give them very clear functionality.

Reusable:

It's not very important for the system to be reusable, as our system has a quite specific goal, which wouldn't make much sense to implement in other systems.

Portable:
People attending a party use different kinds of smartphones, so we want our system to be easy to moved to other technical platforms, so that we can make sure everyone get to use our system, and not just those with a certain platform. We can do this by having the main functionality on a platform independent component, such as a web server.

Interoperable:
As we're using an external system, it's very important that we can connect to this, fast and easy (to be interoperable).

2.5.2 Components

In Section 2.5.1 we specified our overall system criteria, in this section we will describe the major components of our system to give a better overview. As seen in Figure 2.20 our system consists of 4 major components; smartphone application, website, web server and external music service.

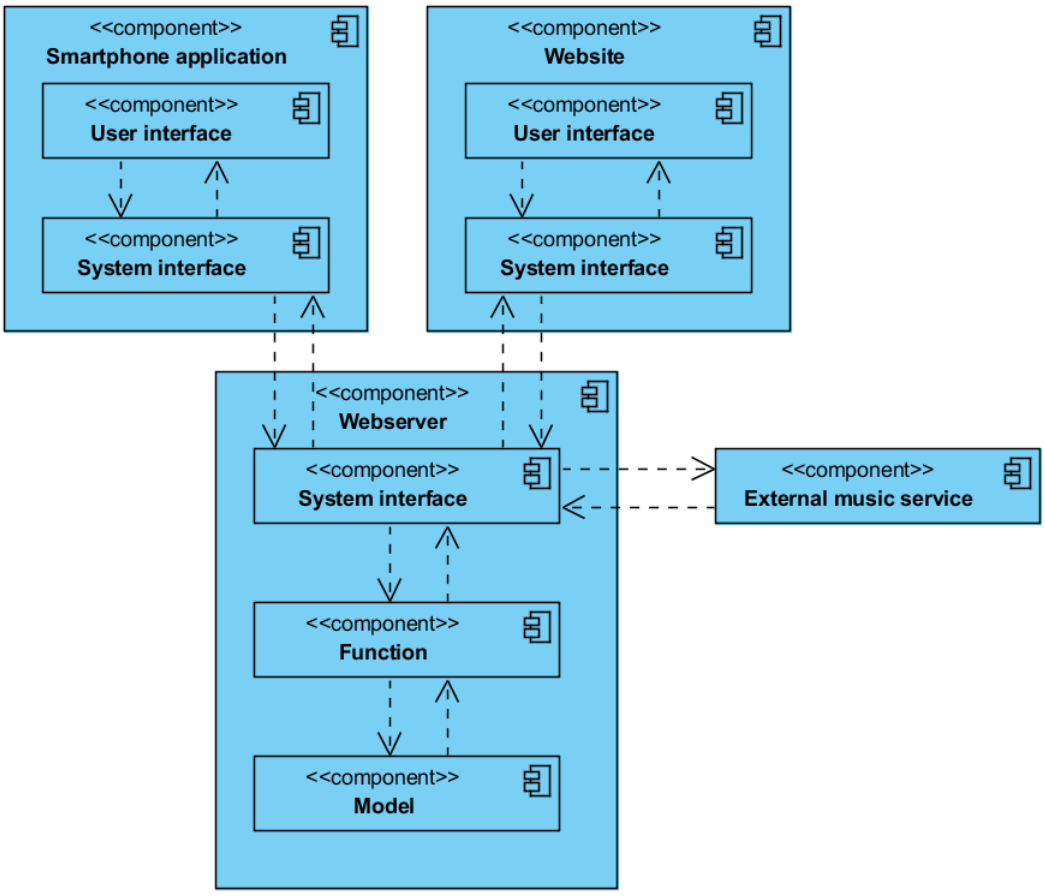


Figure 2.20: Overall system components

We have the smartphone application for the attendees, as the attendees, who are part of the system, have a smartphone with them to a party, and for this reason it is from this component that all functions and operations related to an attendee can be called, while the website is for the hosts, since the hosts have a computer with an internet connection.

The smartphone application and the website does not contain any major functions by itself, and as such only consists of a user interface, described in Section 3.3.1, from where the user may call various functions and operations located on the web server. In our system the web server is where all our major functions and operations are located. All communication between components also happens through the web server, including the connection to the external music service. This structure means that there is a strong dependency between all major components, which can cause problems if the web server is unstable, but it also means that all our functions are easier to keep track of as they are all gathered in one place.

2.6 Component Design

In this section we will describe the final model of the system. We will describe this by showing a description of the system's components and connecting the information we gathered from analyzing the problem- and application-domain.

2.6.1 Model Component

Figure 2.21 shows a revised version of the class diagram from Section 2.3.2. This version shows a modified structure, with attributes added to the classes. The attributes originates from our behavior-analysis from Section 2.3.3. Figure 2.21 also shows two new classes. The Favorite class originates from Table 2.2 in Section 2.4.1, where the User actor can add favorite songs. We wanted to describe this as a new class which aggregates from Attendee, and associates with the Song class, to show that an attendee can have favorite songs connected to him. The other added class, Music, comes from our event table (Table 2.1) from Section 2.3.1. As shown by this table, Music started/stopped affects multiple classes, and happens iteratively. To simplify this, we added the class Music to describe these events, by aggregating the Music class from Song, which connects to the other classes that is affected by the events.

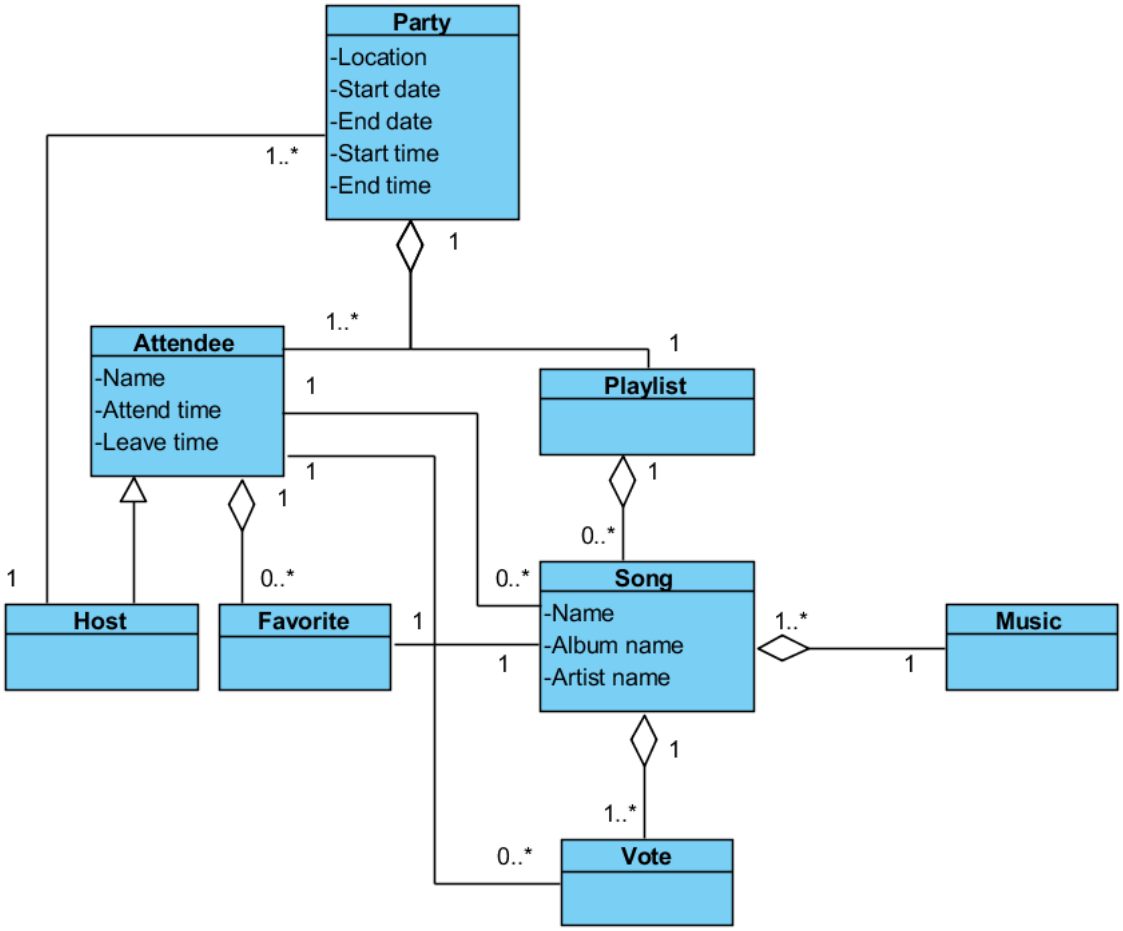


Figure 2.21: Model component

2.6.2 Function Component

After making the model component in Figure 2.21, we will now extend this figure by adding functions from Section 2.4.2 to it, and the result can be seen in Figure 2.22. Since we don't have any complex functions, we won't go into detail with the added functions.

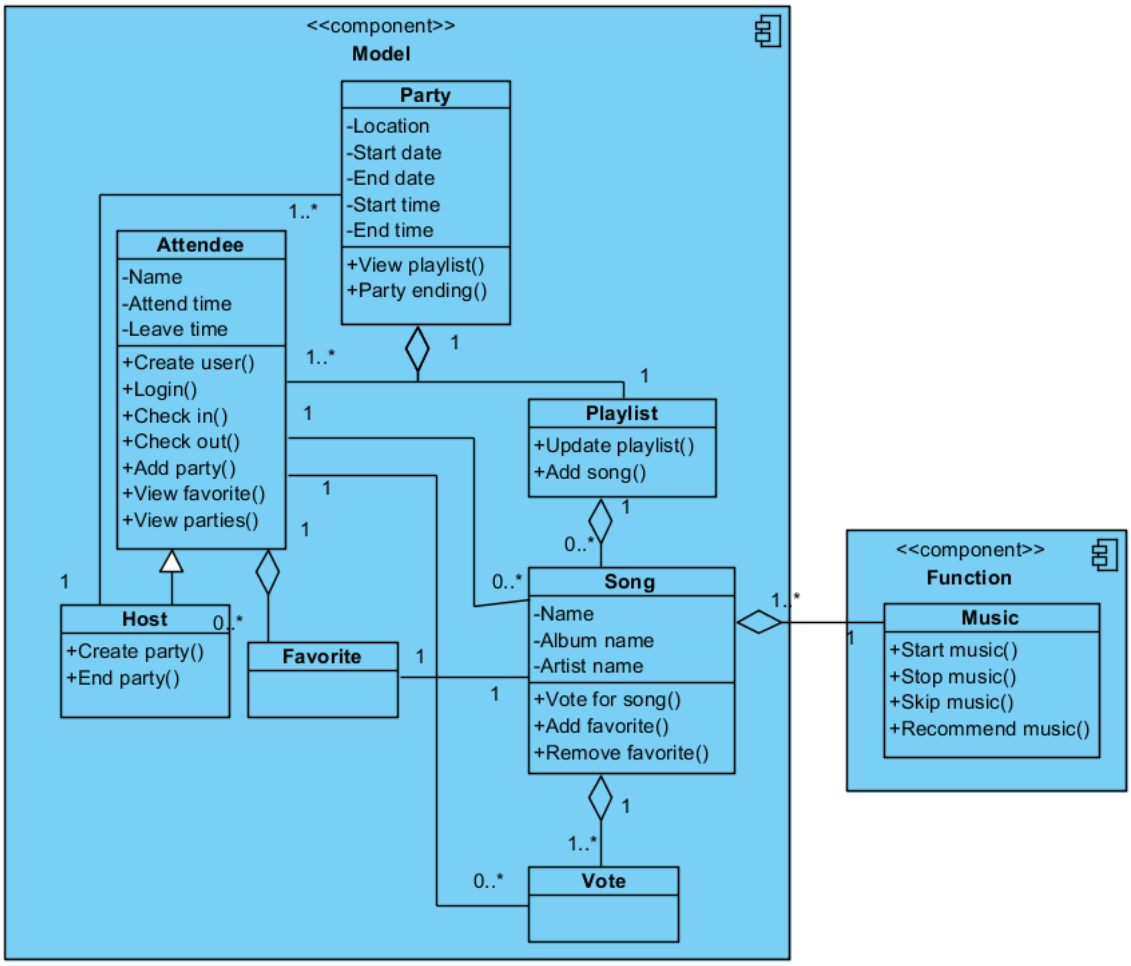


Figure 2.22: Function component

2.6.3 Connecting Components

Figure 2.20 from Section 2.5.2 showed the connection between the different components in our system. The function and model components we showed in Section 2.6.1 and Section 2.6.2 shows that all functions in our system are in those components. This leaves little function to our smartphone application component and website component. Recall from Figure 2.20 that we showed a connection between all components in our system. Because of this, we can have all of our functions in one component, and that the smartphone application and website components can simply just call these functions from buttons in the user interface. Because of this structure we have a "Operation call"-connection between our components, except from the connection between the function and model component which shows a "Class aggregation"-connection. This architecture should give us low coupling and high cohesion, which means that a change in one component should not require a change in other components.

Chapter 3

Design

This chapter is about designing the system from the model we developed in Chapter 2. We'll start by describing the architecture of the system in Section 3.1 more visually. We will show the final navigation map in Section 3.2, and describe the user interface of our final system in Section 3.3. Section 3.4 describes the developed algorithm for our system.

3.1 Architecture

In this section we will show and elaborate on the basic architecture of the system.

The system is split into 5 different components:

- Internet
- Web server
- Smartphone
- External music library
- Computer

Figure 3.1 shows a visual network diagram representation of the system and its components. A component in a square, implies that there can be multiple units of it in the system. The database is a subcomponent of the web server, i.e. the web server hosts the database. The speakers are also a subcomponent of each computer, but are not interesting for this project, but it's important to note its existence in the system. We will now go through every component and talk about its use in the system.

Internet

The Internet is a very abstract component and it is a very strong component, it is, however, not very interesting for our project. This component allows for easy interaction between components without direct connection, and because of this it's a very attractive connection interface.

Web server

The web server hosts a website and the subcomponent; the database. The database is not necessarily a subcomponent as it can be hosted on a standalone server. The website is used by the host to take care of creating parties and setting up the party and it's music player. The database is the storing entity of the system. All the information is stored in the database, such as party information, users and playlists. The database hosts information for each party.

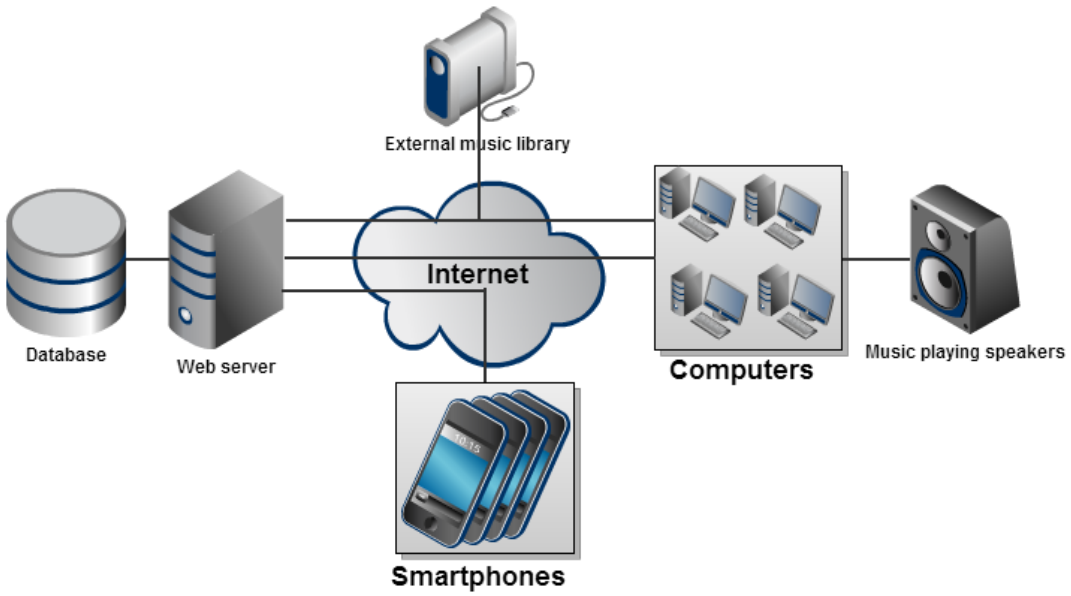


Figure 3.1: System architecture

Smartphone

The smartphone is an important part of the system. Party attendees can manipulate and interact with the party's music directly through the smartphone. The smartphone is mainly used for adding and voting up songs, but also has other important features. This component uses internet connection and has a direct interaction with the web server. The smartphone's internet connection is mainly used to change or receive database data. The adding feature of the smartphone searches the external music library, and adds the song picked by the user to the party's playlist.

External music library

The external music library is quite essential for the system. The library provides us with an almost complete music library and easy to use API solution. Using an external library allows us to focus on what we think are the more important parts of the system, as we do not need to have our own music library and player. As seen by Figure 3.1, the external music library is connected through the internet, allowing easy interaction with the system.

Computer

The computer is either a laptop or desktop computer. The computer's simple task, is to take care of playing the music. The computer should only be used to set up the music player and the system will take care of playing the chosen music. The computer plays the music from the external music library. The system doesn't need any installment, and can be used on any computer without installing new software.

3.2 Navigation Map

In this section we will show and describe the navigation maps for the website and for the smartphone application. We do this to easily show how to navigate in the system.

In the navigation map, as seen in Figure 3.2, each box represents a screen in the application and the first time it is opened it starts at the top screen. An arrow from

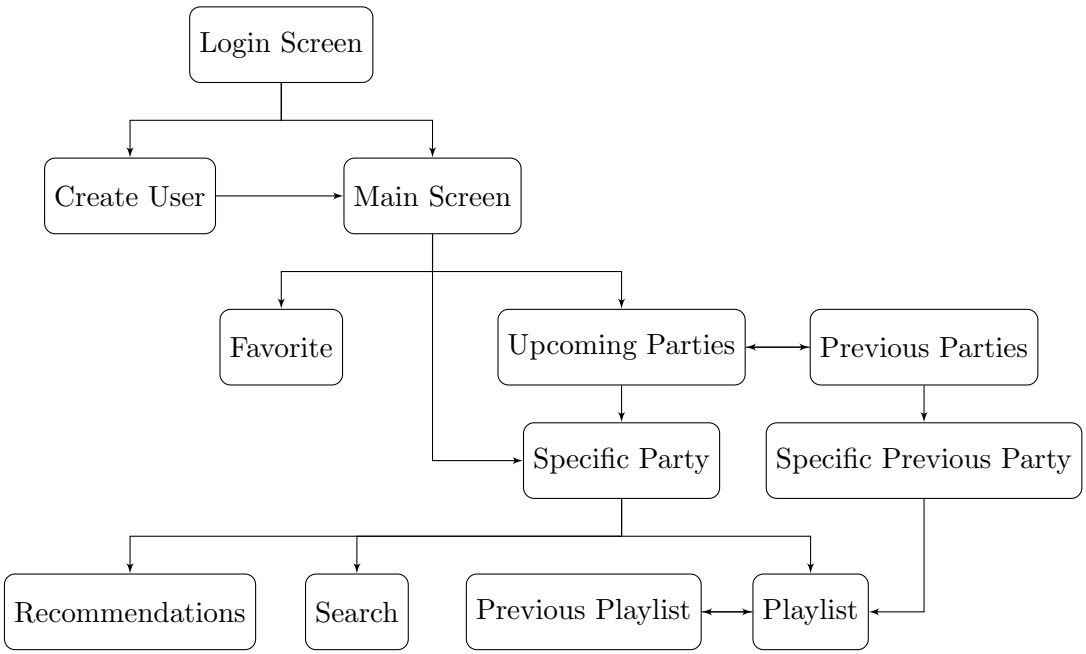


Figure 3.2: Smartphone application navigation map

one screen to another represents a link, which means that the application can navigate from that screen to the other, while an arrow that goes both ways represents tabs in the application. It is always possible to navigate back (up the hierarchy) from every screen after the Main Screen, it is always possible to get back directly to the Main Screen using a "Home" button. The navigation from Create User to Main Screen is automated, as the user is logged in after creating a user.

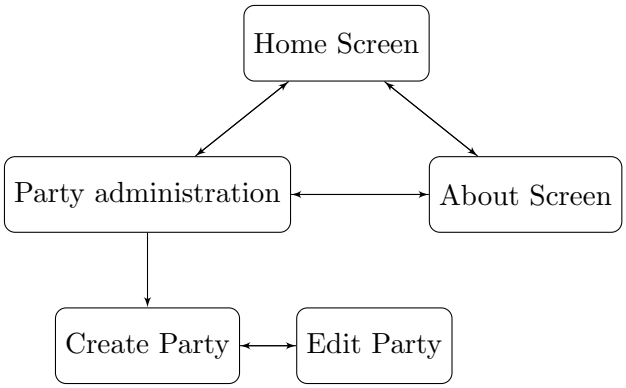


Figure 3.3: Website navigation map

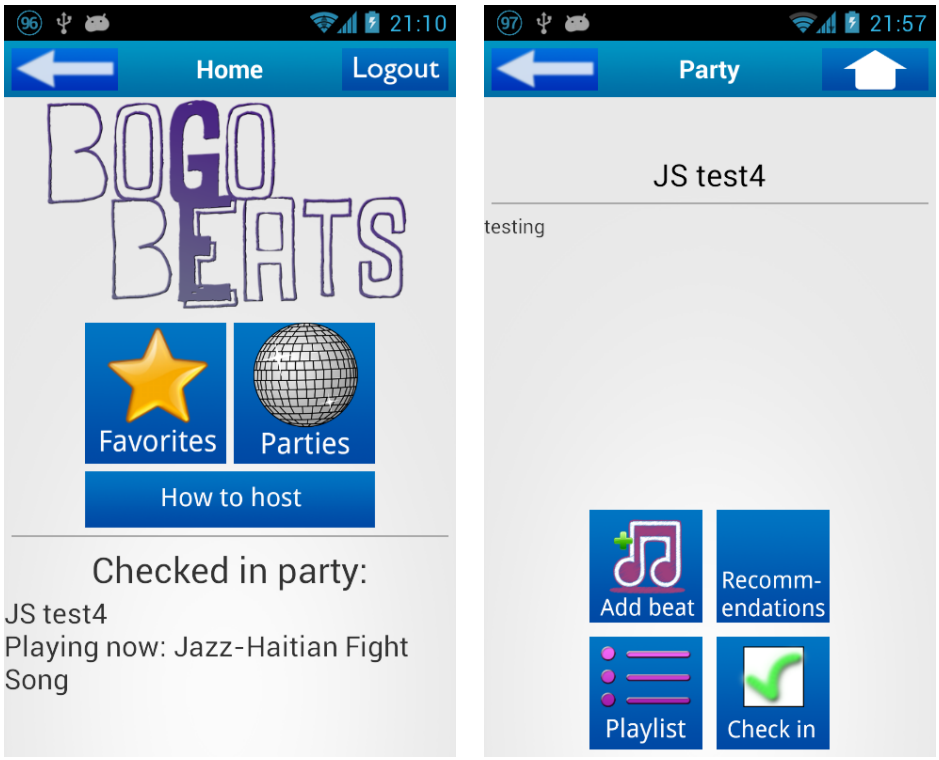
Our navigation map for our website, as seen on Figure 3.3 follows the same principle as the navigation map for the smartphone application, however our website only contains 3 different tabs that can be accessed at any given time on the site.

3.3 User Interface

In this section we will describe the user interface of our system. We have two sets of user interfaces; one for the smartphone application, and one for the website.

3.3.1 Application User Interface

We wanted to make the smartphone application and the website look much alike, using the same color theme and other similarities, such as feels and looks. After a prototype test with the informants (see Section 6.2), we learned that the application should have brighter colors, to make the screen easier to see during parties, where there might be low lighting. Since there's a chance of the users to be under the influence of alcohol, we also had to make sure that we used large buttons and text, so that it would be easy for the users to see and understand the buttons. To give an even better understanding of the buttons and their functions, we also added icons for almost every button, to give an illustrative meaning to the buttons. Each icon represents the function, such as a disco ball represents a party and a musical note represents a song (or a "beat") as shown by Figure 3.4. After the first prototype test, where we learned that we had to redesign our application, we held a design workshop in the group, and went through each screen, designing them from scratch. Pictures from this workshop can be found in Appendix A.



(a) (b)

Figure 3.4: Smartphone application user interface

3.3.2 Website User Interface

For the design of our website we sought an easy way of making a good looking user interface, as we didn't want to use too much time on designing. For this we used an front-end toolkit called Bootstrap made by Twitter, a popular social networking service [2]. Bootstrap is a collection of CSS and HTML degin templates, allowing us to quickly implement a user friendly interface. Bootstrap is open source and licensed under the Apache Licence 2.0 [4].

We wanted the interface of the website to be simple and easy to use, which meant using as few buttons and text as possible, while making the buttons easy to see and

understand. Figure 3.5 shows a scaled screenshot of the website's home page. As seen by Figure 3.5, we added a guide for the users, to make it even easier to understand how to use our system.

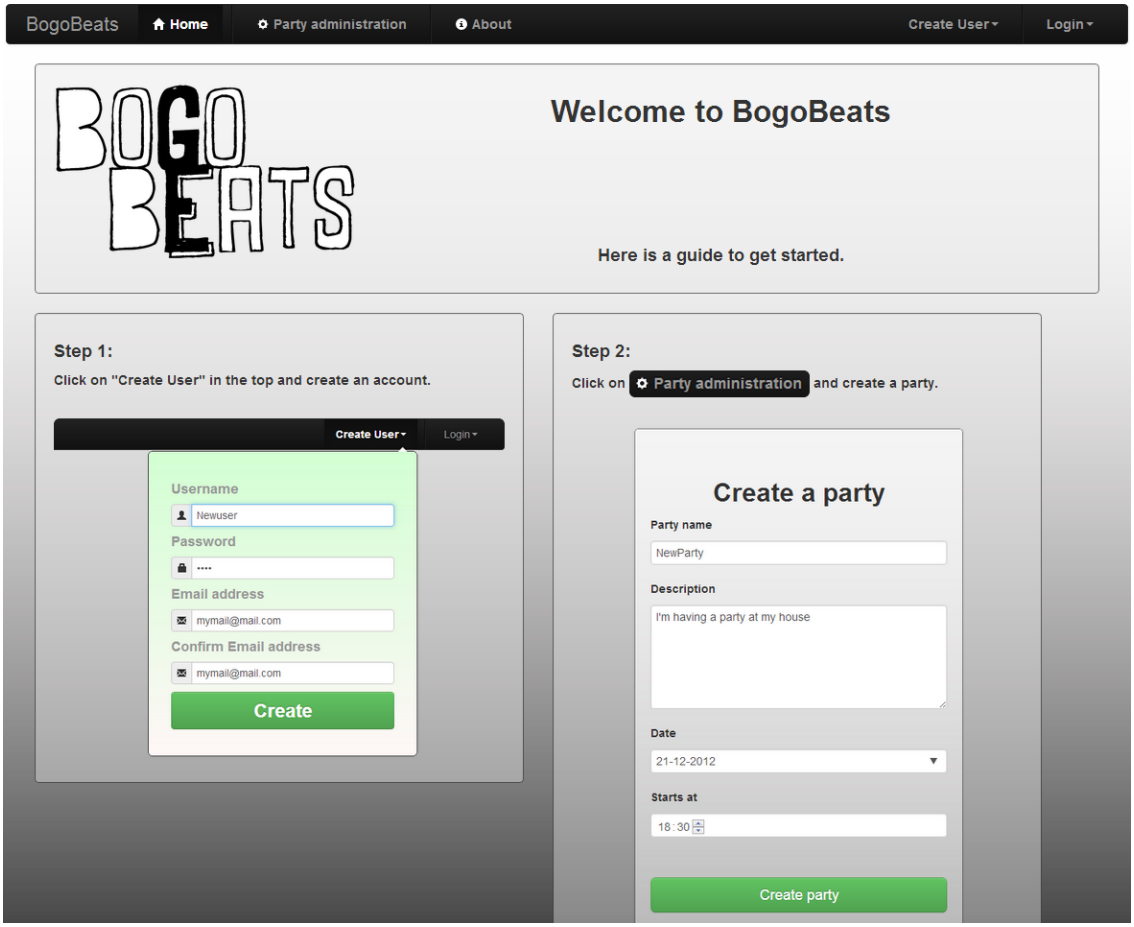


Figure 3.5: Screenshot of the website

3.4 Collaborative Filtering

A way to implement the recommendation component of our system is to use collaborative filtering. Collaborative filtering is the process of collecting user preferences and filtering the interests of a user, giving one the ability to predict what a user might like, based on his history of actions. For instance, consider user A and B buying item I and J. Now, user C buys item I, and with the purchase history of the users, we can predict that he might also like item J. If this is the case, we have successfully helped user C. This is a very practical technique, which we will take advantage of. The approach of suggesting requires one to have a history of party data. This means that in the start, none to few recommendations will be given. We looked at different approaches for giving recommendation. A very practical choice is the Slope One algorithm [14], which we looked into. For simplistic reasons we decided to go with our own algorithm, which will be described in this section.

We looked at different approaches of how we want our users' opinions on songs, and we came up with these two:

1. Make up a rating based on the votes of a song

2. Use a binary representation of the song

We do not want to ask single users for ratings. It does however, make sense to define the rating by the only numerical value the users provide the system; the number of votes given. We do not want to provide single-user-specific ratings, but party-specific ratings, thus all the users are an entity, and are represented by the party. We calculate the rating of a song by the number of votes it has received during the party.

This is a decent way to get good results of a rating system. However this is a rather abstract way of assuming how a party likes a song, since it isn't really a rating, but rather an approximative rating. A binary representation would also work, but seeing as we already have these votes as an indication of popularity, we might as well use the votes for a rating. Defining popularity is a hard thing to do, and we will not go deeper with this choice, as when dealing with peoples preferences, it is not easy to say which way is the best.

It makes sense to say that we rate a song, if it has been played at a party. We know that, if a song has been played, it is most likely played because the given party likes the song. We can assume this because if the song wasn't liked by anyone, it wouldn't have been played. Now with this approach we have an idea of what kind of music the party likes. We can then start giving recommendations to the party based on previous playlists, which are stored in our database. Table 3.1 shows an example of this approach.

	Song 1	Song 2	Song 3
Party A	Played	Not played	Played
Party B	Played	Played	Played

Table 3.1: Example for collaborative filtering

In this example these parties' played songs are quite similar and we want to suggest Party A to play song 2. Based on this example, we need a way to define the similarity of two parties' played music. This can be done with cosine similarity [6]:

$$Similarity(\vec{a},\vec{b}) = cos(\vec{a},\vec{b}) = \frac{\vec{a} \bullet \vec{b}}{||\vec{a}|| \cdot ||\vec{b}||} = \frac{\sum_{i=1}^{n} a_i \cdot b_i}{\sqrt{\sum_{i=1}^{n}(a_i)^2} \cdot \sqrt{\sum_{i=1}^{n}(b_i)^2}} \tag{3.1}$$

Where our vectors are the 0's and 1's for a given party. Party A and Party B's vectors would be:

$$\vec{PartyA} = \begin{bmatrix}1\\0\\1\end{bmatrix} \vec{PartyB} = \begin{bmatrix}1\\1\\1\end{bmatrix}$$

Recommendation Example

A quick example of a run through of our algorithm with 4 parties and 6 songs. The ratings are votes. If a song has 0, it was not played at the party.

Now for every party we calculate their similarity with party D. Realistically speaking, similarity calculations will only happen for "live" parties.

$$Sim(A,D) = 0.624756$$
$$Sim(B,D) = 0.429274$$
$$Sim(C,D) = 0.455083$$

	S1	S2	S3	S4	S5	S6
Party A	10	5	7	0	3	0
Party B	9	4	6	4	0	0
Party C	0	0	2	4	7	10
Party D	3	1	0	0	5	0

Table 3.2: Recommendation example

Now, we want recommendations for Party D, we look for the highest similarity and output the songs that are not in D. In this example the highest similarity is $Sim(C, D)$ which would recommend S3, however it would have recommended more songs if party A had more songs that had not been played by party D.

3.4.1 The Collaborative Filtering Algorithm

Now we can define our algorithm for computing and outputting the recommendations for an ongoing party as seen in Algorithm 1. Notice that the algorithm takes as input 2 arrays. The input parameters are arrays of objects containing a song ID and a rating for the given song, both values are integers. The first input is the playlist of the party which requires recommendations, the second input is an array of older parties' playlists, each k songs long, as we compare an ongoing party with older stored parties.

The operation of Line 18 implies the operation of making vectors of the playlist arrays. For each vector we add the songIDs of the other vector and put 0 as rating, same as we did in the example.

Algorithm 1 Recommendation Algorithm

```
1:  procedure CollaborativeFiltering(Playlist[0...m], StoredPlaylists[0...n][0...k])
2:      ArraySimilarities[0...n]
3:      for i = 0 to StoredPlaylists.length() do
4:          Similarities[i] ← CosSim(Playlist, StoredPlaylists[i])
5:      end for
6:      Greatest ← 0
7:      SimList ← null
8:      for i = 0 to Similarities.length() do
9:          if Similarities[i] > Greatest then
10:             Greatest ← Similarities[i]
11:             SimList ← StoredPlaylists[i]
12:         end if
13:     end for
14:     Return Songs from SimList ∉ Playlist
15: end procedure

16: procedure CosSim(Play1[0...n], Play2[0...n])
17:     Sort Play1 and Play2
18:     Vectorize Play1 and Play2
19:     return Cosine Similarity of play1⃗ and play2⃗
20: end procedure
```

Time Complexity Analysis

In the following table the complexity of the algorithm is shown. n is the number of stored playlists which we want to compute similarity with and m is the number of songs in the given playlist. In Table 3.3, $z = \max(m, k)$. Line 18 takes $O(z \log z)$ because it's simply comparing two lists (playlists).

Collaborative Filtering	
Line 3-5	$O(n) \cdot Cossim$
Line 8-13	$O(n)$
Line 14	$O(n \log n)$
Cosine Silarity	
Line 17	$O(z \log z)$
Line 18	$O(z \log z)$
Line 19	$O(z)$
Total complexity	
$O(n \cdot z \log z)$	

Table 3.3: Time complexity of Collaborative Filtering algorithm

The overall asymptotic time complexity is a polynomial function. Using all playlists stored is unrealistic and would also not be that relevant, seeing as new music is often released and with this a changing music listening style. So using the last 100 playlists should give us a good result. The number of songs for a party can in worst case be 15 million (Groovesharks library size [12]), however this is not a realistic number, as no party will ever play that many songs, but we could limit it to top 100 songs from each playlist to keep costs down. It should be noted that this algorithm should be running whenever a user prompts for suggestions.

Making a recommendation system isn't an easy task because defining the best way of recommending songs is subjective. There are many ways of doing this, and there's no way of defining which way is best. We do it by finding similar parties, and recommend songs from one party to another. This is based on the assumption that we can use votes as a form of rating, which is not necessarily true. The way we do it, will also make it hard to give good recommendations for parties that haven't played many songs yet.We could also have chosen to recommend songs to a specific user, instead of recommending songs to a party. We didn't do this because recommending music to a user wouldn't necessarily help the music at a party. There's also the possibility to recommend songs based on an actual rating, to which we could have let users rate the songs. This would have required more effort from the users, and it would take a lot of votes before it would be any good.

The way we recommend songs is by having a list of songs that is recommended for a party that the users can access by pressing the "Recommendations" button. This means that before it's any good, users would have to look through this list, and add songs they like. We talked about doing so that the songs we recommend gain more votes each time they are being voted on, so they're more likely to be played at a party. We felt that this would be against what we tried to achieve (a democratic playlist). Imagine if the same applied to a political election, giving some people more votes because a system recommends them. This would be absurd, which is why we didn't do this. We could also simply have added the recommended songs to the playlist with a certain amount of predetermined votes, without the need of users to add them. This would, however, feel like we were forcing these songs onto the users at the parties, which we didn't want to. In the end, we came to the conclusion that recommending songs and letting the users choose if they wanted to hear them, would give us the best result.

Chapter 4

Platforms

In this chapter we will describe our choice of platform to implement our application on, and the choice of database. We will look at the major platforms, and then choose one of those platforms. We will describe the choices we made with the web server in Section 4.1, and the choices made for the smartphone application in Section 4.2.

4.1 Web Server

For the server part, there were few hard choices for us. Basically there was a group member with previous PHP and MySQL knowledge. Seeing as we already had quite a few challenges with learning new programming environments, we chose to go with MySQL and PHP. Apart from this, MySQL is a really popular open-source database software solution, and provides a very easy to learn back-end solution. Along with this, PHP is also a very popular programming language for back-end server side scripting.

4.2 Smartphone Application

In this section we will choose the platform we will implement the smartphone application on. We will discuss the choice between a web and a native application, as well as which native application platform that would best for us and our system.

4.2.1 Choice of Platform

Web Application vs Native Smartphone Application

Before we analyze which smartphone platform would be the best to implement our application on, we look at another option: Web Application, or Web App for short.

Because all smartphones can access the internet rather easy with web browsers, making a web app is very attractive when implementing a system that we want to be platform independent. By making a web app, we could reach all smartphones, and the users could even use tablets, laptops and desktop computers. The web app and the native app each have their own pros and cons. Native apps are usually faster, but are more expensive to develop than web apps [13], but since this is a learning process, we won't focus on these, but rather on what we think is best for us.

We chose to work on a native application, as we get to work on developing a real application for a smartphone platform, that will mimic the production of a professional product more than developing a web application.

Smartphone Market Share

We start by looking at the market share and the amount of users each platform has. Since we can't know the actual amount of users due to people being able to have more phones, inactive phones, etc., we will show the amount of devices running the respected Operating System (OS). We will, however, treat these as actual users. Table 4.1 shows that Android dominates the smartphone market, followed by iOS. Even though BlackBerry has been on the market for longer [5], and Windows dominates the desktop market [7], they fall behind here.

	Android	iOS	BlackBerry	Windows	Other	Total
Market share	46.9%	28.7%	16.6%	5.2%	2.6%	100%
Devices	472.75	289.30	167.33	52.42	26.20	1008

Table 4.1: Total smartphone devices (in millions) Q4 2011 [10]

We will now look at the age of the users of each platform, to help determine which would be the best target for our service. Table 4.2 shows the age groups of each platform. As seen by the table, the majority of each platform is young people (18-34). Android have the youngest users, while Windows have the oldest.

	18-34	35-44	45-54	54+
Android	50%	21%	16%	13%
iOS	43%	24%	16%	17%
BlackBerry	38%	25%	20%	18%
Windows	39%	20%	21%	20%

Table 4.2: Smartphone users grouped by age [10]

After we have seen these two tables, we see that Android and iOS would be the best platforms to develop our service to. Android have the most, and the youngest, users. Having 46.9% of the market share, and having 50% of their users in our target group, will give us more than 236 million potential users. But we can't ignore the other platforms, especially not iOS, as they have a lot of potential users for us, with it's 28.7% of the market, where 43% of those are in our target group, meaning almost another 125 million more potential users.

Based on this, we chose to work with developing an **Android** application. It's the most popular platform, and has more users in our target group (percentage wise) than the other platforms.

4.2.2 The Android Platform

As seen by Section 4.2, we chose the Android platform. Android applications are written with the programming language Java, using the Android Software Development Kit (SDK). If you already know how Android applications are made, you can skip this section.

When navigating in an Android application, you switch between different screens each with their own functionality. These screens are known as "Activities" in Android. Each activity has a set of pre-made methods, called callback methods. The methods are automatically called when you create, start, resume, pause, stop, restart or destroy the activity. These methods control the lifecycle of each activity. Figure 4.1 shows an entire lifecycle of an activity.

When you move from one activity to another activity, the current activity stops (calls onStop()), and starts or creates a new activity. We can navigate between activities by

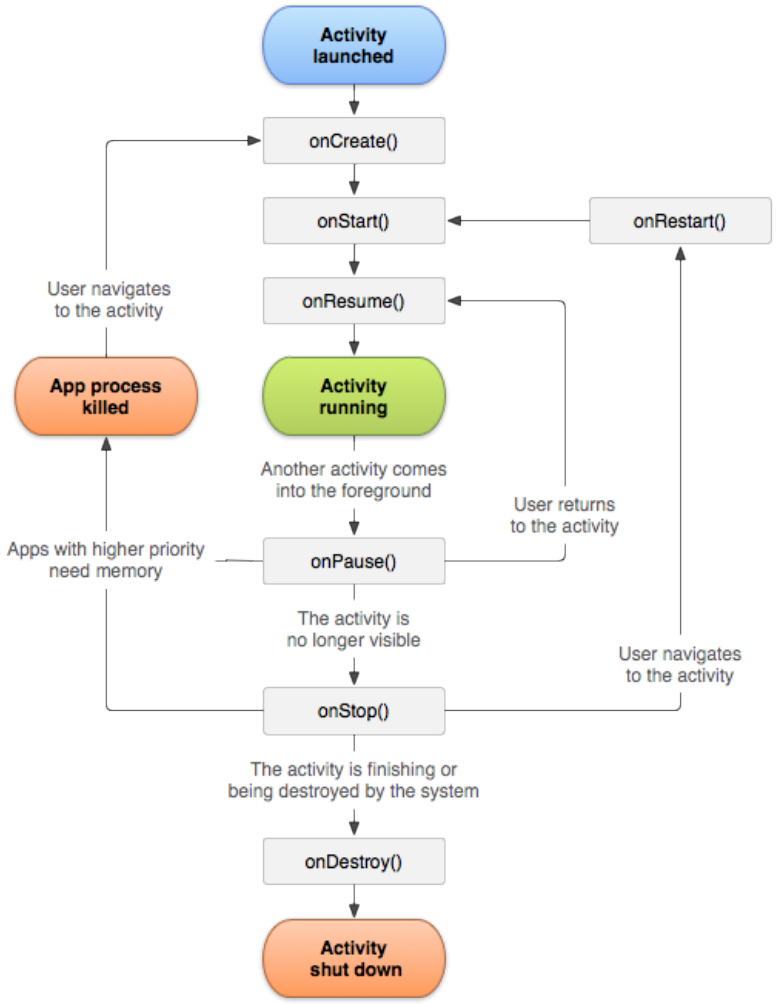


Figure 4.1: Activity lifecycle [1]

calling startActivity() or startActivityForResult(). The first one simply starts a new activity, and the second one starts an activity which should be used to return a result to the activity calling the method. If an activity is started for a result, it will destroy itself when it returns the result. We can then navigate our application by starting activities, but we can also go back from one activity, to the activity that started it, much like a website. We do this by using the "Back" button on Android, or by calling finish() manually. Figure 4.2 illustrates how we can start and return from activities.

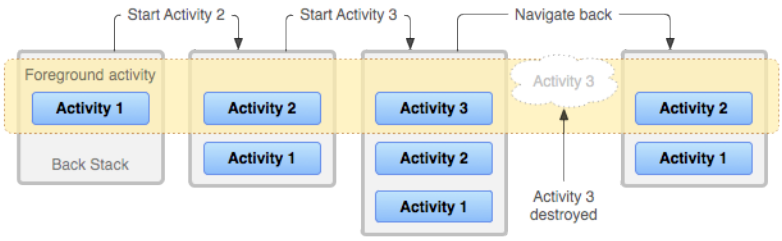


Figure 4.2: Activity navigation [1]

We can send information around in the different activities by using intents. Intents are objects which can hold data. When we start activities, we can start them by sending

an intent with data attached to it, much like how we can call methods with parameters. From the started activity, when can then collect the data.

Every activity should have a graphical user interface (GUI). Making a GUI can be really easy if you use an integrated development environment (IDE) such as Eclipse, otherwise you would have to define it manually by writing XML-code. Eclipse makes it really easy to make a GUI, as you can simply drag and drop elements, and Eclipse will make the corresponding XML-code. We can control the GUI of each activity by calling setContentView("view"), where "view" is the layout we want to set. Setting the content view is usually something you want to do in the onCreate() method, so it's the first thing that will be done when we start the activity, and is only done once.

To sum up: In Android we have different screens called activities, and we can navigate through these, and choose the GUI of each activity.

Chapter 5

Implementation of the System

In this chapter we will describe the implementation of the system. This chapter consists of 4 main sections; Section 5.1 which describes the implementation of the database component as well as the collaborative filtering algorithm, Section 5.2 which describes the external music service and how we use it, Section 5.3 which describes the implementation of our application component on the Android platform, and Section 5.4 which describes the implementation of the website.

5.1 Database and Algorithm Design

In this section we will describe the server part of our system. Our system uses a web server to connect a number of people to a party through their smartphones. The web server basically consist of two parts: A database and an API for reading and writing data in the database, for the app and website, made in PHP. The database is used to store all the data needed to make the system work. The API is used to connect to the database and to calculate results needed by the application. We use a web server to make it easy for the user to be a part of the system.

5.1.1 The Database

The database we are using is a MySQL database. The description of the database is split in two subsections: Logical design and physical design. The logical design is a ER-diagram based on the model component (Figure 2.21) from Section 2.6.1. The purpose of this diagram is to give an overview of the entity-relationships in the database. The physical design will serve as a list of important tables, and a detailed description.

The Logical Design

The relationship between the entities are made using additional entities consisting of the IDs the different entities have. E.g. The p3_users entity is connected to the songs entity in an n:m (many to many) relationship meaning that a user can favorite multiple songs, and a song can be favorited by more than one user. This relationship is handled by using a table named favorit_songs, which consists of a userID and a songID. The uses of a new table to represent the relationship cuts storage cost, since we can reuse a song, meaning we only have to store the name and artist of the song once instead of storing all the information each time a users adds a song to his/hers favorite list. It could significantly reduce the data redundancy thus enhancing the storage efficiency.

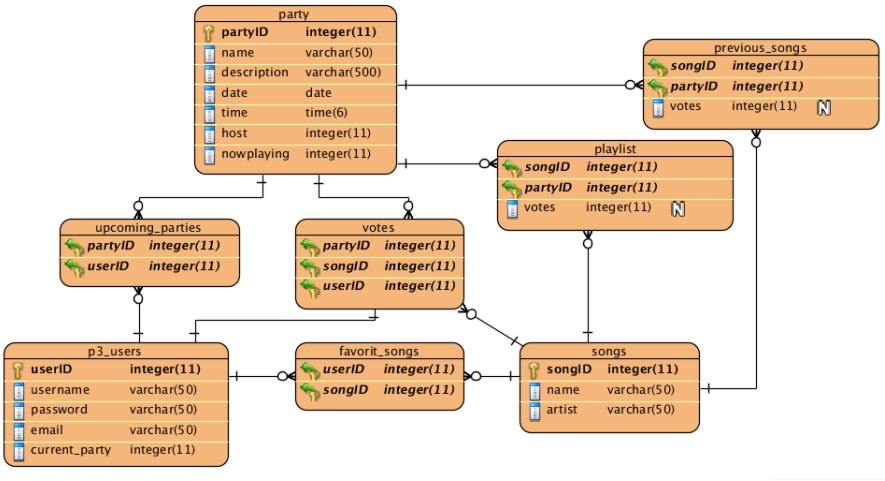


Figure 5.1: Entity-relationship diagram for the database

The Physical Design

The 5 most important tables in our database are: p3_users, party, songs, playlist, votes. The p3_users table, Table 5.1, is used to store information about the users of our system.

Table: p3_users		
Column	**Datatype**	**Notes**
userID	int(11)	primary key, auto_increment
username	varchar(50)	unique
password	varchar(50)	md5 hashed
email	varchar(50)	unique
current_party	int(11)	foreign key(party)

Table 5.1: Physical design user

The userID is the users primary key, and each user have a unique primary key. The userID is stored as an integer, which can be used to identify the user when relating the user to other tables such a parties the user has been invited to or songs the user wants to favorite. The username, password and email are strings that stores information used to authenticate the users, the usernames and passwords are used to make sure which user we are dealing with. The username and password are unique meaning that there will only be one username & password combination which allows the user access to his/hers data. An example of the use of the username, password combination is in our check_login($username, $password) function from the API. The function uses the user's username and password to look for the one and only user with this username/password combination. If it's found, the userID will be returned. Since the send the userID to the application, the application can simply send this ID back when connection to the server, instead of sending the username and password every time. The current_party is a foreign key linking the user directly to a party. The party the current_party links to is the party the user is checked in to.

The party table, Table 5.2, is used to store information about the parties, where partyID is the primary key of the table and is used to identify the different parties. The name, description and date is information to show to the users attending the party, none of these are unique. This means there can be two parties at the same time with the same name and description. The use of unique partyIDs which are used to connect to a party means there is no problem with having two parties with the same name the same day

Table: party		
Column	**Datatype**	**Notes**
partyID	int(11)	primary key, auto_increment
name	varchar(50)	
description	varchar(500)	
date	date	
time	time	
host	int(11)	foreign key(p3_user)
nowplaying	int(11)	foreign key(songs)

Table 5.2: Physical design party

unless a user is assigned to both parties. Host is an integer referencing the userID of the user who created the party. nowplaying is an integer referencing the songID of the latest song pick from the playlist.

Table: songs		
Column	**Datatype**	**Notes**
songID	int(11)	primary key
artist	varchar(50)	
name	varchar(50)	

Table 5.3: Physical design songs

The songs table, Table 5.3, is a little unusual, since the way we define a song in our system is a little unusual. because we are using an external music service. The use of Grooveshark means that we don't need to store the raw data of a song. We store the data needed to make it easy for the user to browse songs on the playlist of a party or scroll through favorite songs. We decided to store the name of the song and the artist, both stored as strings, so that we can easily output these. Besides the name and artist we also store a songID. This songID is the one used by Grooveshark. Using the same songID as Grooveshark uses, means that we can use the songID to play music directly on Grooveshark, using their JavaScript (JS) API. The JS API can be used through a bookmarklet, just like the way djtxt is playing music from Grooveshark.

The playlist table, Table 5.4, consists of references to a party and to a song. These are

Table: playlist		
Column	**Datatype**	**Notes**
songID	int(11)	foreign key(songs)
partyID	int(11)	foreign key(party)
votes	int(11)	

Table 5.4: Physical design playlist

both integers and are the IDs of a party and a song. The playlist also consist of an integer called votes. The value of votes is increased by one whenever a vote is placed in the votes table, when matching the songID and partyID.

The votes table, Table 5.5, consist of only three IDs: userID, partyID and songID. These three IDs are all used as a primary key. Using all three IDs as a primary key means that the table cannot contain duplicates. In other words, it is only possible to vote once for each song in a party's playlist. After a song has been played, the votes for that song are deleted.

Table: votes		
Column	**Datatype**	**Notes**
songID	int(11)	foreign key(songs)
partyID	int(11)	foreign key(party)
userID	int(11)	foreign key(p3_users)
primary key(songID, partyID, userID		

Table 5.5: Physical design votes

5.1.2 Code Examples

The API of the Server

The API consist of all the functions needed to send or receive data from the application or the website. The public functions in the API can be found at the end of this section.

The API is made using a PHP class, which contains all the functions need by the Android application and the website. All the public API functions takes an array as input. The reason all functions takes an array as input, is found in our API connector. The API connector is a PHP script that takes a userID, a function, and some input as function input, as seen in Listing 5.1.

```
5  //Gets the information send to the server.
6  $userID = mysql_real_escape_string($_GET['userID']);
7  $function = mysql_real_escape_string($_GET['function']);
8  $input = mysql_real_escape_string($_GET['input']);
9
10 //Converts the input to an array.
11 $funcinput = explode('|', $input);
12
13 //Inports the API
14 require("bogoAPI.php");
15
16 //Makes a new API object and calls the function with the needed input.
17 $bogoAPI = new  bogoAPI($userID);
18 $out = $bogoAPI->$function($funcinput);
19
20 //Returns the output of the API call to the caller
21 echo $out;
```

Listing 5.1: API connector

When an API function is called, the inputted array gets split into the parameters the function are supposed to work on.

```
32 public function logincheck(array $array)
33 {
34     $username = $array[0];
35     $password = md5($array[1]);
36         .
37         .
38         .
39     return $output;
40 }
```

Listing 5.2: Unfolding an array input

To authenticate our users we need to split the array to $username and $password, seen in Listing 5.2. Logging in with a user using a username; andreas and password; 1234, will look like this:

http://bogosongs.com/Apitest.php?function=logincheck&input=andreas|1234

If there is a user with username: andreas and password: 1234, the userID is returned.

List of public functions with input and output, the functions input is the unfolded array:
Function: logincheck
Input: username, password
Output: If the user is authenticated, the userID will be returned. If the user is not authenticated nothing is returned.
Purpose: The purpose of this function is to authenticate a user, and send the specific userID to the application or website, so the user can log in.

Function: makeuser
Input: username, password and email
Output: If the user is created, "success" is returned. If someone already made this user, "error" is returned
Purpose: The purpose of this function is to store user information in our database.

Function: addparty
Input: userID, partyID
Output: If the user entered a valid partyID, information about the party is returned. If the partyID isn't valid, "party not found" will be returned.
Purpose: The purpose of this function is to assign users to a parties, which allows them to be a part of adding beats to the playlist.

Function: outputparties
Input: userID
Output: If the user is assigned to one or more parties, all the information about the parties is returned, else "error" is returned
Purpose: The purpose of this function is to allow the user to see information about the parties the user is assigned to.

Function: checkin
Input: userID, partyID
Output: If the user can check in at the party with the entered partyID, "success" is returned. If not, "error" is returned.
Purpose: The purpose of this function is to place the user at an ongoing party, allowing the user to add songs directly from the user's list of favorites.

Function: addsongs
Input: userID, partyID, songID, artist, name
Output: If the song is added to the party's playlist, nothing is returned, else "you can only votes once" is returned.
Purpose: The purpose of this function is to allow users to add music to a party's playlist.

Function: checkedinparty
Input: userID
Output: If the user is checked in to a party, information about this party is returned, else "error" is returned
Purpose: The purpose of this function is to make a shortcut to the user's checked in party on the home screen of the Android application.

Function: addtofavorite
Input: userID, songID, artist, name
Output: If the user can favorite the song, "Added" is returned. If not, "error" is returned.
Purpose: The purpose of this function is to allow the user to save music the user likes.

Function: deletefavorite
Input: userID, songID
Output: If the user can unfavorite the song, "Removed" is returned. If not, "error" is returned.
Purpose: The purpose of this function is to allow the user to clean up the favorite list if the users stops liking a song.

Function: getplaylist/getprevious
Input: partyID
Output: If the correct partyID is entered and there are songs on the list, the songs are returned in JSON format, else "error" is returned.
Purpose: The purpose of this function is to show the playlist to the user.

Function: getfavoritesongs
Input: userID
Output: If the user have any songs favorited, the songs are returned, else "error" is returned.
Purpose: The purpose of this function is to show a user's favorite songs to the user.

Function: voteup
Input: userID, partyID, songID
Output: If the user can vote, nothing is returned, else "Already voted" is returned.
Purpose: The purpose of this function is to allow the users to vote for music on the playlist, however, they can only vote once for each song.

Function: cf
Input: partyID
Output: Ten songs recommended for the party.
Purpose: The purpose of this function is expand the users music taste by recommending, songs to add to the playlist.

Collaborative Filtering Algorithm

As mentioned in our design chapter we are implementing a rating based party to party recommendation system to allow our users to expand their music taste. The implementation of our collaborative filtering algorithm is placed on our server, and is coded using PHP and MySQL. In the following example, we call a party we want to get recommendations for; "our party" and the parties we use, to recommend with are "other parties".

The first thing we need to do is to receive the songs played at our party. This is done by running the query in Listing 5.3.

```
10 //Returns all the info about the party entered.
11 $query = "SELECT * FROM previous_songs WHERE partyID = '$partyID' ORDER
       BY songID";
12 $myplaylist = mysql_query($query);
```

Listing 5.3: Sorted playlist

The songIDs of the songs played at our party is fetched to an array for easy access, we do this because, we want to compare these songs, to the songs from the 100 last parties' playlists. We chose 100 as a representative number of party playlists, this can be scaled as wanted.

```
14  //Loads all the songs played at our party to an array.
15  $i = 0;
16  while($mysongs[$i] =  mysql_fetch_array($myplaylist))
17  {
18    $i = $i + 1;
19  }
```

Listing 5.4: Load playlist

In order to get information about the 100 latest incurred parties' playlists without too many database calls, we get the information about the 100 latest parties and load it into an array structure.

```
21  //Returns all the info about the 100 lates parties
22  $startID = $partyID - 100;
23  $query = "SELECT * FROM previous_songs WHERE partyID > '$startID-100' AND
        partyID != '$partyID' ORDER BY partyID DESC, songID";
24  $otherplaylists = mysql_query($query);
25
26  //Loads all the previous_played information to an array.
27  $j = 0;
28  $item = mysql_fetch_array($otherplaylists);
29  $key = $item['partyID'];
30  $othersongs[$key][$j] = $item;
31
32  while($item = mysql_fetch_array($otherplaylists))
33  {
34    //By using the partyID as key we can easily run a foreash loop to return
          song information.
35    if($item['partyID'] == $key)
36    {
37        $j = $j + 1;
38        $othersongs[$key][$j] = $item;
39
40    }
41    else
42    {
43        $key = $item['partyID'];
44        $j = 0;
45        $othersongs[$key][$j] = $item;
46    }
47  }
```

Listing 5.5: Get all partyID's different from the one entered

After loading all the other partyIDs we loop through them and send their playlists and our current playlist to a function which calculates the cosine similarity between the two vectors (the two parties).

```
53 //Calculates the similarity between the inputed party and all other parties.
54  foreach ($othersongs as $key => $songs) {
55    //using the partyID as key to Cosinesim, make it easy to sort the
          partyID's after similarity.
56    $Cosinesim[$key] = calsim($mysongs, $songs);
57  }
```

Listing 5.6: Calculate the similarity between our party and all other parties label

The way we calculate the cosine similarity is a little unusual. We use the formula from equation 3.1 however we only increase the value of B in case the song is in both our vectors. This is done because we want to see the songs played by party B as possible suggestions and not songs that the party A dislikes. The value for A is calculated in the usual way, since we want songs played by our party, but not the other party to decrease the similarity.

```
59 //Getting the similarity beteen our party and another party
60    function calsim($myparty, $otherparty)
61    {
62      //Gets the ammound  of songs played a each party.
63      $mycount = count($myparty);
64      $othercount = count($otherparty);
65
66      //The three values used to calculate the cosine similarity.
67      $topcount = 0;
68      $Acount = 0;
69      $Bcount = 0;
70
71      //Used to keep track of how many songs we have calculated the
            similairty for.
72      $mysongnr = 0;
73      $othersongnr = 0;
74
75      //If our party runs out of songs the similairty will not change further
            .
76      while ($mysongnr + $othersongnr < $mycount + $othercount )
77      {
78          //loads a songobj from the arrays
79          $mysong = $myparty[$mysongnr];
80          $othersong = $otherparty[$othersongnr];
81
82          //If one of our playlist is alot longer then the other, we still
                needs to add it to the calculations.
83          if($mysongnr == $mycount)
84          {
85             $Bcount = $Bcount + (($othersong['votes']) * ($othersong['votes'
                   ]));
86             $othersongnr = $othersongnr +  1;
87          }
88
89          if($othersongnr == $othercount)
90          {
91             $Acount = $Acount + (($mysong['votes']) * ($mysong['votes']));
92             $mysongnr = $mysongnr + 1;
93          }
94
95          //Compare the 'songIDs'
96          //If the songIDs are the same, the top of the cosine similairty is
                incresed,
97          if ($mysong['songID'] == $othersong['songID'])
98          {
99              $topcount = $topcount + (($mysong['votes']) * ($othersong['
                    votes']));
100             $Acount = $Acount + (($mysong['votes']) * ($mysong['votes']));
101             $Bcount = $Bcount + (($othersong['votes']) * ($othersong['votes
                    ']));
102             $mysongnr = $mysongnr + 1;
103             $othersongnr = $othersongnr +  1;
104         }
105         //If our party contains a songs, the other party don't we incese
                only our Acount making the similaity smaller.
106         elseif ( $mysong['songID'] < $othersong['songID'])
```

```
107             {
108                 $Acount = $Acount + (($mysong['votes']) * ($mysong['votes']));
109                 $mysongnr = $mysongnr + 1;
110 
111             }
112             //If the other party contains a song our party haven't played we
                    just moves to the next song.
113             else
114             {
115                  $Bcount = $Bcount + (($othersong['votes']) * ($othersong['
                         votes']));
116                  $othersongnr = $othersongnr + 1;
117             }
118         }
119         //Our topcount is only 0 if none of the parties compared have played
                the same song.
120         if ($topcount == 0)
121         {
122             $sim = 0;
123         }
124         //Calculates the cosine similairty.
125         else
126         {
127             $sim = $topcount/(sqrt($Acount)*sqrt($Bcount));
128         }
129         return $sim;
130     }
```

Listing 5.7: calsim function

After calculating the similarity, we output the songs having the highest rating at the parties with the highest similarity to our party. We are outputting at most 10 songs. The songs are stored in the database so we don't need to run this algorithm every time a user wants to see suggestions. When a new song is played at a party, we update the recommendations for this party. Since the result of the similarity calculations only change when a new song is added we save a lot of calculation, because we might have to show the recommended songs many times.

5.2 External Music Service

In this section we will describe the external music service used by our system, and why we chose it.

In our system we have chosen to use an external music service. This choice was made to deal with the limited resources, and we felt designing our other components of the system was more important, and that implementing a service which would do the same as an external service like Grooveshark, would be too time-consuming and is not our focus. The external music service we've chosen is Grooveshark.

Grooveshark has several attractive components, which makes it ideal for our system. These features includes:

Free to use
The use of Grooveshark music playing is completely free.

Large music library
The music library is supplied by users uploading music, which has resulted in a large library.

Client-less
Grooveshark is a rich internet application, which means that it basically has the functionality of a desktop application, but on the internet as a web service.

Simple public API
This is attractive for us, as we have limited previous experiences working with APIs, and a simple API will make it easy for us to use.

Simple to use JavaScript API
Same as with the public API.

We want to use Groovesharks API and with this, implement search functionality for our mobile application. Grooveshark also has a decent music player which we also want to use, instead of implementing our own, as we can control the music player with the JS API.
Using Grooveshark is not completely without risks.

- Legal issues
- Dependency

In Section 1.2.2 we mentioned the legal issues of Grooveshark, and that it has been blocked by some ISPs in counties such as Denmark. The reason for these legal issues is that Grooveshark haven't at all times followed the copyright laws. This might have an impact to our system, since we use Grooveshark to stream music, much like djtxt from Section 1.2.2. We still feel that using Grooveshark is a good choice, and people who usually uses Grooveshark either has an ISP that doesn't block it, or knows a way around the block. However, we feel like it's necessary to point out that Grooveshark **isn't illegal**, and we are not doing anything illegal.

Besides the legal issues, we also face the risk of using an external system. If Grooveshark's service is down, people will not be able to use our system either. One could argue that this is a bad choice, but the same risks would still be there if we didn't, i.e if we implemented our own service, it could go down as well. From our experience there aren't much downtime of Grooveshark's service. Since we're using Grooveshark's APIs, we might also have to change our system, if their API changes.

5.2.1 Connecting to Grooveshark

To connect and control the music on Grooveshark we can use their JavaScript API. The scripts, however, need to be run on their site, Grooveshark.com. We want to create an *overlay* on their site, to make it feel and look more like our system, rather than Grooveshark itself. For this we need to use a bookmarklet, much like djtxt from Section 1.2.2. This bookmarklet will call a JavaScript on our server, where we can then control the music from.

This JavaScript on our server has two purposes. The first is to make the overlay on Grooveshark.com, and the other is to add songs to Grooveshark's queue and start the music.

Controlling the Music

Since we need to access which songs to add to Grooveshark from our server, we need to allow Grooveshark to access a script that will output the songs to be added. This is done by adding a header to our makeplayer.php script as follows:

```
1 header('Access-Control-Allow-Origin: http://grooveshark.com');
```

Listing 5.8: Cross-site header

We have to add this because cross-site HTTP requests are not allowed normally online for security reasons; if anyone could access other sites' scripts, it would be a disaster.

After allowing Grooveshark to read from our script, we need to make a HTTP request, by making a XMLHttpRequest object (line 18). This object is used to exchange data with a server. We make a "GET" request (line 19), to get the playlist from a partyID, which we prompt the user for (line 12). We then add the highest voted song, by calling addSongsByID() from Grooveshark's JS API (line 31), with the song's ID. Then we have a function that is run every second. This function has a counter that counts the number of iterations, which is used to make sure only one new song is added. We calculated the duration of the song, and get the current duration. Then if there's less than 10 seconds ($1000ms \cdot 10$) left, and the count is about the required number, we add a new song by calling the function recursively, which resets the counter. We also make sure that we don't calculate anything when the song is paused, as seen by line 41.

```
7  function addSong(partyID)
8  {
9      // prompts for a party ID if opened for the first time
10     if(partyID == "-1")
11     {
12         partyID = window.prompt("Enter Party ID");
13         window.setTimeout(function() {window.Grooveshark.play();}, 1000*3);
14     }
15
16     if(partyID > 0)
17     {
18         xmlhttp=new XMLHttpRequest();
19         xmlhttp.open("GET", "http://bogosongs.com/makeplayer.php?partyID="+
               partyID, false);
20         xmlhttp.send();
21
22         // State 4 = ready, status 200 = ok
23         if (xmlhttp.readyState==4 && xmlhttp.status==200)
24         {
25             var songs = [];
26             songs = xmlhttp.responseText.split(",")
27             for(var i in songs)
28             {
29                 songs[i] = +songs[i];
30             }
31             window.Grooveshark.addSongsByID([songs[0]]);
32             overlay(partyID);
33         }
34
35         //Used to count number of intervals
36         count = 0;
37
38         clearInterval(intID);
39         //Loads song played duration and adds new songs
40         var intID = window.setInterval(function()
41         {
42             //We only want to do something when its playing
43             if(window.Grooveshark.getCurrentSongStatus().status == "playing"
                   )
44             {
45                 //Calculated total duration of the song
46                 var duration = window.Grooveshark.getCurrentSongStatus().
                       song.calculatedDuration;
```

```
47                  //How long it has been playing
48                  var position = window.Grooveshark.getCurrentSongStatus().
                        song.position;
49                  count++;
50                  //count makes sure that we only add one song each time
51                  if(position > duration - 1000*10 && count > 5)
52                  {
53                      //Calls the function recursively
54                      addSong(partyID);
55                  }
56              }
57          }, 5000);
58      }
59  }
```

Listing 5.9: addSong JavaScript function

Creating the Overlay

To make it seem more like our system when we are using Grooveshark, and also restrict the user from Grooveshark functionality, we implemented an overlay on Grooveshark's website. We create this overlay by calling a JavaScript function that appends to Grooveshark's website, with a higher z-index that makes our overlay appear on top. This overlay haven't been a primary focus, and so it is very simple and haven't been designed properly as seen by a screenshot of the overlay in Figure 5.2.

#	Song	Artist
1	Moon River	Frank Sinatra
2	West Side Story Medley	Jazz
3	Let me entertain you	Queen
4	White Heat	Jazz
5	Here Comes Santa Claus	Jazz
6	I Belong In Your Arms	Chairlift
7	Met Before	Chairlift
8	Something	Sarah Vaughan
9	Something	Jim Sturgess
10	Sidewalk Safari	Chairlift

Figure 5.2: Grooveshark Overlay

The function to make the overlay is seen in Listing 5.10. This function creates a full paged blue screen, and appends a start/stop button on it. The table in the overlay comes from calling the gsview.php script which gets the top ten most voted songs from a party.

```
48 function overlay(partyID)
49 {
50     //Create overlay and append to page
51     var overlay = document.createElement("iframe");
52     overlay.setAttribute("id","overlay");
53     var str1 = "http://bogosongs.com/gsview.php?partyID=";
54     var str2 = partyID.toString();
55
56     var url = str1.concat(str2);
57         overlay.setAttribute("src",url);
58     overlay.setAttribute("style","background-color: #B4E9EC; opacity: 1;
           filter: alpha(opacity=100); position: fixed; top: 0; left: 0; width:
           100%; height: 100%; z-index: 6777271; ");
59     document.body.appendChild(overlay);
60
61     //Create a box for the control buttons.
62     var box = document.createElement("div");
63     box.setAttribute("id","box");
64     box.setAttribute("style","background-color: #FFFFFF; opacity: 1; filter:
           alpha(opacity=100); position: fixed; top: 70%; left: 45%;  z-index:
           6777273; ");
```

```
65
66      //Create a button the play/pause the music
67      var playpause=document.createElement("BUTTON");
68      var pptxt=document.createTextNode("play/pause");
69      playpause.appendChild(pptxt);
70      playpause.setAttribute("id","playpausebtn");
71
72
73      playpause.onclick = function(){window.Grooveshark.togglePlayPause();};
74
75      box.appendChild(playpause);
76
77      //Create a button to skip a song.
78      var nextbtn=document.createElement("BUTTON");
79      var nextbtntxt=document.createTextNode("next");
80      nextbtn.appendChild(nextbtntxt);
81      nextbtn.setAttribute("id","nextbtn");
82      nextbtn.onclick = function () {addSong(partyID); window.setTimeout(
            function() {window.Grooveshark.next();}, 1000*2);   };
83
84      box.appendChild(nextbtn);
85      document.body.appendChild(box);
86  }
```

Listing 5.10: Grooveshark overlay function

5.3 Application

This section is about the implementation of the smartphone application. This section will contain code examples and explanation of some of the used Android elements.

5.3.1 Overview

In this section we will show an overview of the application using a class diagram. Because of the size of the system, we had to split the overview into two diagrams; Figure 5.3 and Figure 5.4. Figure 5.3 show a complete overview of all the Android activities, and how they're navigated. Because of the size, we had to remove the navigation from any activity to the main screen (using the home button), to simplify the diagram. Figure 5.3 doesn't contain the 4 classes that is very often used; Song, Party, ServerConnecter and NetworkConnection. The dependencies between these classes and the activities are shown in Figure 5.4. A dotted line represents an activity navigating to another activity. A full line represents activity creating (and saving) an object of that class. A triangle before an attribute means that it's package visible, meaning that the whole package (application) can see and use these attributes. A line under an attribute means that the attribute is static. A small "c" in a method icon means that it is a constructor. Be aware that Figure 5.4 doesn't show correct connection between activities, but simply the connection between activities and the respected classes.

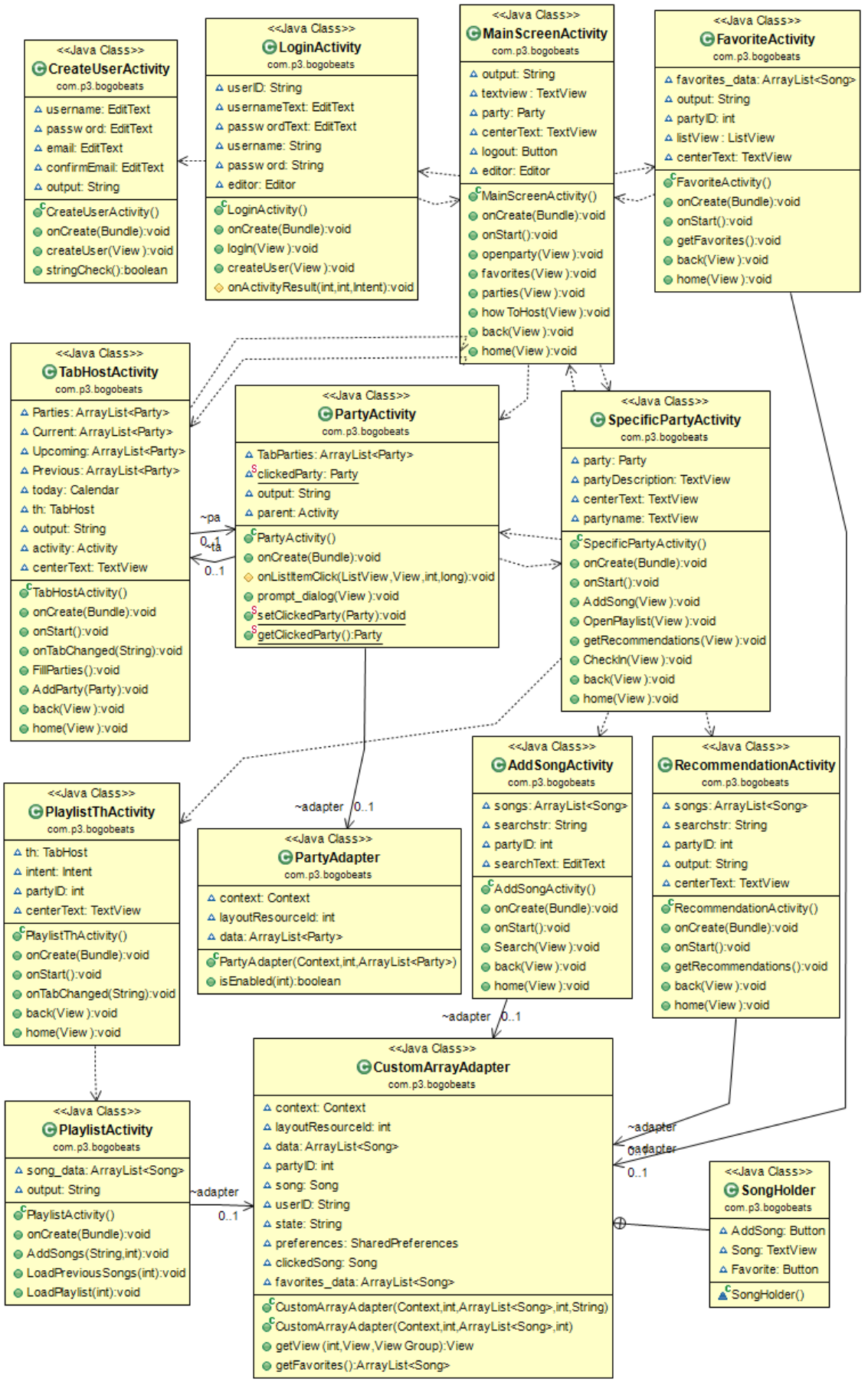


Figure 5.3: Application class diagram

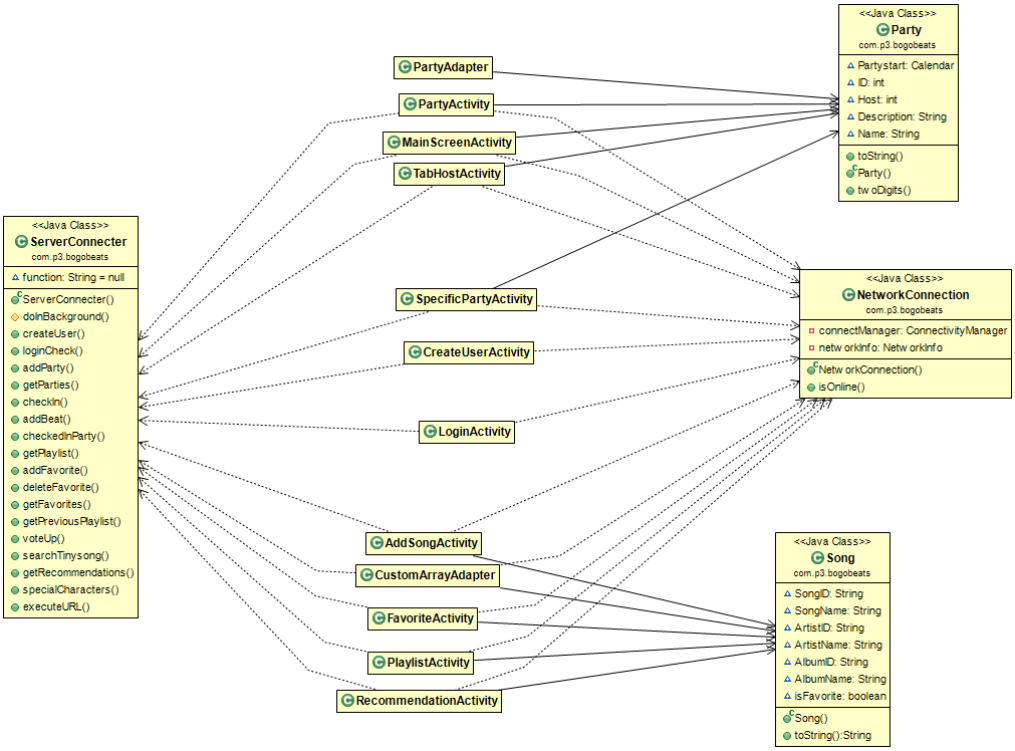


Figure 5.4: Application class diagram with dependencies

5.3.2 The ServerConnecter Class

We use the application to connect to our server several times. For this, we needed a way to connect and interact with the server, which we could use for all connections easily.

For safety reasons, Android doesn't allow the developer to use the main thread to perform network operations. Android has a class called AsyncTask that easily allows us to do something in a separate thread, which is where we will establish our connections to the server. The ServerConnecter class extends the AsyncTask, allowing us to use the methods from the AsyncTask, when creating new ServerConnecter objects. Our activities will call functions using these objects. We will give an example of how a user is created.

```
41 output = new ServerConnecter().createUser(username.getText().toString(),
       password.getText().toString(), email.getText().toString());
```

Listing 5.11: Example use of the NetworkConnecter class - Creating a user

This will call the function shown in Listing 5.12.

```
65 public String createUser(String username, String password, String email)
66 {
67     function = "makeuser";
68     //Have to percent-code special characters for all user text input
69     return executeURL(String.format("&input=%s|%s|%s", specialCharacters(
           username), specialCharacters(password), specialCharacters(email)));
70 }
```

Listing 5.12: Example use of the NetworkConnecter class - Creating a user

The function in Listing 5.12 is located in the ServerConnecter class. As seen by the function, we use another function called specialCharacters on the inputs. We do this to percent-encode the input to avoid unauthorized SQL queries, as well as allowing our users to use a wider amount of characters for their usernames, passwords and to make

sure they can use their emails. This will then return the output from the executeURL function shown in Listing 5.13.

```
165 public String executeURL(String extraURL)
166 {
167     try
168     {
169         //Have to percent-encode the "|" when calling the url
170         return execute(extraURL.replace("|", "%7C")).get(10, TimeUnit.
                SECONDS);
171     }
172     catch (TimeoutException e)
173     {
174         e.printStackTrace();
175     }
176     catch (InterruptedException e)
177     {
178         e.printStackTrace();
179     }
180     catch (ExecutionException e)
181     {
182         e.printStackTrace();
183     }
184     return null;
185 }
```

Listing 5.13: Example use of the NetworkConnecter class - Creating a user

The executeURL function will then return the output from the server after calling execute, which sends the parameters to the doInBackground method in Listing 5.14, which creates a HTTP requests to the server with the right URL, and returns the output all the way back to the activity that called the function. It may not seem like a good idea to do it this way, but we save a lot of unnecessary code by having these links. It does also make it a lot easier to call the functions from the activities.

```
26 protected String doInBackground(String... input)
27 {
28     String hostname = "http://bogosongs.com/";
29     String urlStart = "Apitest.php?function=";
30     String result = "";
31     {
32         //Connects to the server with the right URL call
33         HttpClient httpClient = new DefaultHttpClient();
34         String url = hostname + urlStart + function + input[0];
35         Log.d("URL", url);
36         HttpGet req = new HttpGet(url);
37         HttpResponse response;
38         try
39         {
40             response = httpClient.execute(req);
41             InputStream in = response.getEntity().getContent();
42             BufferedReader bufferReader = new BufferedReader(new
                    InputStreamReader(in));
43             StringBuilder str = new StringBuilder();
44             String line = null;
45
46             // Reads the output from the url and writes it to str
47             while ((line = bufferReader.readLine()) != null)
48                 str.append(line);
49             in.close();
50             result = str.toString();
51         }
52         catch (ClientProtocolException e)
```

```
53          {
54              e.printStackTrace();
55          }
56          catch (IOException e)
57          {
58              e.printStackTrace();
59          }
60      }
61      return result;
62  }
```

Listing 5.14: Example use of the NetworkConnecter class - Creating a user

5.3.3 The NetworkConnection Class

For safety reasons, we must always make sure that we have a network connection, before trying to connect to our server, otherwise the application would crash if it's trying to make a HTTP request while being offline. Since we do this often, we implemented a class, with a boolean method to return true if we have a connection, and false if we don't. Listing 5.15 shows the NetworkConnection class, where we can call isOnline (as shown by Listing 5.16) to get the network information we want.

```
1  public class NetworkConnection
2  {
3      private ConnectivityManager connectManager;
4      private NetworkInfo networkInfo;
5
6      public boolean isOnline(Context context)
7      {
8          connectManager = (ConnectivityManager)context.getSystemService(
               Context.CONNECTIVITY_SERVICE);
9          networkInfo = connectManager.getActiveNetworkInfo();
10         if (networkInfo != null && networkInfo.isConnected())
11             return true;
12         else
13             return false;
14     }
15 }
```

Listing 5.15: The NetworkConnection class

```
1  if (new NetworkConnection().isOnline(this))
2  {
3      //Do stuff
4  }
```

Listing 5.16: Calling isOnline

5.3.4 ListAdapter

In our Android application we have multiple places where the use of lists are very applicable, e.g. the playlist and the list of favorite songs. We retrieve this information from the database and map it in ArrayLists, which is a rich list interface for Java.

In the ArrayList of songs, which is the playlist, we have song objects. These song objects contain different attributes, for example the song ID (a unique identifier for every song), however a user of the system needs this information, so we only want to map some information to the user, in this case it is the "ArtistName - SongName" string. We override the toString() method in our Song class, so that we can toString the songs

properly. In the favorite screen, we want the user to be able to remove favorites and add a favorite song to the checked in party. To do this the user interface for the favorite list should consist of the song-strings and two buttons; remove favorite and add song to playlist for all songs. The result is shown in Figure 5.5:

Figure 5.5: ListView item

To map the toString() and the respective buttons data for every song-object in our ArrayList, we use the ListView class. This class is basically an easy-to-use graphic user interface list-displayer. To translate and display the data via the ListView, one can use a list adapter. A ListAdapter uses the Adapter pattern(wrapper pattern). The design pattern is used to translate an interface for a class, so that it is compatible with another. In this example we want to translate the contents of our ArrayList to our ListView, so we use the ListAdapter. The Android developers have already provided an implementation of the adapter, however this cannot be used when you want buttons. We had to create our own custom adapter for the ListViews, which adds the buttons. The CustomArrayAdapter is created with respect to this video: [11].

The custom adapter inherits from Androids built-in adapter; the ArrayAdapter. We have also created an XML file containing what each element of the ListView should be. This data is represented in code in the adapter like this:

```
1 static class SongHolder
2     {
3         Button AddSong;
4         TextView Song;
5         Button Favorite;
6     }
```

Listing 5.17: Custom ArrayAdapter songolder

Then the getView method has to be overwritten from the superclass, and in here we can add our button events and list data. The calling of this method is handled by the OS. It's main function is to basically fix the correct data at the correct position of the ListView and smart memory handling.

5.3.5 Gson

Gson is an open-source Java library by Google, available at http://code.google.com/p/google-gson/. We use Gson for converting JSON (JavaScript Object Notation) strings into their representative Java objects.

The JSON strings are received from the server to the application. They are received in a JSON string format and they look like this:

[{"SongID":6131653,"SongName":"Somebody to Love","ArtistName":"Queen"},{"SongID":11085636,"SongName":"Satin Doll","ArtistName":"Jazz"}]

Where square brackets denote an array and curly brackets denote array elements. GSON makes it very easy to convert this data:

```
1 Gson gson = new Gson();
2 Type collectionType = new TypeToken<ArrayList<Song>>(){}.getType();
```

```
3 ArrayList<Song> gsonsongs = gson.fromJson(output, collectionType);
4 for (Song gsong : gsonsongs)
5         song_data.add(gsong);
```

Listing 5.18: GSON code

In this code piece, output is the JSON string. We also have a Song class which has matching attributenames with the JSON attributenames.

In line 1 we instantiate the Gson object. In line 2 we use the Java supertype "Type" to get an object of the generic collection type: "ArrayList¡Song¿" since we want to store of song objects in an ArrayList. In line 3 we use the fromJson method which takes a String and a Type as input. Now we have our JSON as song objects and we can use these song elements in the GUI song list in which is added in line 4-5. Then the getView method has to be overwritten from the superclass, and in here we can add our button events and list data.

5.3.6 TabHost

As seen in our smartphone navigation map from Section 3.2, our application uses tabs for navigation in places were two screens are very closely related. Tabs are made in Android by creating a TabHost object. A TabHost consist of a layout and a TabHost activity. The layout must consist of the tabs, but may also include other layout elements, for example in our layout it also includes a navigation bar. To use the TabHost layout together with a layout from another activity, the TabHost activity is opened first.

When we create the tabs we use various methods from the TabHost object. These methods are addTab(), which adds another tab, newTabSpec(), which we use to create a tabId for the new tab, setContent() which sets the content to be that of another activity and setIndicator() which is the text for text tab. En example of creating two tabs can be seen in Listing 5.19

```
39 th.addTab(th.newTabSpec("Upcoming").setContent(createTabIntent).setIndicator
       ("Upcoming"));
40 th.addTab(th.newTabSpec("Previous").setContent(createTabIntent).setIndicator
       ("Previous"));
```

Listing 5.19: How we create new tabs

There are different ways of managing the tabs using a TabHost, the way we do it is by using the tabId to figure out which tab is currently active, and then we change the content that is shown according to the active tab. Tabs can also be managed by starting different activities in the TabHost activity, but since our tabs are very similar, we think it is easier not to have different activities for each tab.

As seen on Listing 5.20, the way we change content according to the tabs is through simple if-statements to check the tabId and then we use two different methods for loading the content of the active tab, specifically the LoadPlaylist() and the LoadPreviousSongs() methods.

```
29 adapter = new CustomArrayAdapter(this, R.layout.
       addsong_favorites_item_layout, song_data, partyID, tabId);
30 setListAdapter(adapter);
31 if (tabId == "Playlist")
32 {
33     LoadPlaylist(partyID);
34 }
35 if (tabId == "Previous")
36 {
37     LoadPreviousSongs(partyID);
38 }
```

Listing 5.20: How we change the content in tabs

5.4 Website

This section is about the implementation of the website. The website can be visited at at www.bogosongs.com. This section will contain code examples in PHP, JavaScript and some HTML to illustrate the key elements of the website. Screenshots of the website can be found in Appendix C.1.

5.4.1 Overview

The website consists of a header and 3 tabs. In the header the user controls the navigation on the website and can login, creates an account or logout. The first tab is the "home" tab which is the welcoming tab. This tab provides the user with the bookmarklet for the player, and a start-up guide for new users. The second tab is the "Party administration" tab. In this tab the user can create a party, show and edit created parties, delete a party or simply show the attendees of the party. The last tab is the "about" tab where the user can read short information about us and about the website. The visual part of the website is written in HTML with CSS provided by Bootstrap. By using CSS from Bootstrap, we save resources designing and end up with a simple and user-friendly design. The visual design of the website reappears in the smartphone application, causing the user to identify the one with the other. Beneath the visual part, we use PHP code and JavaScript to control the website's appearance and to connect with the server to make changes dynamically. The reason why the domain name is called Bogosongs and not BogoBeats is because we bought the domain early in the project, and afterwards changed the name into BogoBeats.

5.4.2 Header

The header is located in the top of the website. In the header the user can login, create an account or navigate through the 3 tabs. When the user is logged in, it is possible to logout as well. When the user click on either "Create user" or "Login", a box will drop down from the header. In the "Login" box the user can type in a username and a password and click login to proceed. In the "Create user" box the user can type in a username, a password and an e-mail, and then click on "Create" to make a new user and proceed. Creating a new user is implemented with the use of PHP to check the input, and connecting to the database via the bogoAPI. When the user clicks "Create" the input LoginState, CrName, CrPassword, CrEmail1 and CrEmail2 is posted to bogosongs.com using simple HTML. The LoginState is posted as "createuser". The PHP code then catches the LoginState being posted, and the following code will be executed.

```
83 // Create a new user.
84 else if (($_POST['LoginState']) == 'createuser'){
85     // Makes sure that Name, Password, Email1 and Email2 are not empty and
       checks if Email1 and Email2 are equal befor proceeding.
86     if (!((empty($_POST['CrName'])) || (empty($_POST['CrPassword'])) || (empty(
       $_POST['CrEmail1'])) || (empty($_POST['CrEmail2']))) && (($_POST['
       CrEmail1']) == ($_POST['CrEmail2'])) ){
87         // inititalizing array with the new party variables.
88         $newuserarray[0] = $newName;
89         $newuserarray[1] = $newPassword;
90         $newuserarray[2] = $newEmail;
91         // Call makeuser function from bogoAPI.
```

```
92          $output = $bogoapi->makeuser($newuserarray);
93          // If the user were created the user will be logged in.
94          if ($output == 'success'){
95              // call logincheak functon from bogoAPI.
96              $userID = $bogoapi->logincheck($newuserarray);
97              //Check if user was logged in or not. If user wasn't found,
                    logincheck  returns 0.
98              if ($userID >= 1){
99                  $usertext = $newName;
100             }
101             else{
102                 $usertext = "Wrong login try again";
103             }
104         }
105         else{
106             $usertext = "Could not create user";
107         }
108     }
109 }
```

Listing 5.21: Header.php : Create Party

This piece of code includes 2 functions from the bogoAPI, the makeuser and logincheck functions. Both functions are also used by the application and will not be described further in this section. If the user was successfully created, the user will automatically be logged in. If the user was successfully logged in or an error occurred, the variable usertext will be printed in the header. It is important to acknowledge that the logincheck functions returns the ID of the specific user. This userID is used throughout the website as an identity for the user. When the user is logged in to the site, the user can choose to logout. When the user clicks the logout button, the HTML code will post LoginState as "logmeout". The PHP code catches LoginState as "logmeout" and the site is loaded without a userID since the site can't recognize any user, and the user is logged out.

The 3 tabs are created within 3 divs also called division tag in HTML. These divs IDs are "home", "partyadmin" and "about". Within each div the content of each tab is written. It's possible to hide and show different content without a new browser load, simply by hiding and showing divs. An additional div with the ID "notloggedin" is made to replace the "partyadmin" div if the user isn't logged in. To hide and show different divs, we use JavaScript embedded in HTML. The 2 functions are identical, only difference is that one show divs and the other hide divs. The functions take the specific divs id as a parameter, finds the div and then turn the visibility style to hidden or visible, depending on the function used. Here is an example of the showDiv function.

```
43 function showDiv(pass) {
44     var divs = document.getElementsByTagName('div');
45     for(i=0;i<divs.length;i++){
46         // If the div is found.
47         if(divs[i].id.match(pass)){
48             // style : visibility is set to visible. Showing the div.
49             divs[i].style.visibility="visible";
50         }
51     }
52 }
```

Listing 5.22: Header.php : Show Div function

When an action is taken on the site, the site reloads with the posted data. To keep track of which tab is being active, a variable is implemented. This variable is called "navbar", and is being posted when the user makes actions on a page to keep an action active. Here is the code behind the "Party administration" button, to give an example of what is happening when the user is using the header to navigate.

```
140 <!-- Party Administration Button within list tag. Followed by divider. -->
141 <!-- If the user isn't logged in the 'notloggedin' tab will be shown instead
        . -->
142 <li class="<?php if (($_POST["navbar"]) == 'partyadmin'){ echo 'active '
        ;}?>" value="partyadmin">
143     <a href="<?php if ($userID < 1){echo "javascript:showDiv('notloggedin')
            ;";} else{echo "javascript:showDiv('partyadmin');";}?> javascript:
            hideDiv('about'); javascript:hideDiv('home');" style="font-weight:
            bold;">
144         <i class="icon-cog icon-white"></i> Party administration
145     </a>
146 </li>
```

Listing 5.23: Header.php : Party administration

5.4.3 Home tab

The home tab is the first tab the user will experience when entering the site. The site include a welcome with the BogoBeats logo attached followed by a guide. The guide is meant to be a beginner's start-up guide. In the guide the user is shown how to create a user, create a party, placing the bookmarklet, use the bookmarklet and how to add the party on the app. The most important feature of this tab is that it provides the bookmarklet, which is to be dragged to the bookmarks bar.

5.4.4 Party administration tab

When the user is logged in and clicking on the "Party administration" button in the header, the "Party administration" tab will appear. When the tab first appears it have 2 columns. The first column shows the parties the user has created. In the second column the user can create a new party or edit an existing one. The first column have the caption "My parties" followed by a table of parties. Within each table row the party's name, ID and date are displayed. At the end of each table row there is an edit button, the edit button is described later in this section. The table rows are created by embedding PHP code within HTML code. Here is a code example of how the table rows are created within the table body.

```
169 <tbody>
170 <?php
171     $tbl_name="party"; // Table name for $sql
172     // Finding parties where UserID is host.
173     $partyrows = mysql_query("SELECT * FROM $tbl_name WHERE host = '$userID'
            ");
174     //Runs through all the list of parties found from $query.
175     while($parties = mysql_fetch_array($partyrows))
176     {
177         //Gets the partyID
178         $partyID = $parties['partyID'];
179         // Finding party with specific partyID.
180         $partyrow = mysql_query("SELECT * FROM $tbl_name WHERE partyID = '
                $partyID'");
181         $party = mysql_fetch_array($partyrow);
182         // Printing each party in a table row.
183         // if the party is the current party, it is highlighted with a green
                and partyID is written with strong.
184         if (($party['partyID'] == $CurrentPartyId) && (($_POST["
                CreateOrEditBox"]) == 'Edit')){
185             echo '<tr class="success">'; // Tablerow.
186                 echo "<td>"; // Tablecolumn.
```

```
187                     echo $party['name'];
188                 echo "</td>";
189                 echo "<td><strong>";
190                     echo $party['partyID'];
191                 echo "</strong></td>";
192                 echo "<td>";
193                     echo $party['dato'];
194                 echo "</td>";
195         }
196         // If party is normal its written normal.
197         else {
198             echo "<tr>";
199 
200                 echo "<td>";
201                     echo $party['name'];
202                 echo "</td><td>";
203                     echo $party['partyID'];
204                 echo "</td><td>";
205                     echo $party['dato'];
206                 echo "</td>";
207         }
208                 echo "<td>";
209                     // The Edit button is created.
210                     echo '<form action="http://bogosongs.com" method="post"
                        >
211                         <input type="hidden" name="navbar" value="partyadmin
                            "/>
212                         <input type="hidden" name="UserID" value="';
213                         echo $userID;
214                         echo '"/>
215                             <input type="hidden" name="CurrentId" value="';
216                         echo $party['partyID'];
217                         echo '"/>
218                             <input type="submit" value="Edit" align="right"
                                class="btn  btn-primary btn-block pull-right
                                "/>';
219                         echo '<input type="hidden" name="CreateOrEditBox"
                            value="Edit"/>';
220                     echo '</form>';
221                 echo "</td>";
222             echo "</tr>";
223     }
224     ?>
225 </tbody>
```

Listing 5.24: Index.php : Table of my parties

The PHP code is echoing HTML code in a loop, this result in a dynamic list of parties.

The second column is where the user can create a party. The user can enter party name and description, set date and time and then click the “Create party” button. When the user click the button, 6 variables are posted. The variables CpName, CpDescription, CpDate and CpTime contains the information the user have typed into the second column. CpHost contains the user’s ID, the last variable is called state. Posting the state variable indicates that either the site shall update an existing party or create a new party. In this case, state is posted as “createparty”. After the post, the PHP code catches the state as “createparty” and creates the new party by first checking the posted variables then creating an array consisting the data and then calling the createparty function from the bogoAPI. This is shown in this example.

```
26 if (($_POST["state"]) == "createparty"){
27     //Uses the $_Post method to get values to the new party.
```

```
28      if (!((empty($_POST['CpName'])) ||(empty($_POST['CpDescription'])) ||(
            empty($_POST['CpDate'])) ||(empty($_POST['CpTime'])) ||(empty($_POST['
            CpHost'])))){
29          // party information into array.
30          $newParty[0] = ($_POST['CpName']);
31          $newParty[1] = ($_POST['CpDescription']);
32          $newParty[2] = ($_POST['CpDate']);
33          $newParty[3] = ($_POST['CpTime']);
34          $newParty[4] = ($_POST['CpHost']);
35          // Calling createparty function from bogoAPI with the array which
                creates the party.
36          $bogoapi->createparty($newParty);
37      }
38  }
```

Listing 5.25: Index.php : Create party

The createparty function adds the party to the party list, but also adds the host as an attendee to the new party. The createparty function is shown in this example.

```
540 public function createparty(array $array)
541 {
542     $PartyName = $array[0];
543     $PartyDescription = $array[1];
544     $PartyDate = $array[2];
545     $PartyTime = $array[3];
546     $PartyHost = $array[4];
547
548     $tbl_name="party"; // Table name
549     // Creating a new party with the party variables in the database table '
            party'.
550     mysql_query("INSERT INTO $tbl_name (name, describtion, dato, time, host)
             values ('$PartyName', '$PartyDescription', '$PartyDate', '
            $PartyTime', '$PartyHost')") or die($status = "create party Error");
551     // Getting the partyID of the new party just created.
552     $newpartyid = mysql_query("SELECT `partyID` FROM party WHERE name='
            $PartyName'");
553     while($aPartyID = mysql_fetch_array($newpartyid))
554       {
555      $thisnewpartyid = $aPartyID['partyID'];
556       }
557     // Adding the host to the party in the database table 'upcoming_parties
            '.
558     mysql_query("INSERT INTO `upcoming_parties` (userID, partyID) values ('
            $PartyHost', '$thisnewpartyid')") or die($status = "Add hoste to
            party Error");
559 }
```

Listing 5.26: BogoAPI : Create party function

When the user clicks on “edit” in the list of parties, the edit form replaces the create party form. When the edit party form loads, the information about the party will be automatically be filled into the input boxes. This happens in a way similar to the list of parties described earlier. The user can now change some of the data and click the “Save changes” button. When the user clicks the button, the new party data will be posted similar to creating a party. The ID of the current party will be posted and the state will be posted as “updateparty”. After the post, the PHP code catches the state as “updateparty” and the following is similar to creating a party.

```
40  else if (($_POST["state"]) == "updateparty"){
41      //Uses the $_Post method to get values to the party update.
42      if (isset($_POST["CurrentId"]) && (!empty($_POST["UpName"]))&& (!empty(
            $_POST["UpDescription"]))&& (!empty($_POST["UpDate"]))&& (!empty(
```

```
        $_POST["UpTime"]))){
43          // new party information into array.
44          $PartyInfo[0] = $_POST["CurrentId"];
45          $PartyInfo[1] = $_POST["UpName"];
46          $PartyInfo[2] = $_POST["UpDescription"];
47          $PartyInfo[3] = $_POST["UpDate"];
48          $PartyInfo[4] = $_POST["UpTime"];
49          // Calling updateparty function from bogoAPI with the array which
                updates the party information.
50          $bogoapi->updateparty($PartyInfo);
51      }
52  }
```

Listing 5.27: Index.php : Update party information

The updateparty function from the bogoAPI is running an update where the partyID is equal to the current party's ID.

```
508 public function updateparty(array $array)
509 {
510     $PartyID = $array[0];
511     $PartyName = $array[1];
512     $PartyDescription = $array[2];
513     $PartyDate = $array[3];
514     $PartyTime = $array[4];
515
516     $tbl_name="party"; // Table name
517     //updates party in database 'party'.
518     mysql_query("UPDATE $tbl_name SET
519     name ='$PartyName', describtion ='$PartyDescription',
520     dato = '$PartyDate', time = '$PartyTime'
521     WHERE partyID='$PartyID'") or die($status = "Update party Error");
522
523 }
```

Listing 5.28: BogoAPI : Update party function

The user can also delete a party. This result in the party being deleted from the "party" table in the database, but also result in deleting attendees from the party in the "upcoming_parties" table in the database. When the user click on the "Delete party" button, the user is asked to confirm or cancel. If the user click the "Confirm delete" button the variable "removeparty" is posted as "removeconfirmed". The PHP code catches the "removeparty" as "removeconfirmed" and then calls the removeparty function from the bogoAPI, with the "CurrentId" as a parameter.

```
18  if(($_POST["removeparty"]) == 'removeconfirmed'){
19      $deleteID = ($_POST["CurrentId"]);
20      $bogoapi->removeparty($deleteID);
21  }
```

Listing 5.29: Index.php : Delete party

The removeparty function from the bogoAPI then remove the party and attendees attending the party.

```
530 public function removeparty($PartyID)
531 {
532     // Remove party from 'party'.
533     mysql_query("DELETE FROM `party` WHERE partyID='$PartyID'") or die(
            $status = "Remove party Error");
534     // Remove host and attenders from the party in 'upcoming_parties'.
535     mysql_query("DELETE FROM `upcoming_parties` WHERE partyID='$PartyID'")
            or die($status = "Remove host from party Error");
```

```
536 }
```

Listing 5.30: BogoAPI : Delete party function

When the user click on edit, a third column also appears. In this column the attendees are written into a table similar to the list of parties described earlier. Here the PHP code is again executed within the table body.

```
376 echo '<tbody>';
377     // Get userID of users attending the party in database table "
            upcoming_parties".
378     $attendersrows = mysql_query("SELECT * FROM upcoming_parties WHERE
            partyID = '$CurrentPartyId'");
379     // Running through all users attending the party.
380     while($attenders = mysql_fetch_array($attendersrows))
381         {
382         $usersID = $attenders['userID'];
383         // Using UserID to get username of the attendee.
384         $sql = "SELECT * FROM p3_users WHERE userID = '$usersID'";
385         $userrow = mysql_query($sql);
386         $user = mysql_fetch_array($userrow);
387         // echo a table row for the attendee including the username.
388             echo '<tr><td><b>';
389                 echo $user['username'];
390             echo "</b></td></tr>";
391         }
392 echo '</tbody>';
```

Listing 5.31: Index.php : Creating attendees table rows

Chapter 6

Test

In this chapter we will describe all our tests. All the prototypes were tested by both our informants. All of the tests will have a combined introduction describing the purpose of the test, the test parameters, two individual parts describing how the test went and then a combined conclusion. Section 6.1 describes the first prototype test where we used a Powerpoint prototype. Section 6.2 describes our first Android application prototype test, where Section 6.3 describes the second application prototype test. Then in Section 6.4 we describe the system test we performed on our system.

6.1 Powerpoint Prototype

This section of the chapter describes the first usability test we made. This section contains the purpose of the test, a description of the prototype, evaluations of the results obtained by testing the prototype with the informants and a combined conclusion on which parts should be changed before we test the application again.

6.1.1 Purpose of the Test

The first prototype we tested on our informants was a simple Powerpoint-prototype. This prototype was made to test if the way we planned to handle the navigation between the screens of the Android application, made sense to people who was not a part of designing the application.

6.1.2 Description of the Prototype

The prototype was made in Powerpoint using simple pictures and hyperlink buttons, linking to other slides as buttons for navigation between the screens. The different screens can be seen in Figure 6.2. This description will not be about the layout of the different screens, but the way the navigation between the screens works as seen in Figure 6.1.

In order to make sure a user can only vote once on each song, we need to authenticate the user. This also makes it easier to receive the correct data from our database, as we can use the user's unique ID. In order to authenticate the users, the first screen of the application is a login screen. After the user is authenticated the main screen is loaded, Figure 6.2(b) and Figure 6.2(c) shows the two different main screens. The only difference is that after checking in to a party, the main screen adds a shortcut to the specific party screen. In addition to the shortcut, the main screen also contains a button to view songs the users have added to his/hers favorite list, and a button to view a list of parties the user has added. From the party screen the users can search for music by visiting the library

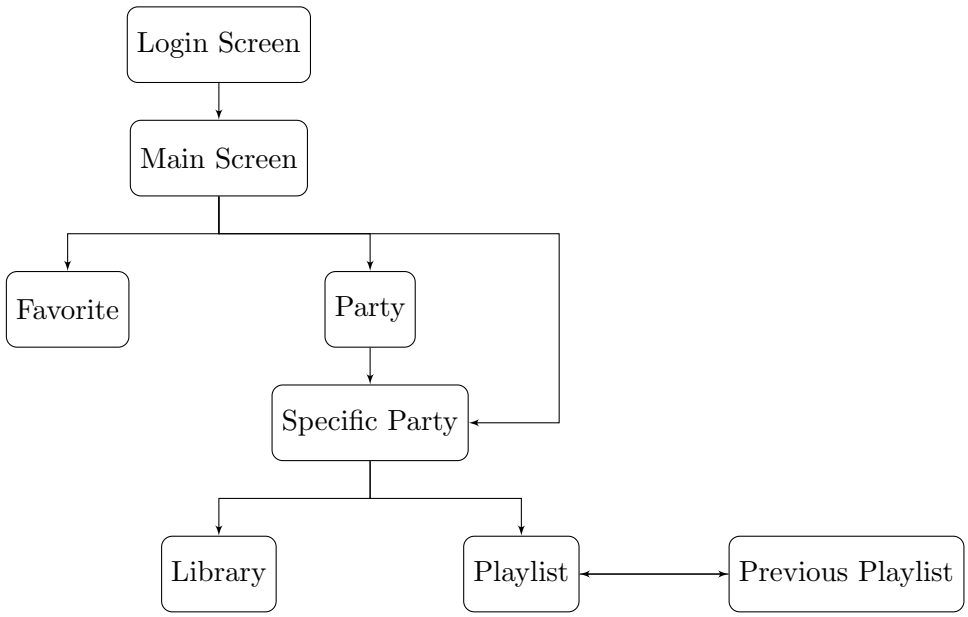


Figure 6.1: Navigation map of the Powerpoint-prototype

screen, or view the playlist by visiting the playlist screen. The playlist screen consist of two screens; one for the songs yet to be played and one for the songs already played.

6.1.3 Powerpoint Prototype Test: Ida

The test showed that the first thing to happen was pressing on the "Login with Facebook" button. She then tried to navigate around the different screens and understood the purpose of the buttons. She had no trouble going from screen to screen, but when she hit a "dead end", it was not obvious that she had to use the build-in Android back button. She thought that the design for the application was functional enough, but wanted some more colorful design and more buttons for navigation, such as a button for going directly from the playlist to the library.

The conclusion of this test is that the functional design of the application worked as intended apart from a few small navigation problems. It also showed that we have to think about the more colorful aspect of the design.

6.1.4 Powerpoint Prototype Test: Rikke

From the login screen (Figure 6.2(a)) the first button Rikke clicked was "Login with Facebook". She liked the idea about the application being connected to to Facebook, because it meant that she did not have to make a new user to be a part of the system. From the main screen (Figure 6.2(b)) there were no problems understanding the buttons, but the empty text field saying "Current party" let to a little confusion. The list showing the different parties added (Figure 6.2(e)) worked fine. Both adding parties and seeing parties were done without any problems. Understanding the three functions available for a specific party (Figure 6.2(f)), was not a problem, nor was understanding the buttons used in the library (Figure 6.2(g)). The playlist was a little problematic (Figure 6.2(h)) as Rikke did not see the previous tab, however she recognized the buttons from the library, and understood the meaning of these buttons. This consistency helped her to gain a better understanding of the screen. Once Rikke got to the previous screen (Figure 6.2(i)), she asked about the meaning of the square on the right side of the screen, which was a mess-up from our side as it should have showed a star instead.

6.1.5 Conclusion

Both the informants understood the navigation between the screens of the application (see Figure 6.1) when moving forward, however getting back only using the build-in back button did not work well. From the main screen (Figure 6.2(b)) there was a little confusion about the text box saying "Current party", which was just floating in the middle of nowhere, when the user was not checked in. From the playlist screen (Figure 6.2(h)) the "Previous" tab was not seen easily enough. There was some general consistency issues, like using different icons for the same thing, and not placing the same icons in the same side of the screen. The "Login with Facebook" on the login screen (Figure 6.2(a)) is a feature both of the informants liked, since that was something they have seen before, and knew how to use.

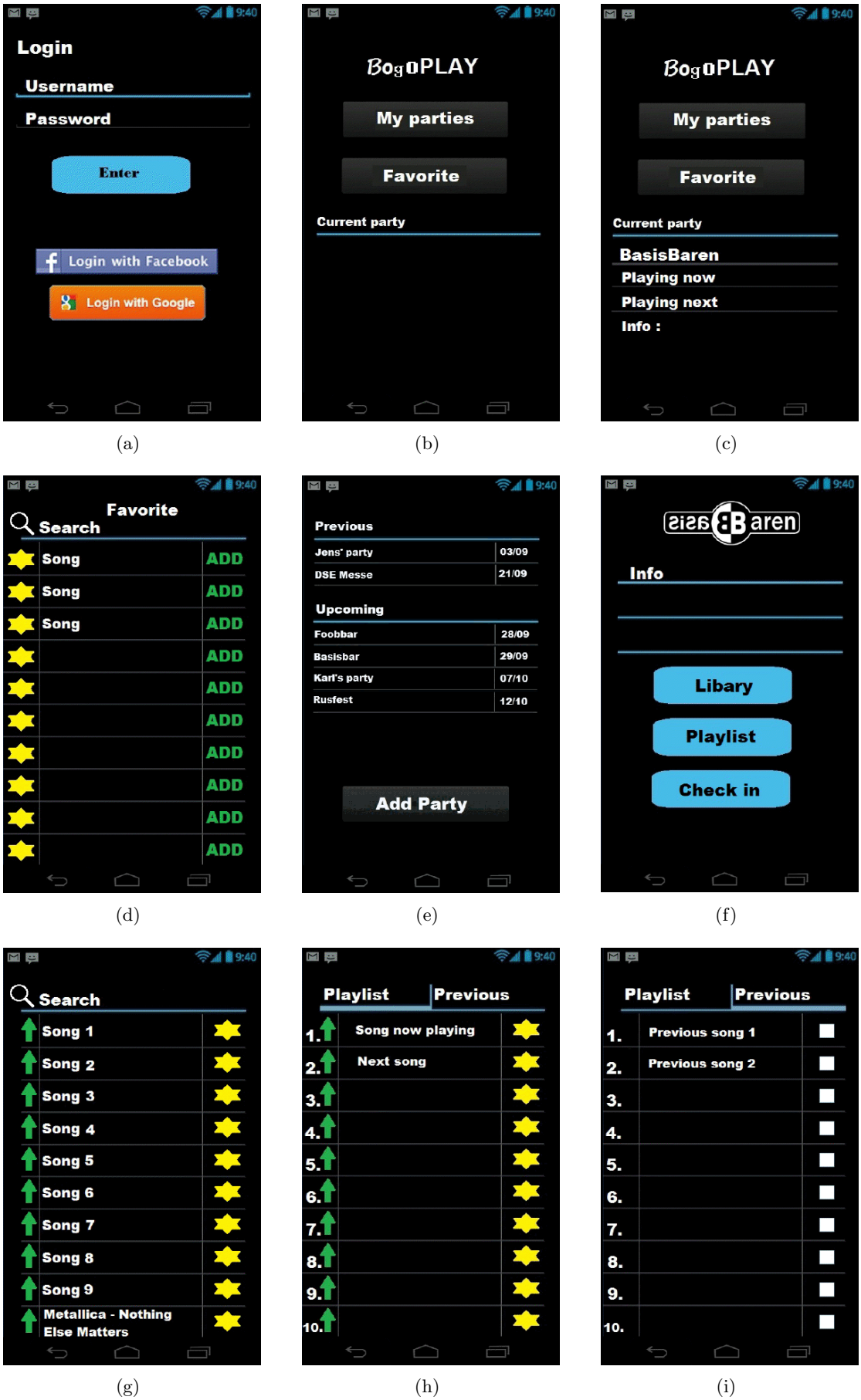


Figure 6.2: Screen from the Powerpoint prototype

6.2 The First Android Prototype

This section of the chapter describes the test of the first Android prototype. The method used for this test is different from Section 6.1, since we made a set of scenarios and tasks for the informants. By making such changes we were more in control of the way the prototype was tested.

6.2.1 Purpose of the Test

The purpose of the test was to gain knowledge about how our design works when it is on a smartphone. The scenarios and tasks were used to make sure the functionality of the application made sense to the users.

Scenarios and Tasks

The scenarios and tasks used in the test:
Scenario 1:
You have been invited to a party, and would like to use the party's music system. To use the system, you must first create a user to login with.
Task 1:
Create a user with your first name as username, 1234 as password and firstname@gmail.com as email, then proceed to login.
Scenario 2:
In the invitation you also received a party ID: 1.
Task 2:
Use the party ID to add the party
Scenario 3:
You have some music you would like to hear at the party.
Task 3:
Add "Ben Howard - Only Love" to the party's playlist.
Scenario 4:
You would like to vote on some of the songs on the party's playlist that you like. Before you can vote, you must be checked in to the party.
Task 4:
Vote for "Queen - Bohemian Rhapsody" on the party's playlist.
Scenario 5:
You heard a song at the party that you want to add to your favorites, so you easily can add it to another party some other time.
Task 5:
Add a previously played song to your favorites.

6.2.2 Design Changes Since the Last Test

The way the navigation between the screens of the prototype works have changed a bit since the Powerpoint-prototype in Section 6.1.2. The biggest change is the way the party list is shown. Instead of being one list with headlines, it is now three tabs, as seen by Figure 6.3.
The layouts of the screens are almost the same, however we have added some more consistency. The buttons to add favorites are all stars, and are now always in the same side. The green arrow and "Add" button have been changed to a plus sign, and all these are placed in same side as well.

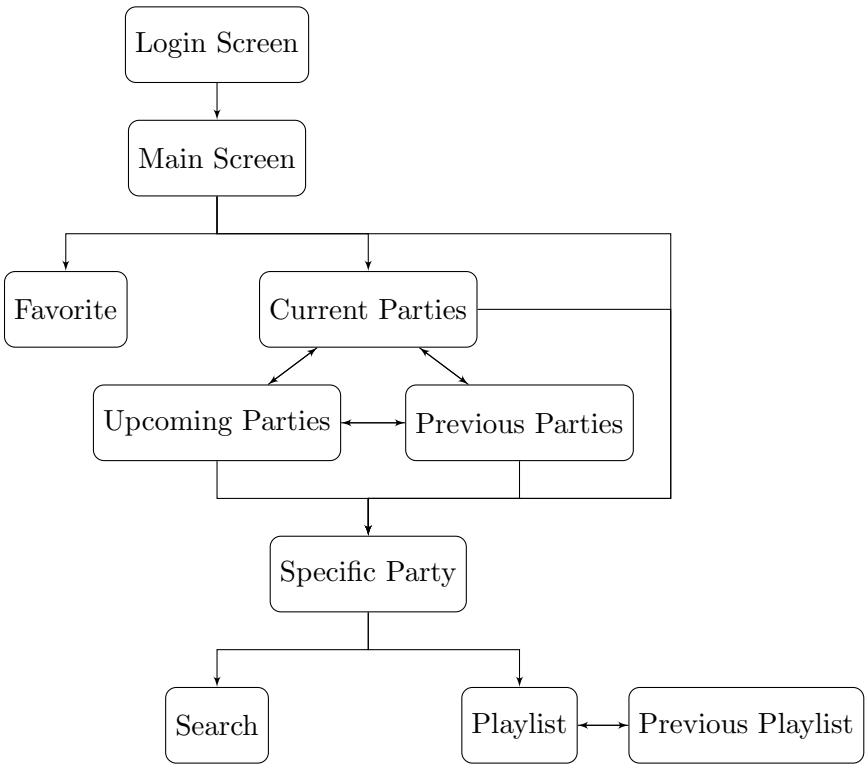


Figure 6.3: Navigation map of the Android application

6.2.3 Application V.1 Prototype Test: Ida

Before the test, she had time to read and understand the different cases and scenarios that we made beforehand 6.2.1.

When the test started she had no problem with task 1, creating a user, apart from the application crashing. After creating a user she had no trouble logging into the application. She had a little trouble figuring out how to do task 2, adding a new party to the list of parties, because the button for adding wasn't obvious enough. There was no problem in figuring out how to do task 3, searching and adding a song to the party playlist. For task 4 she had some trouble figuring out why the icon for adding and voting up were the same, but after a little explanation she figured out how to vote up. For task 5 adding a song to favorites was done without any trouble.

After the test we had a short interview with follow-up questions.

She thought that creating a user was an easy process because you only had to press the "create user" button and the rest made sense. She thought that adding a new party to the list of parties was a bit confusing because all the buttons were the same color, she wanted it to be more obvious which button to press. She thought it was easy to search and add a song to the playlist, but it was confusing that the button for adding and voting up was the same. She kept saying that she wanted a more colorful design and warmer/clearer colors, to make it more appealing and obvious which buttons to press. The reasoning behind this was that if you're intoxicated at a party and want to vote up a song, it can be difficult to navigate through the application if it's just black and gray.

The conclusion of this test is that we have to change the design into something more colorful or at least make it more obvious which buttons to press at a given time. It also shows that the process of creating a user, adding a party, searching and voting for songs in practice works as intended.

6.2.4 Application V.1 Prototype Test: Rikke

Rikke was instructed to follow the scenarios and tasks associated with the testing of this application. From the login screen Rikke, has no problem at all knowing that she need to click "new user" as a part of solving task 1. Rikke is satisfied with the way to create a user, she get a little annoyed when she has to type in her email twice, however she likes the idea of only using a username, password and email to make an account. Clicking create user confused Rikke because she got back to the login screen, and then needed to login. This confusion gave a critical error, because Rikke just clicked login instead of typing in her information again. Solving task 2 required little effort. She understood that to add a party, she needed to click the button saying "my parties" and then click "Add Party". She also noticed the option to view upcoming - and previous parties just by changing tabs. Visiting a specific party did not lead to any problems at all, nor did understanding the functions provided by the specific party screen. Solving task 3 was not a problem at all. Rikke saw the search function immediately, when she was searching, she asked for a function which shows search suggestions dynamically. After searching there is a little confusion about why the keyboard was still up. Understanding the icons was really easy for Rikke. She noticed the popup message saying voted, and understood that she had now voted. Task 4 was solved in no time, however she asked about a search function like the one she just saw when solving task 3. When solving task 5, she was already in the playlist and saw the previous tab within a few seconds, and decided to add a Coldplay song.

During the test Rikke only had one critical error, which was logging in after creating a user. She also struggled a little when moving back from a screen.

6.2.5 Conclusion

Add song, search, add party, favorite, create user worked as intended

The overall conclusion on the second prototype is that we had to be much more aware of the design of the application than we were expecting. The color and size of buttons seemed to matter much in how you understand and navigate in the application, and the informants didn't always think it was obvious what to do at a given screen in the application. In both tests there were some problems logging in to the application which we had to do something about.

For the sake of consistency we discovered that a search function in the playlist, as the one in the library, was a wanted feature, also that when you press down a button that it lights up so that you know you've pressed it down. The message you get when you vote or favorite a song didn't get noticed, and should be changed

Lastly the informants had some problem with the navigation when they had to go back from the screen they were currently on, so more buttons for navigation was a wanted feature.

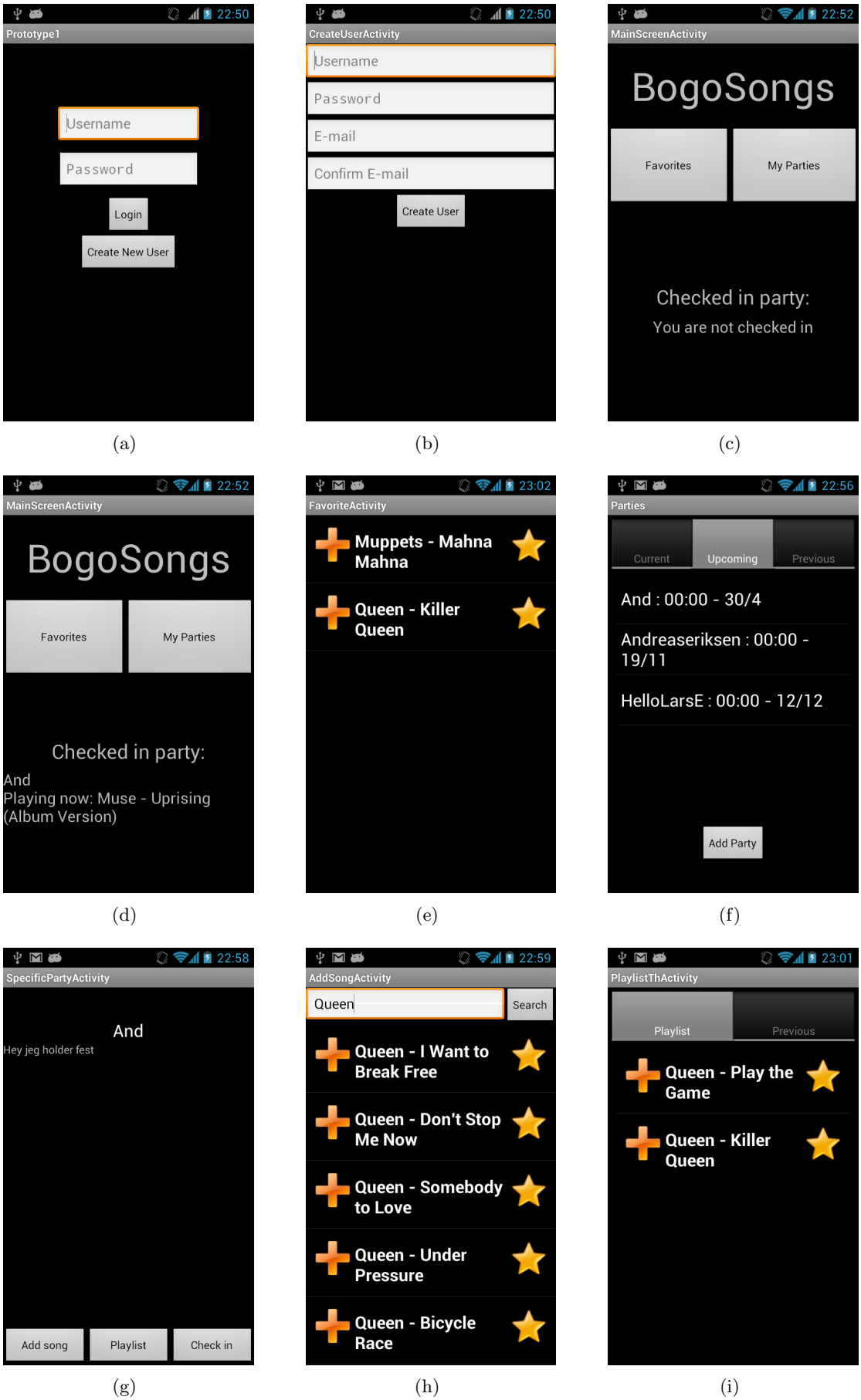


Figure 6.4: Screen from the Android prototype 1

6.3 The Second Android Prototype

6.3.1 Purpose of the test

In the second application test we wanted to test our website, as well as the new navigation and design in the mobile application. We made some scenarios and tasks for the test of the web service.

Scenarios and Tasks

- Scenario 1: You want to use our website for a party you're having
- Task 1: Create a user and log in on the website
- Scenario 2: You've created a user and want to create a party
- Task 2: Create a party on the website
- Scenario 3: You've set in the wrong date for the party
- Task 3: Change the date of the party

The scenarios for the application were the same as in the last test.

6.3.2 Design Changes Since the Last Test

Since the last test we changed the design to be more bright to make it more obvious which buttons to press at a given time. We've changed the buttons to be with icons and text. Furthermore we've added some navigation tools at the top of the application, and when you create a user you log in to the application, instead of having to write in your credentials again.

6.3.3 Application V.2 Prototype Test: Ida

The test of the website started out with Ida making a new user for task 1, there was no problem with this. For task 2 she had to log in and create a new party on the website. There was no problem involving creating a party, but for task 3 where Ida had to change some settings of the party, it wasn't obvious how to get into the setting menu.

Afterwards we tested the new application design and navigation. Ida said it was much better than the precious design, because it was more light and clear which buttons you were to press. She had to log in with the user she created on the website, and test the navigation by going to the party she created and adding some songs.

After she had tested the application we had prepared some questions about some things that we, ourselves, couldn't agree on, so we wanted to ask the informants on what should be done, and take their opinion into consideration. She wanted to be able to create parties from the application. She thought that it was weird that you could only do it from the web service. She wanted the whole interface to follow our color design, instead of some tabs being gray. She didn't like the new back button we had implemented, because it did exactly what the back button on the phone itself did. Furthermore she agreed that it was best with both icons and text for the buttons. As the final question we asked her about if she wanted an arrow or a plus for voting upon which she answered that arrow up was preferred, but it didn't matter as long as we changed the color of the icon to green to symbolize giving a positive value to an object.

The rest of the test wasn't organized, and it just went on as a discussion between us and Ida.

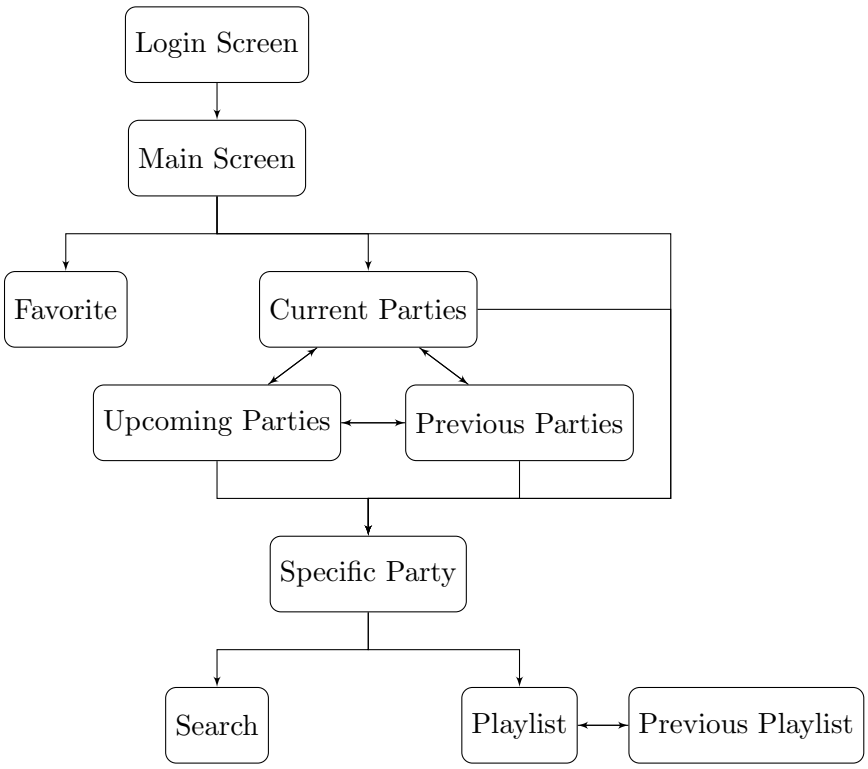


Figure 6.5: Navigation map of the Android application

6.3.4 Application V.2 Prototype Test: Rikke

We started by testing the application on Rikke by using the same tasks as the previous test. In the first task Rikke created a new user and afterwards she logged in with the user. She was a little annoyed by the fact that she had to type in the email in twice (still). In task 2 she easily navigated to the parties page and added the party to the upcoming parties list. Rikke handled task 3, 4 and 5 without further ado.

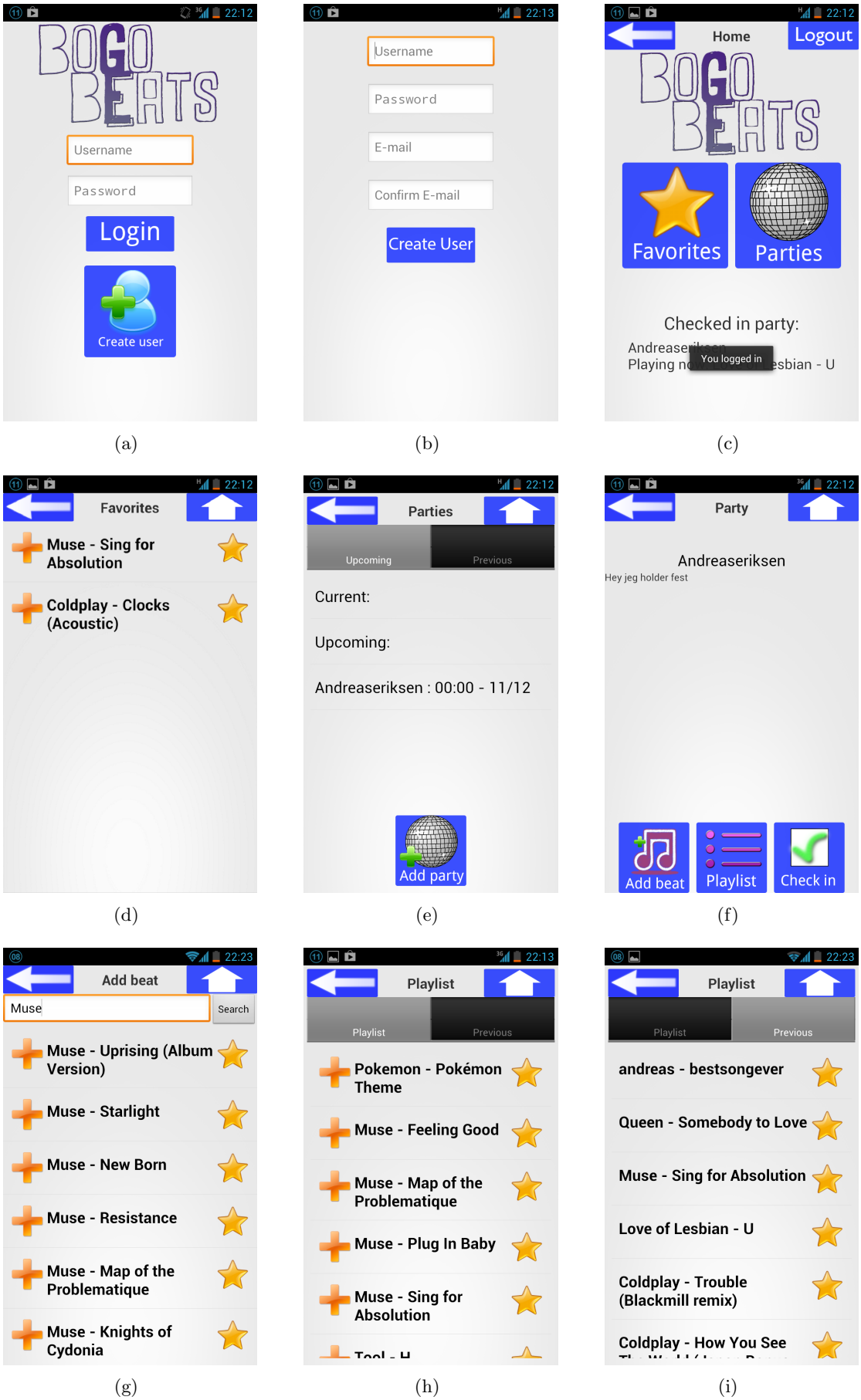
After the test we asked Rikke the same questions as we did with Ida, questions about the applications design. We started by asking her about the checked in text on the home page, Rikke hadn't noticed that it was clickable. She preferred it to be more inviting to click on. When we asked her about the gray tab design she thought it was fine, but the text wasn't visible enough. Rikke liked both the back buttons and the home buttons design and functionality. We then agreed that the bigger buttons should both have text and icons. As the final question we asked her about the buttons used for voting. She didn't like the plus icon and rather preferred an arrow up button in the same color.

After talking about the application, we moved on to the website. Rikke started out by logging in. She easily created a party due to the fact that she found the layout very simple. When we asked her to change the date on the party she just created, she had some trouble navigating to the edit form. On second attempt she navigated to the edit form and then changed the date of the party without problem. At first when she had just tested the application, she didn't find the idea of managing parties on the website good. She rather wanted it to be done on the application, but after testing the website she liked the idea of it being separated.

6.3.5 Conclusion

Both our informants were happy about the new design, and they agreed that there should be icons with text for all the buttons in the mobile application. They both agreed that

there should be an arrow up button in the playlist for voting up, because they thought it made more sense. They agreed that the checked-in feature in the home screen should be more like a button, because they did not see it as a thing you could actually click on. For the website they thought that it should be more obvious how to edit the information on a party.



(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

Figure 6.6: Screen from the Android prototype 2

6.4 System Test

In this section we describe the system tests we have during for our system. A complete overview of the test, the test parameters and the result can be found in Section 6.4.10 at the end of this section.

6.4.1 User Login/Creation

We wanted to make sure users cannot create and log in with users that would create errors in the system. For this we tested the following:

- Everything, except password, should be lowercased
- Can't insert any null values
- Email must contain a "@" and a "." (example**@**domain**.**com)
- Email and confirmation email should be identical
- Usernames and emails should be unique
- Should be able to handle special characters

These first 5 system tests were quite trivial, as we could simply test by creating users with capital letters in the name. For the first test, the database automatically reads everything as lowercase. It's also not possible to enter any null values in the textboxes. We also make sure that the email is of the pattern example@domain.com by using an email validater from the Android API. In the database, we defined username and email with the UNIQUE keyword, so we make sure two users can't have the same username nor email.

For the test with special characters, we had to use *percent-encoding* [3], as the application calls URLs, where certain special characters might have special purposes. We need to be able to handle special characters for security reasons, as users could run SQL queries on our database if the URL could contain characters such as " (quotation mark). By using percent-encoding we can format any character that might be harmful or cause problem into a non-harmful character. The database can then read the percent-encoding, and transform the character back to it's original value. We do not allow |to be used, because we use character when dividing inputs in the URL, which is why it is being seeing as "error" in Appendix B.1.

6.4.2 Adding a Song & Voting

Following are the things we wanted to make sure worked:

- Song is always added with 1 vote from the adder
- Songs can be added when not on the playlist
- All search results show up
- If song is already on playlist, the song will receive another vote
- All votes will increment the vote count by 1 per unique votee
- Does Tinysong support special characters properly?

99 songs were added by searching with: “a”, “b”, “c”, “d”, “e”, “f”, “g” and adding all song results. All results given by Tinysong’s search was shown, and all songs added by this were also added to the playlist with 1 vote. After this another userID was used to do the same and add the same songs. All these songs ended with 2 votes. An unforeseen problem arose with this test; certain characters cannot be interpreted with the search implementation, for example songs with Chinese letters would crash the application. It also turned out that songs with quotation mark would break the JSON code, as it would end the string at the quotation mark. We added a ”\”, to escape quotation marks in Java. Tinysong only works with alphanumerical characters. We can conclude that the functionality of adding a song works correct for all parameters.

6.4.3 ListView Test

To make sure our system can withstand large amount of data being processed, we performed a quantitative test on our ListView widget in the Android application.

The test we did here was:

- Adding a large amount of songs to a ListView in the application

Whenever we’re showing songs in the application, we use a ListView (Android GUI element). We wanted to be sure these ListViews could handle a realistic number of items, so we chose to perform a test to see whether the Favorite list in our application could handle 2000 songs. For testing we added 2000 songs to a list of favorites. The ListView could handle all 2000 songs. From the Android documentation, we can see that there’s no limit on the ListView widget, and it’s only limited by the smartphone RAM (Random-Access-Memory) [11].

6.4.4 Playlist/Previous Playlist

To make sure our playlist and previous playlist are functioning as intended we have tested the following:

- Playlist must be sorted by votes descending
- Can vote songs up from playlist
- Previous playlist is sorted by played time
- Can vote for songs that have already been played
- Songs must be able to be added to favorites from both lists

The results from these tests showed that the playlist was sorted in the correct way, by votes descending and that voting is possible from the playlist. The playlist doesn’t get updated dynamically, so if a song gets more votes than another while the playlist is opened, it can only be seen when loading the playlist again, and does also apply to the previous playlist. This is because we get the playlist from the server, and we only get the playlist when opening the playlist the first time. From both playlists, favorites could also be added. Songs could also be voted for again after they had been added to the previous playlist, without being duplicated on the previous playlist. Songs on the previous playlist were not sorted by when they had been played.

6.4.5 Grooveshark Playing

We have also tested our interaction with Grooveshark, this was performed using the following parameters:

- There will always be music playing
- All songs will be added from playlist (and played)
- There will never be more than one song in Grooveshark queue

All songs were successfully added and played from the playlist during the testing, and there was never more than one song in the Grooveshark queue. The music did, however, stop after the playlist had been played, without the system adding songs from the recommendation list.

6.4.6 Favorites

To make sure our favorite function is functioning as intended we have tested the following:

- Any song can be favorited
- Any song can be unfavorited, if favorited
- Favorite list is correct, all favorites shown

The test was done by favoriting songs with special characters, and then checking in the application and in the database. The test for unfavorite was tested by unfavoriting in the application and then checking the database if the favorite was gone. For checking if the list was correct, we checked if all the songs that were favorited, were in the application and in the database. We can conclude that all these tests passed.

6.4.7 Add Party

To make sure our add party function is functioning as intended, we tested these following parameters:

- Any party ID can be added if it exists
- Party IDs are unique
- Non-existing party IDs cannot be added

By adding all the known party IDs to a users list, we tested that any party can be added if it exists. To test if each party has a unique party ID, we simply created new parties and checked if any of the IDs were used. By trying to add party IDs that did not exist, we tested if non-existing party IDs could be added. After these tests, we can conclude that all functions regarding adding parties work as intended.

6.4.8 Party Lists

To make sure our list party function is functioning we have tested the following:

- Parties are put in the correct list depending on the date
- All parties added are shown

We tested if they were input correctly by adding parties to the parties list and checking each date. This worked as intended, however they were not sorted correctly by date. To test if all the parties were shown, we simply added a lot of parties to the list and checked if they were there, which worked as intended.

	S1	S2	S3	S4	S5	S6
Party A/1355	10	5	7	0	3	0
Party B/1356	9	4	6	4	0	0
Party C/1357	0	0	2	2	7	10
Party D/1358	3	1	0	0	5	0

Table 6.1: Recommendation test

6.4.9 Recommendations

We tested the following:

- Theoretical similarity, same as actual calculated values
- Can vote, add and add to favorites from suggestions

We set up an example, similar to that of Section 3.4:
The similarities we got from party D are as following:

Party A/1355: 0.624756
Party B/1357: 0.455083
Party C/1356: 0.429273

As can be seen, the results are exactly the same as the example of Section 3.4, and we can conclude that our server calculates the right similarities. Adding, voting and adding to favorites also worked with the Android application.

6.4.10 Table of Results

Test	Input	Result	Appendix
Everything in lowercase	Usernames with uppercase letters	Passed	None
No null fields	Null input in textfields	Passed	None
Valid email	Invalid email, valid email	Passed	None
Unique usernames/emails	Create users with duplicate names	Passed	None
Handle special characters	Special characters	Passed	Appendix B.1
Voting	Votes	Passed	None
Adding songs to playlists	Songs	Passed	None
All search results show up	”a”,”b”,”c”,”d”,”e”,”f”,”g”	Passed	None
If songs is on playlist, add vote	Adding songs on the playlist	Passed	None
Tinysong searching w	Special characters	Passed	None
ListView limit	2000 songs	Passed	Appendix B.2
Sorting playlists	Playlists	Failed	Appendix B.3
Adding song from favorite list	Songs from favorite list	Passed	Appendix B.3
Music always plays	Play until playlist is empty	Failed	None
Whole playlist played	Play a playlist on Grooveshark	Passed	None
Maximum 1 song in queue	Playlist with more than 1 song	Passed	None
Favorite any song	Songs with special characters	Passed	None
Unfavorite any song	Songs with special characters	Passed	None
Favorite list is correct	Favorite list	Passed	None
Adding parties	Existing and non-existing party IDs	Passed	None
Parties are in the right list	Upcoming/Current parties	Passed	None
All parties shown	Parties for a user	Passed	None
Similarity is theoretical correct	Example from Section 3.4	Passed	None
Adding recommended songs	Recommended songs	Passed	None

Table 6.2: Table of system test results

Chapter 7

Conclusion

7.1 Results

The research question we worked on answering in this project was as follows:

> *How can we develop a system which can help people at private parties listen to the music that they want to hear, without interruptions?*

Using the an iterative process in this project, we received help from two informants to help us develop a system. The help we received from the informants was primarily in the form of prototype testing, which we then evaluated to create a system that could be used by people from our target group. We implemented this system as an Android application, while using a database to store data on and a website to administrate party-specific information. We did a full system test on the implemented system, to get rid of bugs and errors and to ensure the quality of the system.

We conclude that we have made a system that is usable by our target group, which makes administrating music at parties easy, letting the majority control the music using votes. Using a recommendation system, users can find music they might like. The system uses an external music service for playing and storing the music. The music does, however, stop if the playlist is empty, and therefore we haven't complied with all of our hard constraints from Section 2.2.

7.2 Discussion

In Section 2.2 we had hard and soft constraints that we wanted out system to comply with. The hard constrains had to be complied with to be accepted as a successful system by us, and we wanted to comply with as many of the soft constraints to ensure that our system had good quality. As seen by Section 7.1, we comply with all but one of the hard constrains. We will now see if we comply with all the soft constraints. The following soft constraints are being complied with:

User uniqueness functionality
: Each user get their own username and password, which they can use on the website and in the application. Because we have this, the next soft constraint was easy to comply with.

A favorite list
: Each song on Grooveshark have a unique song ID, which we can store for each user if they favorite it. Then we can simply output these songs by their ID in the application.

A list of previously played songs
We store the songs that has been played at each party, so that we easily can find and output these songs again in the application. Having this, made it possible to recommend songs to other parties, since we could compare two parties, and recommend songs from similar parties.

Party check-in
It's possible to check in to parties.

These next soft constraints are not fully complied with:

Connection with social media
There's is no connection with social media in our service. We looked into using Facebook login to log in to our system, but we never implemented it.

Administrator permissions
We wanted to let the host administer the music at the party he's hosting. In our implementation, the only way to administer the music is using the button on our Grooveshark overlay, which everyone at the party can use - Which means that the host doesn't get any special permissions.

Platform independent
Our system can only be used by Android phones as of now. However, since we have most of our functions on the database, it would be easy to implement our system to another technical platform. If we have had more time, our implementation would be quite platform independent. If we had made a web application instead of a native application, our system might have been fully platform independent.

Option to have maximum songs on the playlist
We didn't implement this feature, because we didn't felt like it was important enough to use our time on.

While we only comply with half of the soft constraints we put up, we don't feel that our system is in a poor quality. Connecting to a social media and having a platform independent doesn't affect our system in a way that makes it any less good. These constraints are only to gain a larger user-base and letting these users enter our system faster, and doesn't affect the system in the same way as a missing feature like administrator permissions, which could lead to unsatisfied users. Because of our voting system, we don't feel like the option to define a maximum songs on the playlist is very bad for our system, as the popular songs will always be played first.

Good design is, as recommending, not something that can be defined. The way we designed our system (especially our application) is by evaluating on our informants. From the last prototype test, our informants said our design was good, and only wanted few things changed. However, this is only two persons preferences (as well as our own), which doesn't necessarily mean that we have made a good design. We agree that there's still some areas that could use some better design, such as making some buttons look more clickable, and especially designing our Grooveshark overlay properly. We are, however, overall happy with our design.

One of the requirements for this project was to use informants to help develop a good system. We used informants primarily to test prototypes, and hear what they thought about the system. We believe that we have taken everything our informants have said into consideration and evaluated on what they have told us, which we have used to make

a system that our informants would be happy to use. This is especially noticeable in the design and the way we navigate in the system, where we had made major redesigns after evaluating on the responses from out informants.

Our informants seem happy with the system, and we believe that they would use it, but what about others people? From observations during the project and the responses we've received from people we've talked with, the system is something that a decent amount of people would use. However, this is of course before they have used it, where we have to admit our system might need some improvements. Our two biggest concerns are the fact that it's only for Android, and the dependency of Grooveshark (including the overlay).

Our system would need to be implemented on at least iOS before it could be used properly at parties, as only allowing some people at parties vote, doesn't give good results.

Using Grooveshark can be a weakness as we have already seen during the development of our system. We had at one point a API StreamKey handed to us by Grooveshark, which we used to add songs from their top 50. This key was then later removed without any notice, which meant that our function to add songs from their top 50, had to be removed again. We have to use our overlay because we're not allowed to stream music from Grooveshark on our website, and since our overlay is so poorly designed, people might not want to use our system. The two Grooveshark problems we've faced could be solved by getting a full StreamKey from Grooveshark, by having a contract with them. With a full StreamKey, we could stream music on our website, and implement a proper layout, and we wouldn't have to use a boomarklet, which few people know what is.

7.3 Further Development

In our project we chose Grooveshark as our external music service. This meant that we had to get access to their API and while we did get access to most of it, there were still a few features which we could not use. The most important feature of their API that we were missing, was the music streaming. If we had access to the full API we wouldn't need to use a bookmarklet for playing music, as that would have made it possible to make a music player that could stream music from Grooveshark, and this player could then have been on our website and/or the Android application.

Another function we would like our application to have, is a kind of host control from the application itself. The host controls could be music related, like being able to skip, play and pause songs, which could act as a remote for a music player on a pc, or in the app itself.

Initially our system was going to support login functionality via Facebook, which would have made the system a lot easier to get started with, since Facebook is very popular. We also thought it would be a good idea to have Google account login functionality, again because it is popular, but also because we had decided to make an Android application, thus making a Google account an requirement to get our application in the first place.

We also considered social media functionality for our system, which could be made by using either a Facebook login or a Google account for Google+, thus making good use of a direct login to Facebook or Google from the login screen. The social media functionality could be used to create an event on Facebook or Google+, based on a created party in the system, and from this event it would be possible to invite friends from your Facebook or a Google+ account, thus making it possible to invite friends without them needing to have the application in the first place. In this event, information about our system and how to get started with it could be included, making it easier for people who have not

used our system before to start using it.

As mentioned earlier, making a platform independent system would attract a larger user-base, so that's definitely something that's worth doing. The service we've developed is only usable by Android users. While that's still the majority of smartphone users, developing our system to other smartphone operating systems like iOS and Windows would make sure almost everyone at parties are able to use our system, not just giving them the opportunity to vote, but also making sure that our system is better used, since more users equals better results.

Bibliography

[1] Android developers. http://developer.android.com/index.html.

[2] Bootstrap. Github Project. http://twitter.github.com/bootstrap/.

[3] Percent-encoding. Wikipedia Article. http://en.wikipedia.org/wiki/Percent-encoding.

[4] Apache license, version 2.0, January 2004. http://www.apache.org/licenses/LICENSE-2.0.html.

[5] Blackberry. Wikipedia Article, November 2012. http://en.wikipedia.org/wiki/BlackBerry.

[6] Cosine similarity. Wikipedia Article, November 2012. http://en.wikipedia.org/wiki/Cosine_similarity.

[7] Desktop operating system market share. Website, October 2012. http://www.netmarketshare.com/operating-system-market-share.aspx?qprid=8&qpcustomd=0.

[8] Iterative and incremental development. Wikipedia Article, November 2012. http://en.wikipedia.org/wiki/Iterative_and_incremental_development.

[9] djtxt. http://djtxt.me/.

[10] GO-Gulf. Smartphone users around the world - statistics and facts [infographic]. Website, January 2012. http://www.go-gulf.com/blog/smartphone.

[11] GoogleDevelopers. Google i/o 2010 - the world of listview. Youtube video, November 2012. http://www.youtube.com/watch?v=wDBM6wVEO70.

[12] Grooveshark. About us, December 2012. http://www.grooveshark.com/about.

[13] Sam Laird. Mobile site or mobile app: Which should you build first? [infographic]. Online Article, January 2012. http://mashable.com/2012/06/06/mobile-site-mobile-app-infographic/.

[14] Daniel Lemire & Anna Maclachlan. Slope one predictors for online rating-based collaborative filtering, February 7th 2005. http://lemire.me/fr/documents/publications/lemiremaclachlan_sdm05.pdf.

[15] J. A McCall, P. K. Richards, and G. F. Walters. Factors in software quality. Book Vol. 1, Department of Commerce, Springfield, November 1977.

[16] Andrew Orlowski. Grooveshark blocked in denmark by copyright warriors. Online Article, February 2012. http://www.theregister.co.uk/2012/02/23/grooveshark_blocked_in_denmark/.

[17] Spotify. Spotify premium. Website, November 2012. http://www.spotify.com/dk/get-spotify/premium/.

[18] TuneTug. Website. http://tunetug.com/about.html.

Part II

Academic Report

Project Description

The project we worked on this semester was the largest and most complex project we've worked on so far. We wanted to make a system that we would use ourselves, and came up with the idea of creating a social music service. This meant that we could work with almost any informants, since music is a part of everyone's lives, and that we could make a large system with several components, including several platforms.

This project have been quite a challenge for us, since we were aiming very high with the system we wanted to develop, especially considering that we had little knowledge on how to do most parts as we haven't had any courses in creating mobile applications, databases or websites. However, we feel that we've gained a lot of useful knowledge working on this project, and there's a lot from this project we can use in our later projects.

Making a Large System

In this semester we chose to make several components varying in size:

- Web page
- Android application
- MySQL database
- PHP back-end
- JavaScript music player

Having to deal with making all these components served to be quite a challenge. It gave some complications, such as coordinating our workforce. It proved difficult to populate the different areas of our components, and we ended up with a bad distribution. For example one group member only worked with the PHP back-end, resulting in a strong dependency of this members' knowledge and abilities, instead of a distributed workforce where a much needed second person would ease the dependency. Another challenge which we had to face, was to combine our components. An example of this was our early Android prototype where we had a 2-man-team working on coding the Android UI and making it ready for data from the database and another 2-man-team working on actually incorporating this data from the database. We ended up with different ideas of how this information was to be presented and other miscommunication, which resulted in a redoing of our work. A big challenge we faced was to work with an external API, this was a very interesting learning experience and forced us to research and talk with the provider, and in the end gave us a lot of functionality so we could skip this and work on the more important parts of our system.

While working with several components and having to learn several programming and scripting languages was quite a challenge for us, it was a great experience as we could develop a large system that we got to work on different platforms. We believe we can use the knowledge we've gained in this project when we have to work with for example databases in a later semester. But we should also realize that we might have been aiming too high with so many new areas to learn, and we should consider working on fewer components later on.

Dealing With Informants

In this project it was required for us to have at least two informants. The informants had to be someone that did not have a developer background. There was little or no consideration in the selection of the informants, since we used a friend and a sister as the informants.

Our iterative working process was perfect when combined with informants. For each iteration we created a prototype which the informants tested and we got some feedback to take into consideration. The course "Design and Evaluation of User Interfaces" taught us some methods how to do usability testing by using scenarios and tasks which we had asked the informants to do. This gave us a reliable and controllable test environment, which made it a lot easier to process the data we got. We first wanted to transcribe the interviews we had with our informants, but after analyzing the time spend and what we could use it for, we decided that it wasn't worth it.

Sadly we never got the time to try using the usability laboratory at Cassiopeia, which we believe would have been a great experience since we could have all seen how our informants, and other potential test subject, would use our system, instead of just the few people who were out interviewing the informants. If we had time to use the laboratory, we could also have learned how to evaluate using methods such as IDA (Instant Data Analysis).

The experience we got from working with informants is something we can take with us in the future. It's a great way of getting a whole new perspective on your project, and maybe giving you some ideas that otherwise would not have been considered.

Working Iteratively

In this semester we were working iteratively, which was a new experience for us, as we have been more used to working using a waterfall method. Working iteratively meant that our work process has had a huge focus on prototyping and then keep improving on the prototypes throughout the rest of the project. One of the first prototypes we made was a Powerpoint prototype. The Powerpoint prototype was used to test our ideas for the first time with our informants. Being able to test our system with our informants that early on, in our project, was a huge benefit, as we could figure out whether or not the informants liked our idea, according to both function and design, before we started coding. If we had not been working iteratively with prototypes, we would only had been able to tell our informants about our idea, but not have anything visually to show them. We think that having something visually to show the informants, helped them understand our system more, than if we only told them about it.

Working iteratively meant that shortly after we had shown our Powerpoint prototype we had to decided how our next iteration of our prototype should be, we decided that the next prototype should be an actual working application prototype. This lead us to start coding very early on, which also resulted in poor planning and more cluttered code, but also meant that we had more time to improve it.

We were, however, perhaps too quick to decide on our system, which is partly because of the iterative work method since we had to make prototypes rather quickly.

While the iterative work method is a great way of developing a system to the users' needs, it's a lot harder to plan for. As we were used to working with a waterfall method, we thought we could plan far ahead and make a proper time schedule. It turned out that we couldn't do this properly, as there were a lot of changes to be made all the time, that we couldn't account for.

We have different opinions on the iterative work method, as some prefer to working

with prototypes, always be changing the system, whereas some prefer to work using the waterfall method that is more easy to plan with, and overall more organized, but might result in a worse product.

Developing a System Using Activities

The system development course we had this semester taught us how to make a proper object oriented analysis and design when developing a system. Using the methods we learned in this course helped us to define our system in a new way, which is definitely something we can use in later projects.

Since it's the first time we were using these methods from the system development course, our system analysis wasn't as well made as it could have been, which is clear due to the difference between our system design and the implemented system.

Being the first time using these methods, we didn't really see how they would help develop a system, and we felt that we only did them because it was clear from the project guidelines that we should do the activities from the course, and we ended up doing them slavishly. This lead to a very poor quality of the activities, which rendered the results quite useless. It wasn't until later in the project we saw how they could be used properly, and had to redo all of our activities. By then we had already made most of our product, and we actually had to reverse develop our system, by making the activities fit the implementation, and not the other way around, which meant we never got to benefit from using these methods.

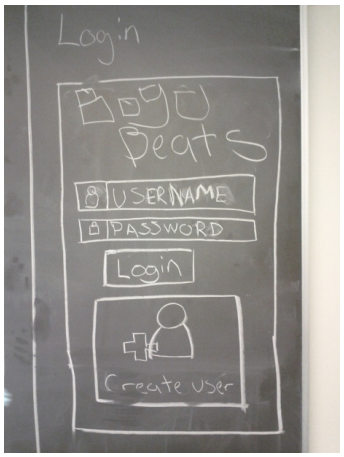
The course, however, couldn't keep up with the project either. We couldn't make a proper system analysis before we had to get a prototype ready, since we were using an iterative work method. If we have had the whole course in the first weeks of the projects, we could have finished a system analysis and could have designed our system properly before working on our prototypes.

Because we used an iterative process with prototypes, we had to revise our system design several times, since our system changed a lot during the prototype testing, which meant that we spend even more time rewriting our system development chapter in the development report, as well as editing the diagrams and figures. We therefore see the methods from the course better put to use, using a waterfall work method, as the system doesn't change that gradually, and we, the developers, get more time to make a full system design before we have to start working on prototypes.

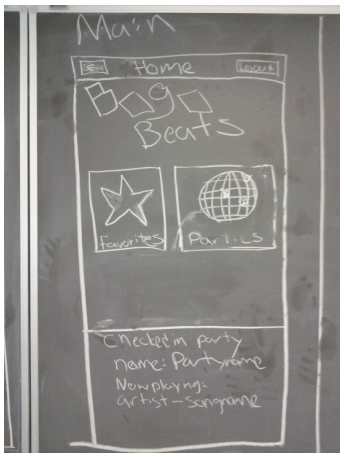
In the end we feel like we have learned a good method to design systems that works for both the iterative and waterfall work methods, but is best being put to use for a waterfall work method. We all agree that if we received such a system analysis and design, we could easily implement it effectively as programmers.

Appendix A

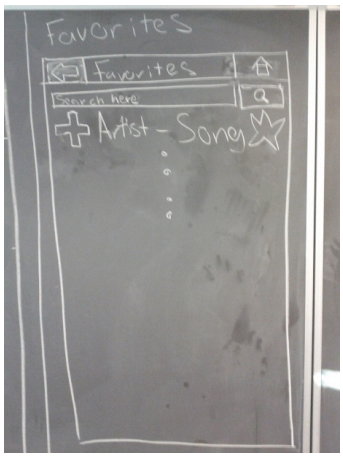
Design Workshop Pictures


Login
Boju Beats
USERNAME
PASSWORD
Login
Create user

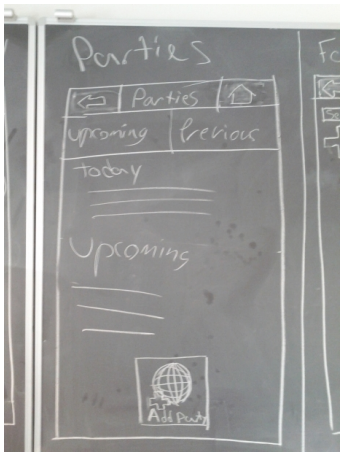
(a)


Main
Home
Boju Beats
Favorites
Parties
Checked in party
name: Partyname
Now playing:
artist – Songname

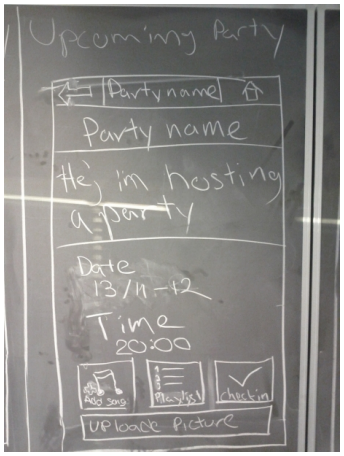
(b)


Favorites
Favorites
Search here
Artist - Song

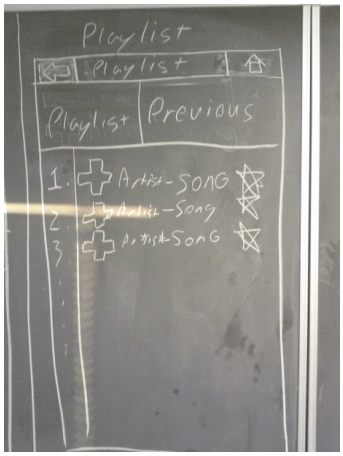
(c)


Parties
Parties
Upcoming
Previous
Today
Upcoming
Add Party

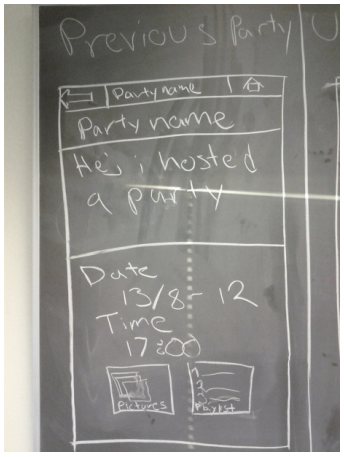
(d)


Upcoming Party
Partyname
Party name
Hej, im hosting a party
Date
13/11-12
Time
20:00
Add Song
Playlist
Check in
Upload Picture

(e)


Playlist
Playlist
Playlist
Previous
1. Artist - SONG
2. Artist - Song
3. Artist - SonG

(f)


Previous Party
Partyname
Party name
Hej, i hosted a party
Date
13/8-12
Time
17:00
Pictures
Playlist

(g)

Appendix B

Test Results

B.1 Special Characters Test

```
17 String[] specialChars = {"!", "\"", "#", "¤", "%", "&", "/", "(", ")", "=",
       "?", "'", "|", "@", "£", "$", "{", "[", "]", "}", "^", "~", "*", "<", ">
       ", "\\", ",", ";", ".", ":", " ", "–", "-", "‘", "á", "½", "§"};
```

Listing B.1: List of special chars

```
28 /** Creates users with special chars in the name */
29 public void CreateUserTest(View v)
30 {
31     for(String specialChar : specialChars)
32     {
33         output = new ServerConnecter().createUser(specialChar, "password", "
               email@email.com");
34         Log.d("Character: " + specialChar, "Result: " + output);
35     }
36 }
```

Listing B.2: Create users with special characters test

```
46 /** Logs in with the users with special chars as usernames */
47 public void LoginTest(View v)
48 {
49     for(String specialChar : specialChars)
50     {
51         output = new ServerConnecter().loginCheck(specialChar, "password");
52         Log.d("Character: " + specialChar, "Result ID: " + output);
53     }
54 }
```

Listing B.3: Log in with users with special characters test

Character	Result
!	success
”	success
#	success
¤	success
%	success
&	success
/	success
(	success
)	success
=	success
?	success
’	success
|	error
@	success
£	success
$	success
{	success
[	success
]	success
}	success
^	success
~	success
*	success
<	success
>	success
\	success
,	success
;	success
.	success
:	success
(space)	success
-	success
_	success
‘	success
á	success
½	success
§	success

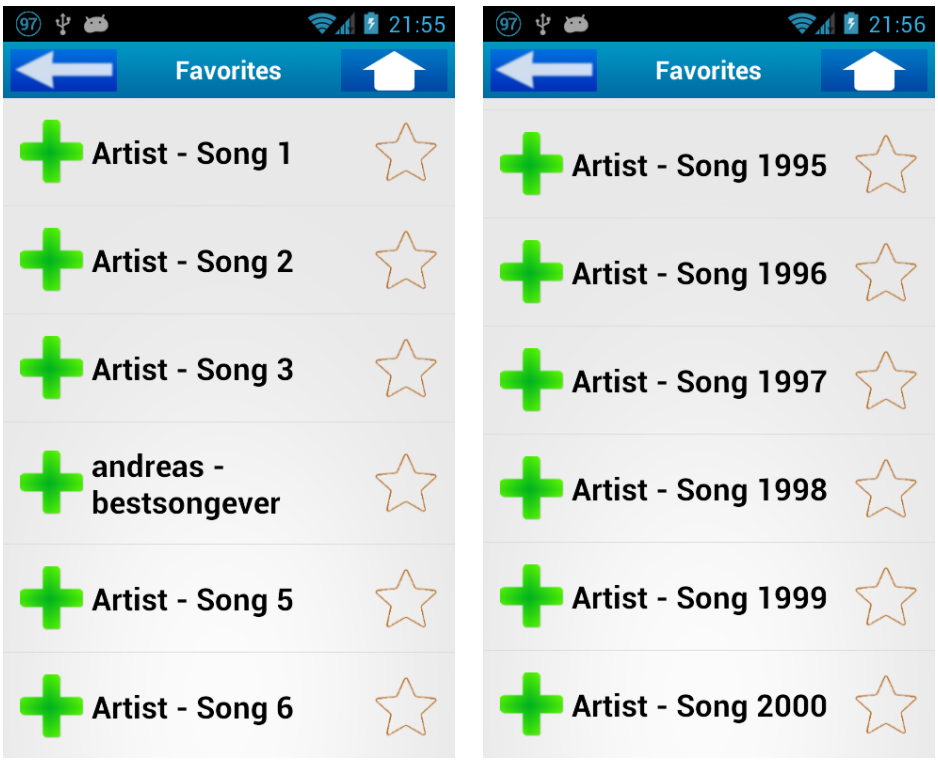
Table B.1: Test results of creating users with special characters

B.2 Favorite List Limit

```
77 /** Checks if the lists of songs can handle a large amount */
78 public void FavoriteLimitTest(View v)
79 {
80     testUserID = "2";
81     for(int i = 1; i <= 2000; i++)
82     {
83         output = new ServerConnecter().addFavorite(testUserID, String.
               valueOf(i), "Artist", "Song " + i);
84     }
```

```
85
86      testData = new ArrayList<Song>();
87      output = new ServerConnecter().getFavorites(testUserID);
88      if (!output.equals("error"))
89      {
90          output = new StringBuffer(output).deleteCharAt(output.length() - 2).
                toString();
91          Gson gson = new Gson();
92          Type collectionType = new TypeToken<ArrayList<Song>>(){}.getType();
93
94          ArrayList<Song> gsonsongs = gson.fromJson(output, collectionType);
95          Log.d("Number of favorites added:", String.valueOf(gsonsongs.size())
                );
96      }
97  }
```

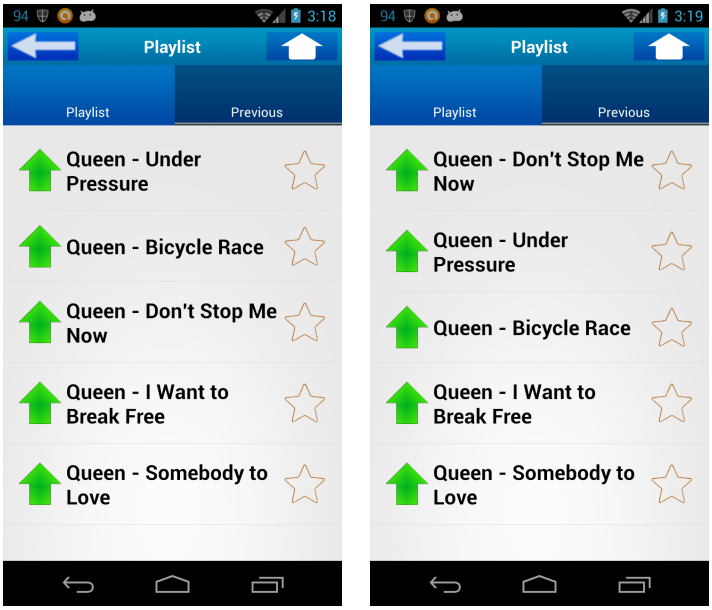
Listing B.4: Favorite limit test function



(h) Start of the ListView (i) End of the ListView

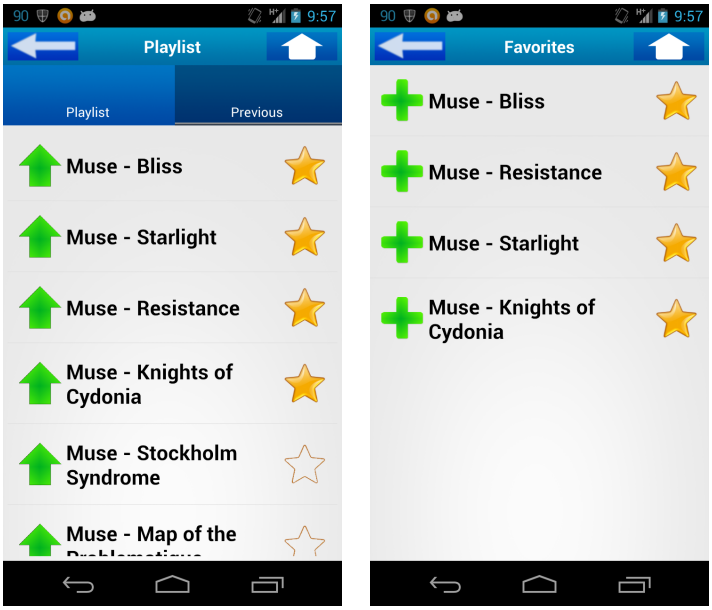
Figure B.1: Result of favorite limit test

B.3 Playlist/Previous Playlist



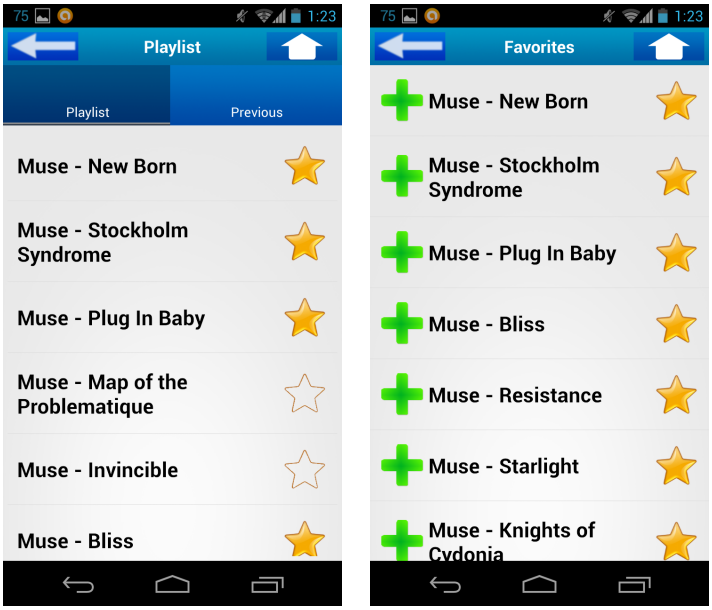
(a) Before vote from playlist (b) After vote from playlist

Figure B.2: Result of playlist vote and ordering test



(a) Adding favorite from playlist (b) After vote from playlist

Figure B.3: Result of adding favorite from playlist test



(a) Adding favorite from previous (b) After vote from playlist

Figure B.4: Result of adding favorite from previous playlist test

Appendix C

Screenshots

C.1 Website Screenshots

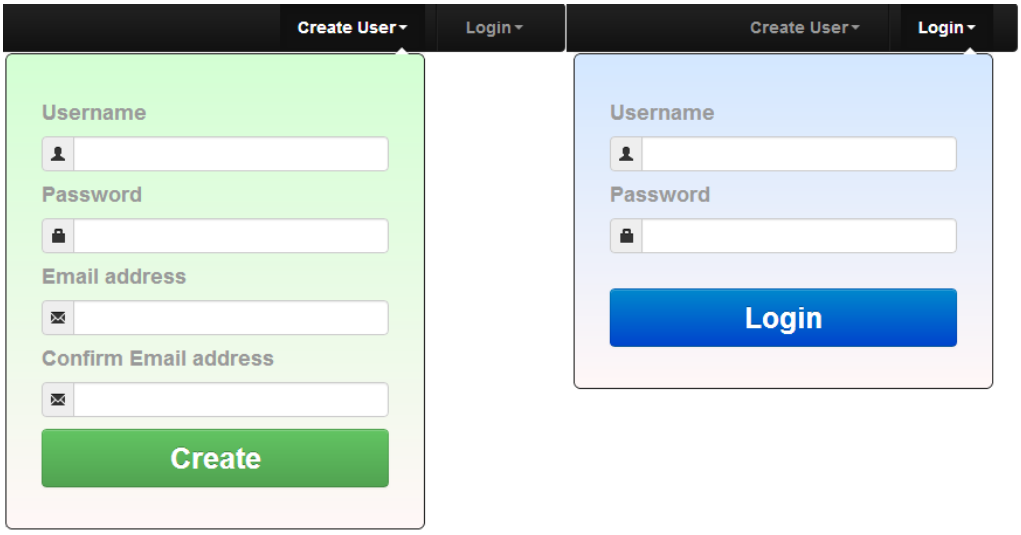


Figure C.1: Create user and log in

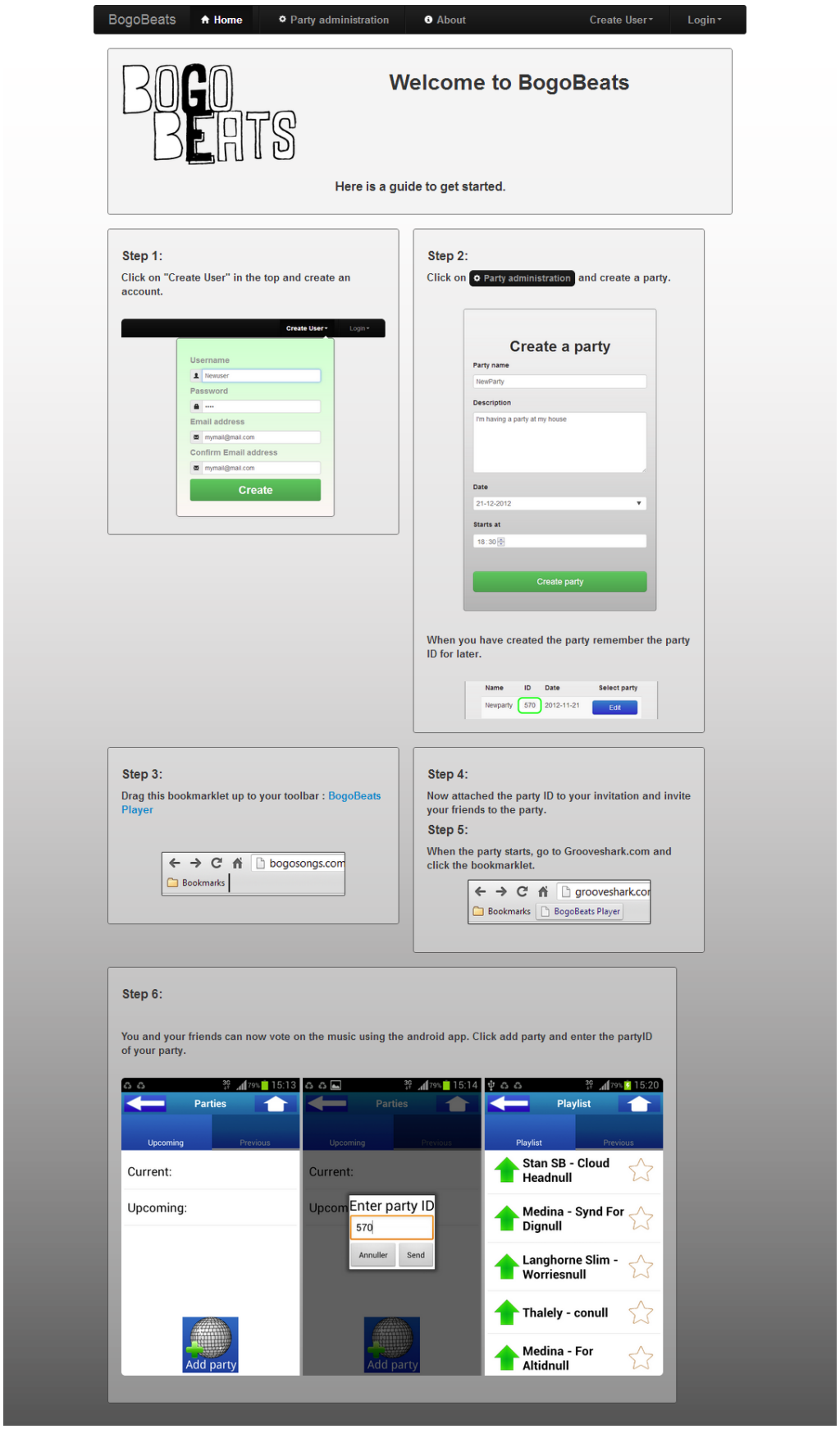


Figure C.2: Website frontpage. Home tab with guide

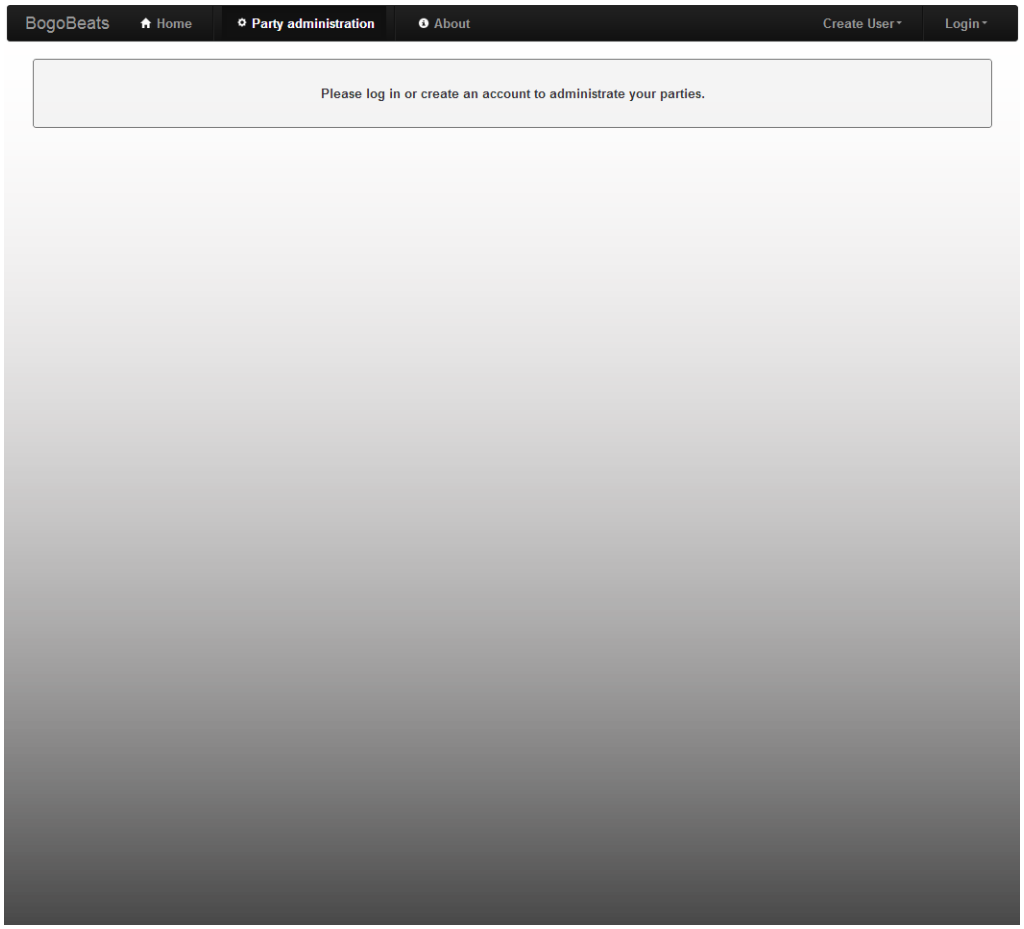


Figure C.3: Website party administration - Not logged in

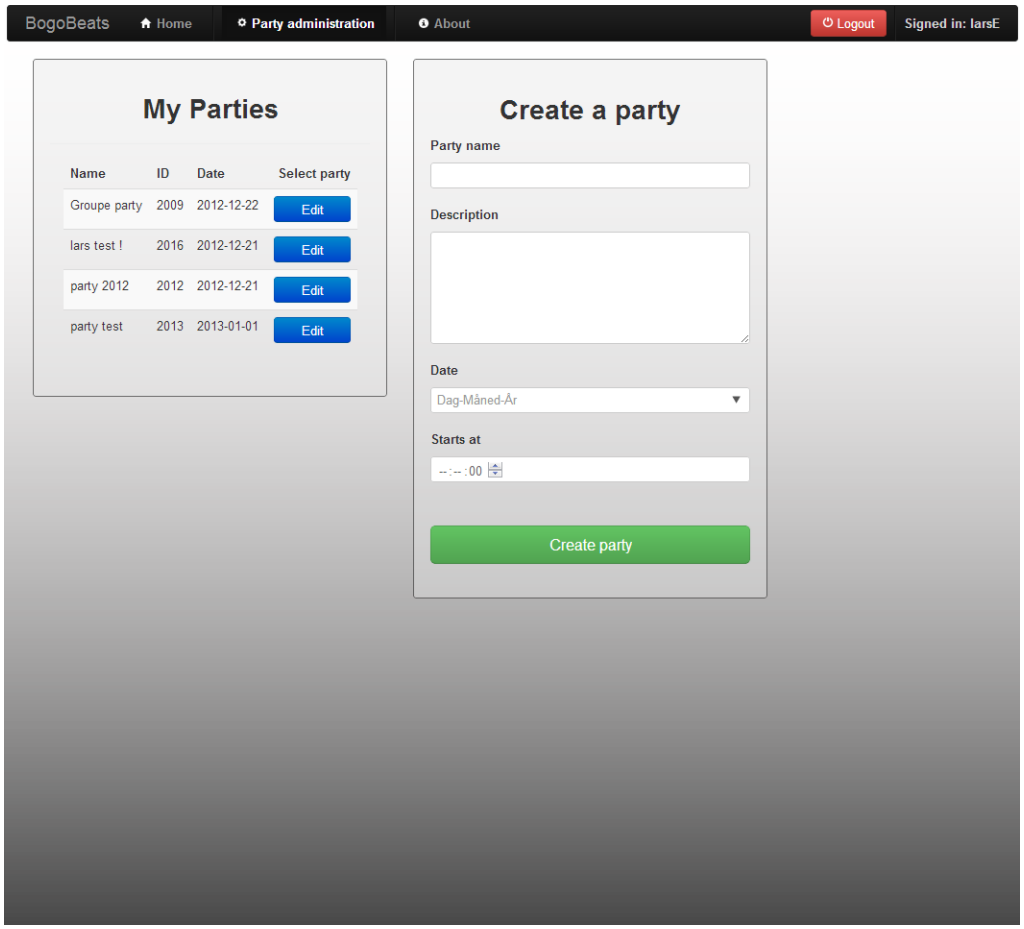


Figure C.4: Website party administration - Create party

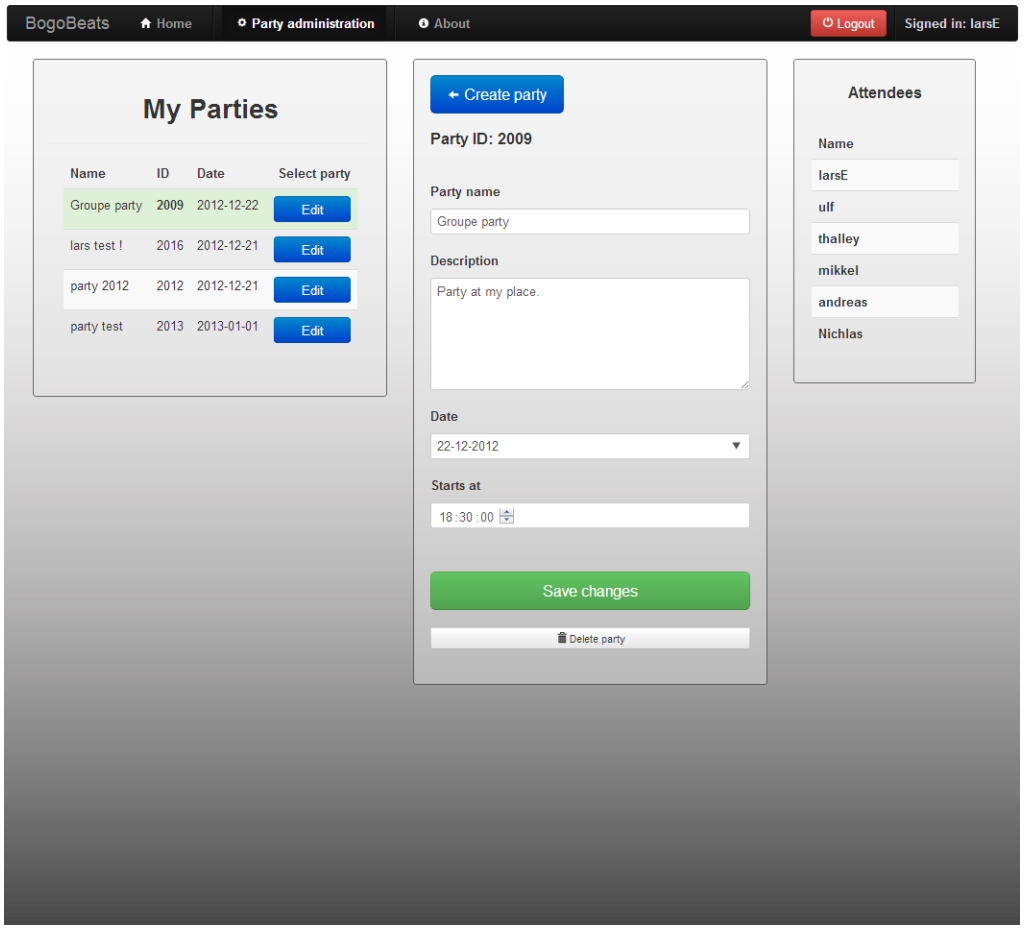


Figure C.5: Website party administration - Edit party

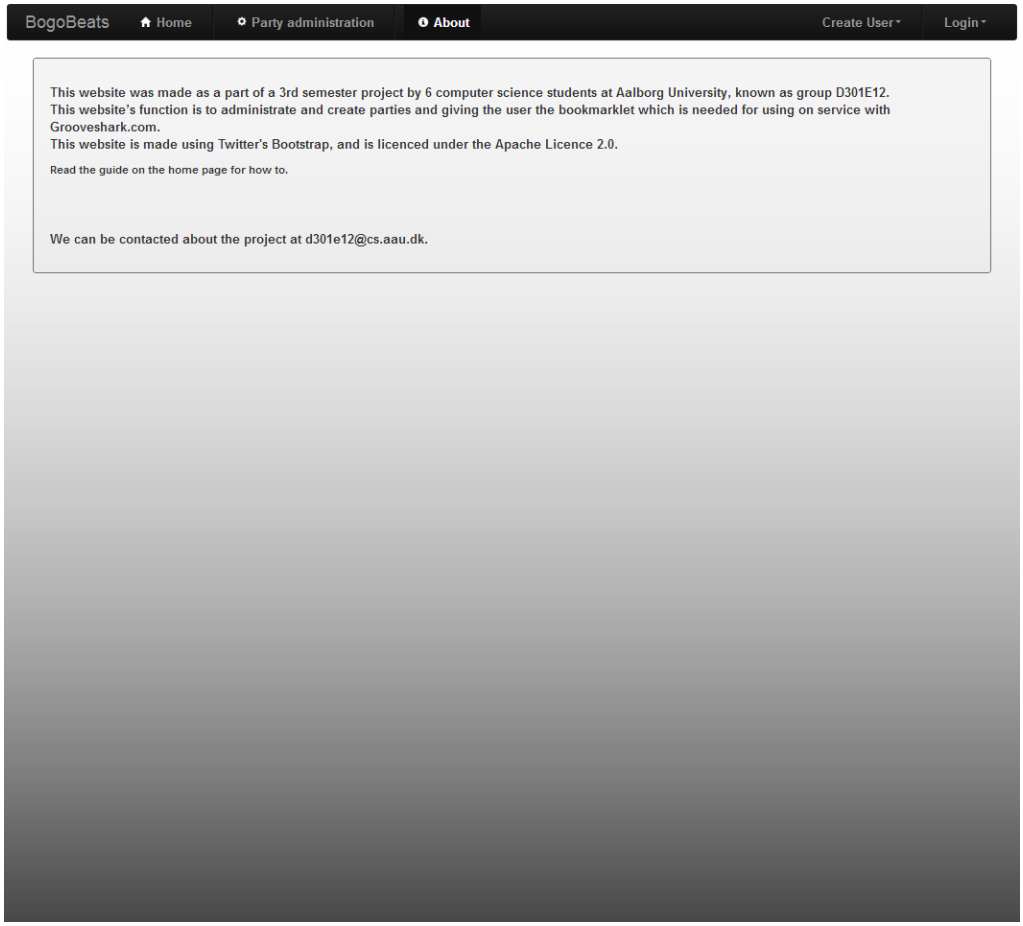
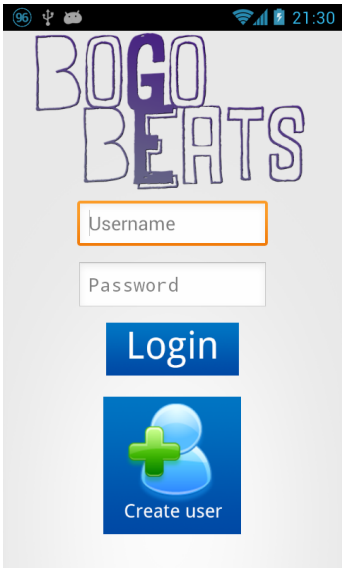
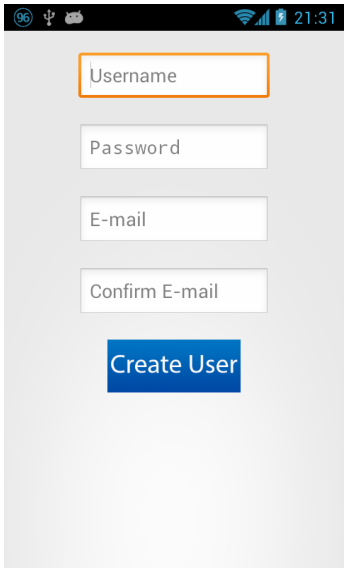


Figure C.6: Website about page

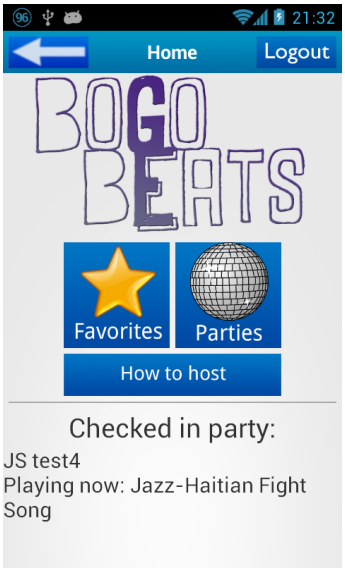
C.2 Application Screenshots



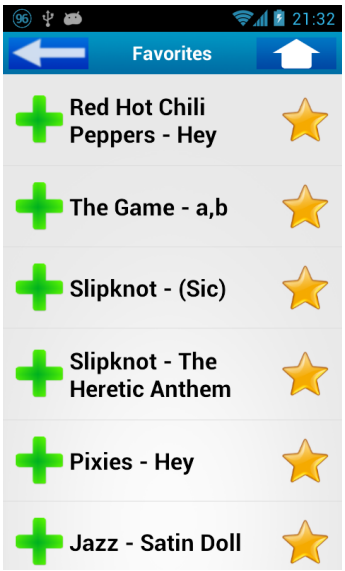
(a)



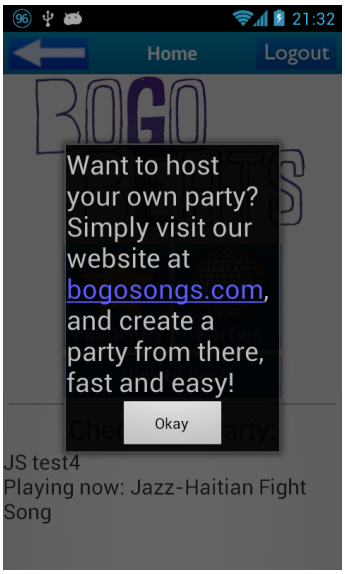
(b)



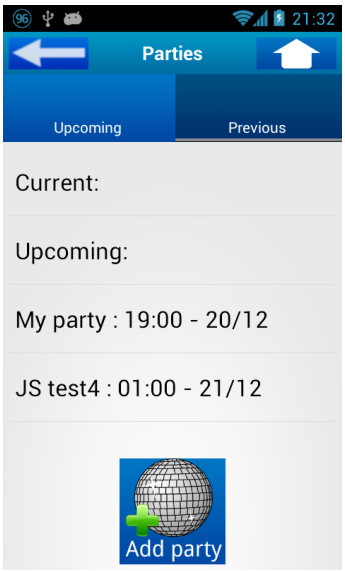
(c)



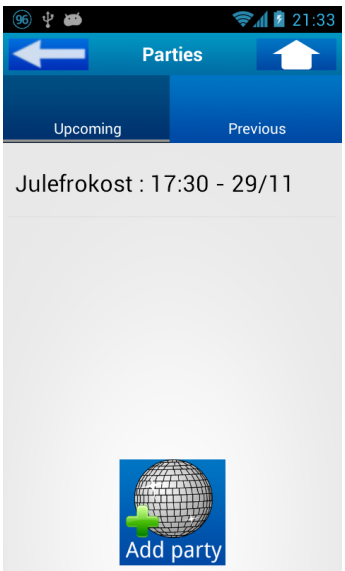
(d)



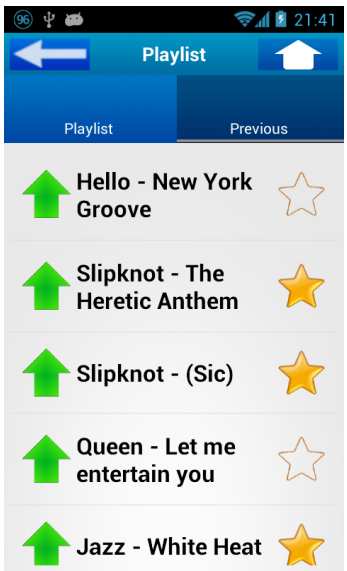
(e)



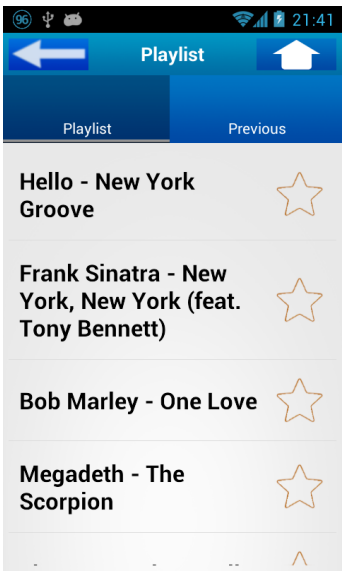
(f)


21:33
Parties
Upcoming
Previous
Julefrokost : 17:30 - 29/11
Add party

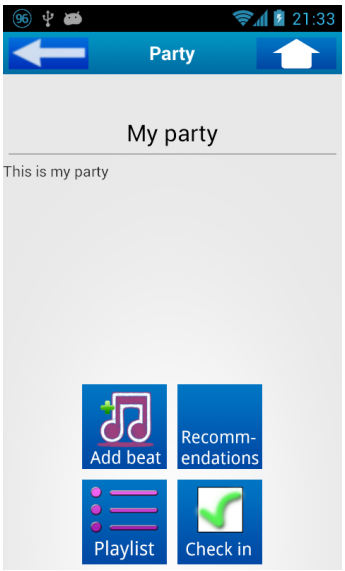
(g)


21:41
Playlist
Playlist
Previous
Hello - New York Groove
Slipknot - The Heretic Anthem
Slipknot - (Sic)
Queen - Let me entertain you
Jazz - White Heat

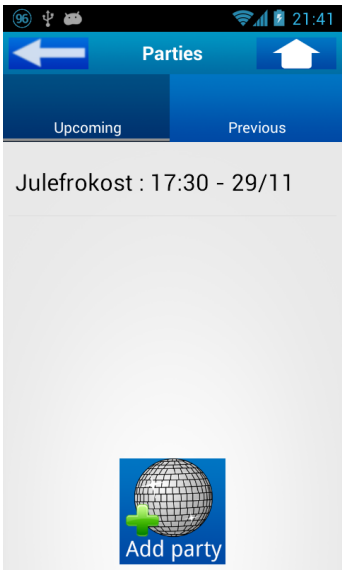
(h)


21:41
Playlist
Playlist
Previous
Hello - New York Groove
Frank Sinatra - New York, New York (feat. Tony Bennett)
Bob Marley - One Love
Megadeth - The Scorpion

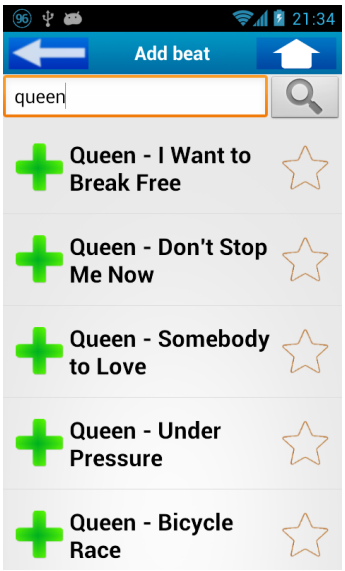
(i)


21:33
Party
My party
This is my party
Add beat
Recomm-endations
Playlist
Check in

(j)


21:41
Parties
Upcoming
Previous
Julefrokost : 17:30 - 29/11
Add party

(k)


21:34
Add beat
queen
Queen - I Want to Break Free
Queen - Don't Stop Me Now
Queen - Somebody to Love
Queen - Under Pressure
Queen - Bicycle Race

(l)