WASTELAND
- A GIRAF Database

P6 Project
Group SW603
Department of Computer Science
Aalborg University
June 4th 2013

Student Report
AALBORG UNIVERSITET

Institut for Computer Science
Selma Lagerlöfs Vej 300
9220 Aalborg Ø
Phone 99 40 99 40
Fax 99 40 97 98
http://www.cs.aau.dk/

Title:

WASTELAND - A GIRAF Database

Theme:

Developing Complex Software Systems

Project period:
6th semester 2013 SW

Project group:
SW603F13

Group members:
Barbara Flindt
Hilmar Laksá Magnussen
Jeppe Blicher Tarp
Simon Jensen

Counselor:
Katja Hose

Circulation: 6

Number of pages: 103

Number of Appendices: 3

Finished 4th of June 2013

Abstract:

We describe the design and implmentation of a central server application for the GIRAF system with a database, an API for communication and synchronization between the central database and a local counterpart on an Android device. This is done in the context of a multiproject consisting of 8 groups, all working on various aspects of the GIRAF system. We end up with a working implementation and describe future work and tips for future students working on top of this project.

Preface

The following report is the result of the SW6F13 project for 6th-semester students of Software Engineering at Aalborg University. The project is a subproject of the GIRAF project.
The report expects the reader to have a basic understanding of databases, client/server communication, C++ and MySQL, SQLite, Java and the agile project development method SCRUM.
The purpose of this project is to design a central server for the GIRAF project and implement synchronization between a local and central database.
The product of this project is this report and an implementation of a MySQL database, an API for database communication, a server application, local database on an Android device and synchronization between the two databases.
The group would like to thank Katja Hose for excellent supervision during the semester.

Contents

I Common Report **11**

1 The GIRAF Project **13**
1.1 Vision for GIRAF . . . 13
1.2 Previous Years . . . 13
1.3 Target Platform . . . 14
1.4 Autism . . . 15

2 The GIRAF Project 2013 **17**
2.1 The Goals for 2013 . . . 17
2.2 Definition of a Multi-project . . . 17
2.3 Group and Work Structure . . . 18
2.3.1 Development Method . . . 18
2.3.2 Development Tools . . . 19
2.4 Decision Making - The Process . . . 21
2.4.1 The Weekly Meeting . . . 21
2.4.2 Rules of Conduct . . . 21
2.4.3 Committees . . . 22

3 What was developed **23**
3.1 Pictograms, Morgana, and Design Guidelines . . . 23
3.1.1 Pictogram . . . 23
3.1.2 Morgana . . . 25
3.1.3 Design Guidelines . . . 25
3.2 The Project of 2013 . . . 26
3.2.1 Admin . . . 26
3.2.2 Cars . . . 26
3.2.3 Croc . . . 26
3.2.4 Parrot . . . 26
3.2.5 Tortoise . . . 27
3.2.6 Train . . . 27
3.2.7 Wasteland . . . 27
3.2.8 Zebra . . . 27
3.3 Acknowledgement . . . 28

II Wasteland Contribution **29**

4 Introduction to Wasteland **31**
4.1 Workflow . . . 31
4.2 GIRAF Architecture . . . 32
4.3 Problem Statement . . . 33

5 Process and Progress **35**
5.1 Approach . . . 35
5.1.1 SCRUM implementation . . . 35
5.2 Sprint 1 (Week 10 & 11) . . . 36
5.3 Sprint 2 (Week 12 & 13) . . . 36
5.4 Sprint 3 (Week 14, 15 & 16) . . . 36
5.5 Sprint 4 (Week 17 & 18) . . . 36
5.6 Sprint 5 (Week 19) . . . 37
5.7 Sprint 6 (Week 20, 21 & 22) . . . 37
5.8 Product Backlog . . . 37

6 Analysis **39**
6.1 2012 Material . . . 39
6.2 Necessary Data . . . 39
6.3 Requirements Analysis . . . 41
6.3.1 Contact Group Requirements . . . 41
6.3.2 Specific Applications . . . 42
6.3.3 Admin Group . . . 42
6.3.4 Security . . . 42
6.3.5 List of Requirements . . . 42

7 Design **45**
7.1 Database Design . . . 45
7.1.1 Profile . . . 45
7.1.2 Application . . . 47
7.1.3 Department . . . 47
7.1.4 Pictograms . . . 47
7.1.5 User . . . 48
7.2 Server API . . . 48
7.2.1 Philosophy . . . 48
7.2.2 Data Serialization Format . . . 49
7.2.3 Request Structure . . . 49
7.2.4 Response Structure . . . 49
7.2.5 Overview . . . 50
7.3 Server Application and Modules . . . 50
7.3.1 Connection Module . . . 50
7.3.2 Database Module . . . 51
7.3.3 API module . . . 52
7.3.4 Overview . . . 53
7.4 Synchronization Design . . . 54
7.4.1 The Application . . . 54
7.4.2 Creating a Local Database . . . 54
7.4.3 Uploading and Downloading Changes . . . 55

8 Implementation **57**
8.1 SQL . . . 57
8.2 Database Module . . . 58
8.2.1 Database Class . . . 59
8.2.2 QueryResult Class . . . 59
8.3 Connection Module . . . 59
8.3.1 The Connection Class . . . 59
8.3.2 The Listener Class . . . 61
8.3.3 The Framework Functions . . . 61
8.4 Builder Functions . . . 63
8.4.1 Fix . . . 63
8.4.2 Extractors . . . 64
8.4.3 Builders . . . 64
8.4.4 Validators . . . 66
8.5 API Calls . . . 66
8.5.1 Read and delete calls . . . 66
8.5.2 Create and Update calls . . . 68
8.5.3 Link . . . 71
8.6 Synchronization . . . 74
8.6.1 Main Activity . . . 74
8.6.2 Connection . . . 74
8.6.3 SQLite Database . . . 74
8.6.4 Downloading From the Central Database . . . 75
8.6.5 Uploading Updates to the Central Database . . . 76
8.6.6 Known Limitations of the Current Version . . . 78

9 Test **79**
9.1 Unit Tests . . . 79
9.2 Acceptance Test . . . 79

10 User Manual **81**
10.1 Installation Instructions . . . 81
10.1.1 Hardcoded Information . . . 81
10.1.2 Prerequisites for the server application installation . . . 81
10.1.3 Building the Program . . . 82
10.1.4 Running Unit Tests . . . 82
10.1.5 Prerequisites for the Puddle Android application . . . 82
10.1.6 Building Puddle Android Application . . . 82
10.1.7 Running the Puddle Android Application . . . 82
10.1.8 License . . . 82

11 Reflection **85**
11.1 Conclusion . . . 85
11.2 Project Status . . . 85
11.3 Future Work . . . 85
11.3.1 Known Issues . . . 86
11.3.2 New Functionality . . . 86
11.4 Project Evaluation . . . 87
11.4.1 Recommendations for next year . . . 87

III Appendices **89**

Appendix A API Documentation **91**

Appendix B Database Schema **97**

Appendix C Unit Test Example **101**

Part I

Common Report

Chapter 1

The GIRAF Project

Graphical Interface Resources for Autistic Folk (GIRAF) started out in 2011 as a semester project targeting children with autism and their guardians. In the following chapter, the overall vision for the GIRAF project will be presented, the projects from previous years will be explained briefly along with the platform for the project. Lastly a section describing autism is included.

1.1 Vision for GIRAF

The vision for GIRAF is to create a multi-purpose application based on *Android* that can simplify and ease the lives of autistic children and their guardians.
The purpose of GIRAF is to replace physical items that are being used daily by the children and their guardians with digitized versions. The idea being to gather several functionalities in one object and allowing customization for each individual child.
This will also optimize work procedures on the individual institution in such a way, that guardians will save time doing repetitive tasks such as making pictograms. This time could be spent with the children instead.
As of the spring of 2013 three schools and institutions for children with autism in Northern Jutland are involved in the development, but the hope is that GIRAF will be distributed across all similar institutions in Denmark.

1.2 Previous Years

During the first year of development, four parts of the GIRAF project were developed. The four projects were developed during the spring semester of 2011 and included the projects:

Admin An administration interface used for administrating different aspects of the GIRAF system.

DigiPECS A digitized version of "Picture Exchange Communication System"[11] a system used as an aid for communication with people with special needs such as autism.

Launcher A home screen application and distribution platform for Android.

aSchedule A visual schedule for the Android platform.

During the spring semester of 2012, five new software groups continued development of the GIRAF project. The projects developed during 2012 were:

Launcher An enhancement of the launcher project developed during the spring semester of 2011.

Oasis An enhancement of the admin project from 2011. Furthermore the Oasis project developed a local database for the GIRAF system.

Parrot An enhancement of the DigiPECS project from 2011. The project was renamed because of trademark issues.

Savannah A server side database with web interface for the GIRAF system.

Wombat An Android application for measuring and visualizing time.

During the spring semester of 2012 two databases were developed, however synchronization between them was never achieved.

Problems with Initial Implementation

As the spring semester of 2013 started, an "install party" for the students was held. The party was intended to help the students compile and deploy the projects from 2012.
Even though representatives from each of the 2012 groups were present, some compilation problems still occurred.
The repository used for distributing in 2012, was disorganized and difficult to navigate, i.e. due to:

- Multiple copies of the same project.
- Unclear dependencies among the different project.
- Projects only meant to be compiled from Eclipse for Windows.

During the following week a working workspace was created and shared with the rest of the students, along with install instructions. The install instructions were later updated to a more clear edition.

1.3 Target Platform

Android is an open-source operating system originally developed by Android Inc, and later bought by Google Inc. The first release came in 2007, where it was launched by Google Inc. together with Open Handset Alliance (OHA), which includes companies such as Samsung, HTC, LG and Google.
Before the first students were involved in the project in the spring of 2011 Ulrik Nyman considered two platforms for the development of the project. The Android and iOS platforms. The Android platform was chosen for three main reasons:

- That the platform is open source.

- That in Android the developers can take control of the functionality of the home button.
- That distribution of the software is possible outside the official marketplace.

For the the two following years it has been chosen to stay on the Android platform. This is done both to be able to reuse the source code and because Android compatible hardware is available for the students. In the very long term the system could support multiple platforms.

1.4 Autism

Autism is a spectrum disorder, meaning that it appears in different variants and not all people who are diagnosed have the same symptoms. The disorder can often be observed within the first three years of a child's life. Autism is a physical condition and is linked to abnormal chemistry in the brain, however the exact causes of these abnormalities are still unknown.[1]

Symptoms

Children with autism usually have difficulties understanding the concept of "play pretend", meaning that they have a hard time imitating the actions of others when playing and therefore prefer to play alone. Furthermore they have difficulties with social interaction and communication – verbally and non-verbally.
People diagnosed with autism may:

- Be very sensitive to light, noise, touch, and taste.
- Have a hard time adjusting to new and changing routines.
- Show unusual attachments to objects.

Autism diagnosed individuals may have a hard time starting and maintaining a conversation. They may communicate with gestures instead of words, develop language slower or faster than normal and some do not develop any language at all. Furthermore the lack of social interaction means they might have a hard time making friends, may be withdrawn and may avoid eye contact.[1]

Signs and tests

If a child fails to meet any of the following language milestones, it may be an indication that it needs to be tested for autism:

- Babbling by 12 months.
- Gesturing (such as pointing or waving goodbye) by 12 months.
- Saying single words by 16 months.

Children failing to meet any of the previously mentioned language milestones might receive a hearing evaluation, a blood test and a screening test for autism. Since autism covers a broad spectrum of symptoms, a single brief evaluation cannot predict what abilities the child has. Therefore a range of different skills are evaluated, such as:

- Communication
- Language
- Motor skills
- Speech
- Success at school
- Thinking abilities

Some parents might be scared of having their child diagnosed, however without a diagnosis, the child might not get the necessary help.[1]

Treatment

Autism cannot be cured, however an early diagnosis and treatment can greatly improve the child's quality of life. Different treatment programs usually build on the child's interests and are highly structured to their needs and routines.[1]

Chapter 2

The GIRAF Project 2013

When working in a multi-project consisting of eight groups, it is important to have a common goal for the project. This chapter describes this goal as a story. Furthermore the chapter includes a description of the development process and the rules of conduct.

2.1 The Goals for 2013

Within the first couple of weeks, when all the groups had been assigned a project, a major story for the overall project was written.

The Major Story for 2013

> "The guardian arrives at the institution, and turns on the tablet. The guardian is aware of the arrival of a new child at the institution after lunch. The guardian sets up and customizes a profile for the child, this includes creation of new pictograms. Furthermore the guardian prepares games and a life story for the child.
>
> After lunch the new child and the guardian meet. The child is introduced to the communication tool Parrot. After some introduction they sit down to do some communication practice using the tool.
>
> Afterwards the child wants to go outside to see the rest of the institution, and needs to put on some outdoor clothes. The guardian introduces the child to the Zebra tool, and together they put on the child's outdoor clothes.
>
> When the child comes back in, the guardian and the child play the games prepared earlier by the guardian.
>
> When they are done playing, the child and the guardian read the child's life story using Tortoise."

2.2 Definition of a Multi-project

A multi-project is a project that includes multiple groups that each work on their own sub-project, which is part of a larger project. In this case, the larger

project is the GIRAF system and each group works on a separate part of the system.
Compared to working on a single project in isolation, working together creates new challenges. The software produced by each group has to be integrated to ensure the entire system works properly. Some projects are more independent of the rest, while others depends heavily on some projects like the database project Wasteland described in Section 3.2.7. Groups have to be flexible and pass any requirements to other groups' projects early to prevent halts.
To ensure the project is successful and no misunderstandings occur, there must be good communication and cooperation between the groups. This requirement is amplified by the fact that there are no definitive authoritative figures, other than those chosen by project members.

2.3 Group and Work Structure

This section describes the development methods used during the spring semester of 2013, including stories and project management tools.
The section is rounded off by a description of the development tools used, including Redmine, Git, and Jenkins.

2.3.1 Development Method

Having a development method is one of the main ways to structure the work process of a project. A development method is a collection of methods and structures, from the way to have meetings, gathering requirements and structuring the development. There are many development methods, each is structured and handles issues differently, however, it is rare that one fits a development problem perfectly. Different methods are often combined and customized to fit the problem at hand.

Implemented Development Methods

This project's nature calls for agile development, due to team collaboration, user feedback, product focus, and continuous integration. Agile development focuses on a flexible but structured work progress suited for projects with many unknown variables. The agile development method has the ability to adapt to changing requirements throughout the project and focuses on having a shippable product at the end of each iteration.

Stories

User stories is one of the tools that helps streamline the work process, it keeps focus on a shippable product and is the main component for management of the project. First of all the product story works as a common problem statement for all work groups. A product story is the agreement on what is necessary for the product to be finished. From the product story each group can extract what is required of them to complete the story.

Management

The semester coordinator, Ulrik Nyman, has supervised the project since it's beginning. Ulrik Nyman himself has a child with autism and will continue being a part of the project for the time to come, conveying his knowledge of the development process and the product. To help fit the product to the needs of guardians, for which the product is intended, a number of representatives are included for more detailed feedback on the process and the product.
To keep as many work hours in development and to keep a good overall management, common meetings are held weekly. The common meetings focus on sprints and team cooperation. Problems that need further discussion and/or development are discussed by a committee consisting of a few representatives from each group.
The common meeting and committee meeting are further specified in sections Section 2.4.1 and Section 2.4.3.

2.3.2 Development Tools

A number of tools were used in order to optimize team collaboration and to make the projects more accessible. These tools are further explained in the following sections.
A dedicated Linux server was commissioned for the entire GIRAF project and several services installed to facilitate collaboration and agile development. Common to all current services are their free, open-source nature and support of LDAP authentication, allowing all students and supervisors to log in using their AAU credentials.

Redmine

Several tools were audited for use in the project management aspect of development, including Trac, PivotalTracker and Github. Redmine, a Ruby-On-Rails web application, was selected primarily due to its support of multiple projects and support features such as wikis, forums, milestones and various charts. The features most broadly used are:

Projects All projects live in a shared project space, and can be placed in a hierarchy under a super project. In this regard, the primary multi project served as the base of each of the eight groups' underlying projects.

Issue handling Redmine's primary feature is its issue handling. Project members can create and react to issues within custom-defined domains. For GIRAF, this was primarily development tasks, but could just as well be used for report-related tasks or general maintenance in an attempt to manage time usage.

Burndown Charts Redmine does not have native support for burndowns, but does support it through a Free and Open-Source Software (FOSS) third-party plugin. Burndowns are a visual aid of each subproject's progress throughout a sprint, giving quick summary of development speed and whether proactive action may need to be taken.

Milestones A generic milestone feature in Redmine is Versions. Versions are simply markers with a set date, and can be open or closed for attachment of issues. The burndown plugin couples a version's end date with attached issues and their progress to generate the related charts.

Wiki A per-project wiki module exists in Redmine. The basic wiki markup has been expanded to allow referencing of almost any other element in the project hierarchy, such as projects, issues, files and VCS revision.

Redmine has many more features not directly applied during this project period. However, many could be applied to create a more centralized and structured development experience in future projects. Examples include file and document hosting, advanced issue workflows, permission management and VCS integration. Future multiprojects may consider expanding into these fields if they feel proficient in Redmine's basic usage.

Version Control System

The university's IT services offers only a single version control system, Subversion. Although centrally supported and backed up regularly, Subversion's shortcomings were challenged before main development had begun. Most notably, the system's centralized workflow and high operation cost. Many of SVN's actions require access to the central server. Two alternatives without these issues were suggested: Git and Mercurial (Hg). The former was chosen as a general question of broad platform support and popularity. A primary strength of these systems is their support of separate branches of development without the constant need to connect to a central server. This allows developers of each project to synchronize with a main branch while maintaining several development branches on their own workstation.
Most groups used Github as hosting solution for development of their projects, as a git hosting solution was not immediately forthcoming (contrary to Subversion and Mercurial, Git does not have a default server implementation). At the conclusion of the project period, a solution was configured using Apache-based LDAP authentication, deferring authorisation and repository management to Gitolite, a low-footprint open-source offering.
In the interest of easier cross-project code contribution and inspection, an improved web solution may prove a better choice. Due to time constraints, a few solutions were briefly audited but ultimately discarded in preference of Gitolite. Gitlab should be mentioned as it featured an interface and features very close to those of Github itself, but proved difficult to install and maintain.

Jenkins

A principal element of agile development is continuous integration, the automated concurrent building of new code as it is pushed to central repositories which ensure constant availability of newest binary packages while catching coding errors before pushing them to the public. Jenkins, a fork of Oracle's Hudson, was suggested early and, given no proponents, was implemented. Build jobs were set up for each project, polling their origin repositories for new Git builds to main branches. If a repository has new code, it is downloaded and built. In case of build errors, the project developers are notified by email. To facilitate the

deployment phase of each sprint, all projects are rebuilt every Thursday night and pushed to a public FTP server as well as making them publicly available by HTTP.
Git support is not part of Jenkins' core feature set, but is available as a plugin. During development, unhandled exceptions in the plugin code resulted in thousands of of superfluous builds as a failed build due to unexpected circumstances was not marked as failed.

2.4 Decision Making - The Process

The following section will describe the decision making process, set in place to ensure that everyone would be heard on an equal and democratic footing. The decision making process during this semester's multi-project consists of two different steps.

2.4.1 The Weekly Meeting

It was strongly recommended by the semester coordinator, Ulrik Nyman, to hold a weekly meeting for all software students on the bachelor semester of 2013. The meeting's agenda consists of a few points of formalism at the very beginning, in which a secretary and a moderator are chosen by means of voting. Candidates for these roles are entirely self-appointing and a vote is issued to pick one of the candidates.
Though the weekly meeting is established to ensure a higher level of communication between students, as well as ensure that decisions will be taken on a multi-project level scale, not all points are actually discussed at this meeting. Instead, a committee approach is agreed upon, see Section 2.4.3. The purpose of establishing committees is to ensure that relevant discussions to a given topic can be had, but within a smaller audience.
Committees are discussed at the weekly meeting where voting determines which committees are established. A chairman for a committee is self-appointed and a vote determines if there is consent to let the given person be chairman.
The meeting will then proceed and discuss the ideas and suggestions agreed upon within each committee from the previous week and at the multi-project level determine, by voting, which ideas are okay, or if any of the points concluded by one of the committees are subpar and should be reworked.

2.4.2 Rules of Conduct

During the first weekly meeting some general rules of conduct were established, including decisions on how voting should be done. A number of ways to do this were suggested. Ultimately it was decided that every person present at the meeting has an individual vote, and the idea of a group based voting system was therefore discarded. Furthermore in the event that there is a 50/50 split, the vote will have to be reissued. There must be majority 'for' or 'against' a decision. Guidelines for when a decision should be taken at the weekly meeting were established as well. If a decision involved only two or three groups, then it would not be necessary to discuss at the weekly meeting. If, however, the

decision impacted everyone, a committee would be established to make these decisions.
During a committee meeting every group has a single vote. It is possible to send as many group members as is deemed necessary to the committee meetings, however, it does not increase the number of total votes a group has.

2.4.3 Committees

A committee ideally consists of a representative from each multi-project group and a chairman agreed upon at the weekly meeting. The chairman is responsible for setting up the meeting, time, place, agenda as well as writing down the details of what is agreed upon during the committee meeting.
The resulting work product of the committee is a document, that potentially answers every question on the agenda, ready to be presented at the next multi-project meeting.

Important Committees

The following section describes an extract of some of the most important committees, that were established during one of the first weekly meetings.

- Wiki: *Ensures that the multi-project wiki page on Redmine is created in a uniform way by establishing guidelines for new articles.*
- Design Guidelines: *Ensures that the User Interface design of the GIRAF application is uniform (e.g. in regards to font, color scheme and various buttons - green for 'yes' and red for 'no').*
- Common Report: *This committee is responsible for the creation of the common-report chapters, which you are reading now, that are at the beginning of every project report.*
- Pictogram Class: *Because every group requires a common pictogram class, it was decided to create a Pictogram Class committee to determine the functionality that this class needed.*
- GIT: *The GIT committee is responsible for working out a common structure across all repositories to create uniformity and make it easier to continuously integrate.*
- Public Pictogram: *Determines guidelines for how pictograms are handled in the database (e.g. who has access rights to what and why?).*
- Story: *The story committee is responsible for creating a story to follow every sprint. It puts the sprint's tasks into an overall context.*
- CI/Git: *This committee is responsible for coming up with solutions to potential issues that might occur as part of the Continuous Integration step when using GIT.*

Chapter 3

What was developed

This chapter describes the work done for GIRAF in the year 2013 and is rounded off by acknowledging the involved contacts and the semester coordinator.

3.1 Pictograms, Morgana, and Design Guidelines

In this section the notion of a pictogram is be presented followed by a description on how pictograms are currently being used and why they should be digitized. Furthermore the section includes a description of the Morgana library.
The section is be rounded off with the overall design guidelines for the entire GIRAF system.

3.1.1 Pictogram

In the context of this report a pictogram is defined thus: *A pictogram is an image representing a living being, a physical object or some form of action.* Pictograms can contain a text-label, describing the respective images, for clarification. There is currently no standard for the layout or contents of pictograms, due to the specific needs and opinions of the users. User **A** might like to have black and white images with text labels whereas user **B** might want colorful images without text. The images can themselves vary from cartoons to photographic representations. Pictograms are commonly used as means of communication, especially by those requiring assistance with communicating, including but not limited to individuals with autism.

Current Use

During the spring semester of 2013, when this report was written, the use of pictograms is mostly in the form of physical images. The images need to be drawn and/or edited, printed, cut out and then laminated to extend their lifespan. After this process the pictograms are ready for use, generally for one individual, making this repetitive and tedious for the guardians.When the required amount of pictograms have been created for an individual, they need to be organized and made accessible with the help of some sort of container. This container

Figure 3.1: Pictograms in use 2013

can be a folder with a pocket for the pictograms and a velcro-like strip for arranging the pictograms. For communication an individual can choose to form sentences by arranging the pictograms accordingly or use a single image to simply express needs and wants. Another purpose of the pictograms is depicted in Figure 3.1 where instructions are graphically represented for various tasks, in the form of "do **A**, followed by **B** and lastly do **C**" for individuals requiring special assistance.

Digitizing the Pictogram

The GIRAF project focuses on simplifying and digitizing a medium used by individuals with autism and their guardians. This includes digitizing the pictograms, making them available on devices running Android with added functionality. Added functionality includes the option to make the pictograms play a sound, dynamically change the layout of text-labels and editing images. Digitizing the pictogram also makes it possible to share them easily, carry them between devices and make backups of them. Previously, with the same idea in mind, it was attempted to digitize the pictograms. It was considered unsatisfactory (see section below) and therefore the re-implementation in this semester's project.

GIRAF Pictogram Design

The digitized pictogram consists of an image, text-label and a sound. With all elements included, it can be presented as each of the three, two parts combined or all three in union. This viewable container is designed as an extension of

the *Android* view class, making it easy for developers to include and present in their applications. The idea is to have users sharing the same pictograms, with the option to customize their contents without affecting the pictogram itself. The previous GIRAF pictogram design lacked documentation, portability and functionality such as text-labels. Therefore a new design was implemented, which hopefully fits the needs of both future GIRAF developers and GIRAF users.

3.1.2 Morgana

The Morgana library project was initially intended to make it possible for all the GIRAF applications to use both the Wasteland database, see Section 3.2.7, and the local Oasis database seamlessly, however in the time allotted it was not possible to finish this functionality, so the focus was shifted to making it parse and write JavaScript Object Notation (JSON) objects for use in calls to the Wasteland database.
The library implements a Java class for each value object documented in the Wasteland Application Programming Interface (API), each class parses a JSON object and turns it into an object which can be used by GIRAF applications, it is also able to create JSON objects from the stored Java object.

3.1.3 Design Guidelines

The purpose with the guidelines is to get a consistent look and feel across all of the different applications included in the GIRAF system. The design guidelines have been discussed among all of the project groups, and they are as follows:

- Keep the existing color palette
- Font: Helvetica
- Font size: use common sense. *Android* offers extra small/small/medium/large/huge
- Minimize the use of text, use images instead of text
- Graphical User Interface (GUI) in vector graphics
- Green and red are universal colors for 'accept'/'cancel'
- Applications have animal icons
- Icons are non-customizable
- Every application should be locked in landscape mode

The color palette will be the same as in the 2012 version of GIRAF. With regards to font type and size, Helvetica has been chosen and developers need to keep in mind, that the text has to be readable on the tablet.
The aim is to use more images and less text as the target audience are mostly children, many of which have communication and/or reading difficulties and some have problems imagining objects purely from text.
The GUI will be in vector graphics, because it scales well, which makes it possible to reuse some of the images. Green and red are universal colors for 'accept'/'cancel'. It may sound obvious but other applications have been developed with different colors. Tool-applications should have animal icons.
Lastly everything will be in landscape mode as this eliminates additional implementation for responsive layout, when the tablet is rotated.

3.2 The Project of 2013

3.2.1 Admin

This project focuses on the creation of an administration interface for the GIRAF system. The Admin system consists of two parts, one for a desktop computer and one for *Android*. The desktop part will run on a Linux, Apache2, MySql and PHP (LAMP) stack and communicate with the database using the database API provided by the Wasteland (see Section 3.2.7) project. The *Android* part will run on the tablet using the same code base as the desktop part, using a web server application. The main focus of the project is for department managers and guardians to be able to administrate the GIRAF system.

3.2.2 Cars

The aim of the Cars project is to develop an application, which will help children with infantile autism to be more comfortable in using their voice. To ensure that the children learn to use their voice in creating different types of sounds, and not just speak in a monotone way, the application will require the children to create sounds covering different sides of the frequency spectrum.
Cars is a game in which the player has to lead a car through a street into a garage, controlling it with high or low frequency sounds. The car has a matching colored garage at the end, which when entered completes the game successfully. Randomly placed obstacles are used to force the player to avoid them to reach the end.

3.2.3 Croc

The Croc project aims to create an application for creation of pictograms for use in the GIRAF system.

Pictograms can be created in a number of ways:

Camera take a picture with the camera and turn that picture into a pictogram.

Drawing draw a pictogram.

Audio record sounds to attach to pictograms.

3.2.4 Parrot

Parrot is an enhancement of the Parrot project of 2012 and is an application for communication between guardian and child. Its development is based around the currently used physical system Section 3.1.1. The original Parrot application from 2012 also included the administration of categories. It was therefore technically possible for a child using Parrot to access these administration tools, and it is for this reason, that the currently developed version has relocated the administration to a separate application named Category Administration Tool. The version developed during this project will focus on making improvements to the GUI design, adding subcategories (such as breakfast item under the food category) and handle the interaction with pictograms. The primary focus for

Parrot remains the same; providing an easier way for children to communicate with guardian in a way that they are familiar with.

Category Administration Tool

Category Administration Tool (CAT) focuses on administrating categories and subcategories. Currently CAT is also responsible for communicating with other applications that need specific pictograms, such as the Tortoise (Section 3.2.5) and Zebra (Section 3.2.8) applications, by providing search/deliver functionality.

3.2.5 Tortoise

The Tortoise application focuses on helping children learn about their own lives and strengthen their social skills. The hope is, that by letting the child interact with pictures and sentences, that are associated with their life, the child can develop an identity. By developing their own identity, the child will learn how to interact with other people by learning what kind of topics to talk about in a conversation with others.

3.2.6 Train

The inspiration for Train comes from an exercise, that one of the guardians practices with the children. The purpose of the game is to create a dialogue between the child and the guardian. The child has to drag pictograms from a train station onto the train wagons and make the train drive. When the train arrives at the next station, the child has to drag the correct pictograms from the train and onto the station. The correct pictograms are decided by the station category.
The category for each station is chosen by the guardians by clicking the category picture frame and browsing CAT (Section 3.2.4) for the picture they want to use. After selecting a category, they select which pictures they want associated with the station.

3.2.7 Wasteland

The purpose of the Wasteland project is to handle all of the data for the GIRAF system. In order to achieve this goal, a database will be implemented on a central server and a local database will be kept on the tablet. The two databases will synchronize data on a regular basis.

3.2.8 Zebra

The aim of the Zebra project is to create a software application aiding guardians in their work. The application should aid the guardian in situations where a child is to perform an ordered sequence of actions. These actions are typically represented by pictograms. Zebra should replace the current paper based version of this system. The guardian should be able to create and manage digital versions of such sequences specific to each child. Upon selecting a sequence for the child to follow, the child should be able to mark actions as done when they are completed to illustrate their progress.

3.3 Acknowledgement

The group of students working with GIRAF during the spring semester of 2013, would like to thank the contacts, who were:

Tove Søby - speech therapist, and contact for three groups.

Mette Als Andreasen - kindergarten teacher at Birken Langholt, and contact for two groups.

Kristine Niss Henriksen - kindergarten teacher at Birken Vodskov, and contact for one group.

Drazenko Banjak - teacher at Egebakken Vodskov, and contact for one group.

Mette Frost - teacher at Egebakken Vodskov, and contact for one group.

In addition the group would like to thank Ulrik Nyman, semester coordinator, for his help, guidance and engagement during the project.

Part II

Wasteland Contribution

Chapter 4

Introduction to Wasteland

In this chapter the workflow of the Wasteland project is presented along with the overall architecture of the GIRAF system and the problem statement for the Wasteland project.

As mentioned in Chapter 1, GIRAF's main objective is to improve communication between children with autism and their guardians. The primary tool for the communication will be a tablet that the children will use either alone or with their guardian. In this context a single tablet needs to be able to accommodate many different children with individual preferences. The children should not be dependent on one particular tablet, but should be able to use any tablet with the GIRAF system installed. As a consequence the GIRAF project will need two types of databases. A local database on the individual tablets and a central database that the tablets should be able to synchronize with. As children with autism have very specific needs, their individual preferences need to be stored on one tablet and possibly retrieved on another. Without the databases the GIRAF system will not be able to store individual preferences and will not be able to synchronize preferences across different tablets.
A problem statement will be given at the end of this chapter.

4.1 Workflow

This section will describe the act of accessing the central database through an API from the Android applications, and explain the steps taken for this to happen. There are two ways to access the database, one for the Android applications, and one for the administration interface. This is done because the administration interface needs to be able to edit settings and manage users and profiles for the children. This means that the administration interface can connect directly to the central server via a PC.
When an Android application is requesting data from the local database called OasisLib, and the data is not in the local database, the request will be sent to the central server. The central server then makes a request to the central database, and sends the results back to the local database. The application will then get the results from the local database. This is done so the application does not have to know whether the requested data is in the local or central database,

the application will get the data either way, if there is a steady connection to the server. Requests between the applications and the local database, as well as between the local and central database, will be written in JSON. JSON was chosen because the request has to be interpreted by different systems written in both Java and C++.
To be able to accommodate many different Android application settings in the database, all settings are stored in binary large objects, also known as blobs. Blobs are stored as a single entity in the database, and will include all settings of an application. Because of this, every application must be able to encode and decode their own blob.
Every Android application can have different settings for each user, in most cases likely a child, that has access to use it. This, for example, makes it possible to save each child's favourite pictograms, without having to save them for other children. This feature is handled by the database.

4.2 GIRAF Architecture

In this section the overall architecture of the entire GIRAF application will be presented. First the top level architecture will be described and subsequently the architecture on the tablet will be presented.
As mentioned in Chapter 1 the GIRAF project is divided into several subprojects. Each subproject is responsible for a specific part of the system. The itemized list below is a reminder of what the various projects are responsible for.

- Admin - Tool for administration interface for the entire GIRAF project
- Cars - Game developed with sound input
- Croc - Tool for creation of pictograms
- Parrot - Tool for pictogram categorization and child to guardian communication
- Tortoise - Tool for creating a life story for the child
- Train - Games with visual focus
- Wasteland - Tool for database synchronization
- Zebra - Tool for sequencing of pictograms

The top level architecture for GIRAF is depicted in Figure 4.1. The Admin group's desktop interface is connected via Internet to the central database. This enables guardians to manage the various children in the database. The Local database is installed on the tablet along with the applications. The applications save their settings on the local database, this enables efficient use of the tablet without an Internet connection. If the tablet has a wireless Internet connection it is able to synchronize the local database with the central database thus saving the settings across the entire GIRAF system.
The applications installed on the tablet are divided into two categories namely games and tools, respectively. The tools provide GIRAF with core functionality

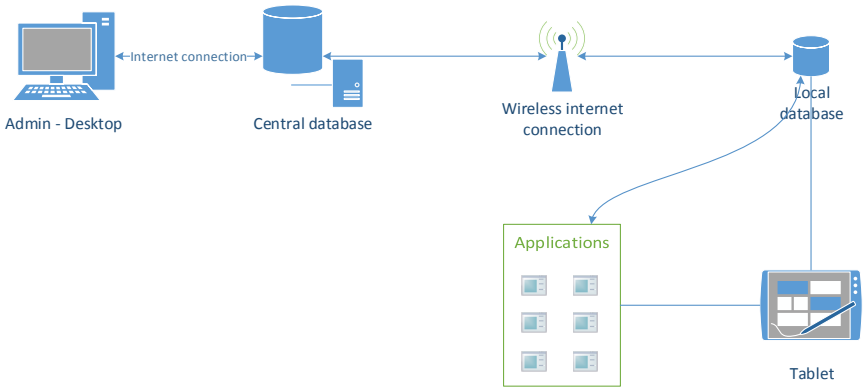

Figure 4.1: Top level view of GIRAF

such as allowing guardians to take pictures with the tablet's onboard camera and add the new picture to one or several children's personal image galleries. They also provide the ability to use pictograms across all applications on the tablet. The games provide some fun learning aids e.g. where the children can learn to control pitch and volume of their voices through a game.

4.3 Problem Statement

The GIRAF system should seamlessly synchronize information across a large number of tablets.
Today the GIRAF system is able to store its information locally on a tablet. But each tablet needs to be customized to individual children's specific needs. However the children should not be dependent on a specific tablet. This means that some sort of synchronization is needed.
This requires a central database to be set up on a server, allowing the users to connect and update their local databases via the Internet.
We will use a SCRUM development method in addressing this issue. We will develop a central server application to facilitate synchronization between tablets.

Chapter 5

Process and Progress

This chapter describes the development method used during the semester and the progress of the Wasteland project. The product backlog is presented along with an overview of which tasks were completed during a sprint, and which were delayed to later sprints. Furthermore reflection on the sprints and the work performed during them is described.

5.1 Approach

As described in Chapter 1, a multi project of this nature requires a lot of structure and planning. In one of the first multi group meetings, all of the groups agreed on a development method, namely SCRUM. The development method used by the individual groups was up to them to decide. It was decided that all of the groups would meet on a weekly basis and discuss the progress of the individual groups and problems that might arise.

5.1.1 SCRUM implementation

For simplicity the group decided to use the same development method that was used on the multi-project level. There were however some alterations made to better suit the group's development style.

Sprint Lengths

Lectures during the semester meant that work was divided into half days. The group decided to use the sprint lengths agreed upon at the common meetings.

Pair Programming

Pair Programming has been borrowed from the XP-development method as the group had positive experiences with the technique and found that it was well suited to the project.

5.2 Sprint 1 (Week 10 & 11)

For this sprint the goal was to create a connection library, a database library, a method of authentication and a database schema. At the end of the sprint the following was achieved:
Connection Library - The connection library was able to receive and send requests between two different computers. Unit tests were also written for this library. The connection library was finished and compiled on Linux.
Database Library - The basic database library functionality was completed, and unit tests were written for this functionality.
Authentication - Authentication was postponed to a later sprint.
Database Schema - A database schema was designed with attributes and constraints. At this point in time the group expected the schema to evolve further as new data needed in the database might be discovered.

5.3 Sprint 2 (Week 12 & 13)

The goals for this sprint was to design a database API, set up basic communication between this project and the Admin group, and to make a problem statement.
This sprint was very short due to a lot of courses and Easter holidays. Because of this, communication was postponed to a later sprint. The first draft of an API was created.

5.4 Sprint 3 (Week 14, 15 & 16)

Goals for this sprint included creating a JSON encoder and decoder, creating an OasisLib dummy, making the database build script and creating API-calls for all reads.
JSON encoder/decoder - The JsonCPP library was chosen for this task. It made it possible for to receive JSON API-calls and translate them to C++ data structures used by the server application.
OasisLib dummy - Made it possible for other groups to bypass the local database (which could not communicate with the central database), and connect directly to the central database when making calls using the API. The idea was that when a local database was functional, other groups could make use of it without changing their applications.
During this sprint JSON encoder/decoder, OasisLib dummy and database build script were finished. The API-calls, however, were delayed to sprint 4.

5.5 Sprint 4 (Week 17 & 18)

The goal for this sprint was to finish the API-calls in order to complete the work on the central database. And to set up communication with the Admin group. Work on synchronization was started.
Most of the API-calls were not completed in the previous sprint. Apart from some of the read-calls, all the calls still needed to be implemented and tested.

Most of the calls were implemented and unit tested during this sprint. The central server crashed due to bugs in the code and required restarting a few times. The synchronization was not done at the end of this sprint so that, and the rest of the API-calls, was postponed to sprint 5.

5.6 Sprint 5 (Week 19)

This sprint was intended for debug, but due to delays in earlier sprints, the group had to implement the last API-calls and test them, and kept the server up and running. The group encouraged the Admin group to build the server on a local machine to ensure that they could continue working if the server crashed. One-way synchronization between the central database and a local database was finished during this sprint, but work on the testing and the two-way synchronization spilled over into the next sprint.

5.7 Sprint 6 (Week 20, 21 & 22)

The goal for the final sprint of the semester was to complete the documentation of the work done in the form of a report and to finish up the synchronization. Some of the documentation was written along the way but most of the report was still not done.
In this sprint the documentation was completed and proofread and the final version of basic two-way synchronization was implemented and tested.

5.8 Product Backlog

For requirements management the tasks were prioritized, estimated with regards to complexity and added to a product backlog and as work progressed new task were added to the backlog and tasks that were completed were marked as finished. Due to the fact that estimation in software development is often very difficult, the estimates were done in numbers representing the expected effort required to complete them, not in hours or days. The priority goes from 1-5, with 1 being highest priority. The following table contains the product backlog for the project.

ID	Name	Priority	Estimated time	Status
1	Connection Library	3	5	Finished.
2	Database Library	4	3	Finished.
3	Authentication	2	5	In progress.
4	Database schema	1	2	Finished.
5	Design database API	2	5	Finished.
6	JSON encoder/decoder	2	2	Finished.
7	OasisLib dummy	5	3	Finished.
8	Implement API calls	1	21	Known bugs.
9	Establish SQLite database	2	3	Finished.
10	Implement synchronization	3	8	Finished.
11	Write report	1	13	Finished.
12	Proofread report	4	8	Finished.
13	Install instructions	4	2	Finished.

Chapter 6

Analysis

In this chapter the material from the 2012 database version of GIRAF is analyzed and evaluated for future use. An assessment is made of the data needed in order to capture everything that the institutions need for the GIRAF system to suit their needs. Lastly a requirements analysis is performed with a resulting list of requirements.

6.1 2012 Material

The objective for the project was to continue the work on the GIRAF system that was handed over by the 2012 bachelor students.
Initially the task was to enable synchronization between the central database and the local databases on the individual tablets. Unfortunately the latest version of the 2012 central database project seemed to be missing, because the only version that was available did not compile. And as mentioned in Chapter 1 the documentation on install instructions were lacking to say the least. So the only thing that was reusable from the 2012 semester was the database schema which could be used as a mock-up. This meant that the workload had increased substantially as the group now had to implement a central and local database as well as handle the synchronization between the two.
The 2012 server was written in Java and the argumentation was that the students had a lot of experience using Java. This group on the other hand, has had introductory courses in C and C# and has little to no experience using Java. Another argument was that a server has to be able to handle a large amount of requests. Given this fact, the group felt confident that C++, a language based on C, would be excellent for the development, considering that C++ performs better than Java.[5]

6.2 Necessary Data

In order to create a suitable database schema, the first step is to figure out what should be in the database. This analysis focuses on the structure of the institutions involved in the GIRAF project.
An institution can have several departments. Each department has a number of employees, hereafter referred to as *guardians*, assigned to it as well as some

children that attend the department. Each department has one or more administrators, an administrator is a guardian with some extra privileges and authority. Each guardian is responsible for a few specific children, but is of course not limited to only taking care of the ones he or she is responsible for. This is however something that they handle internally. As Figure 6.1 illustrates, each department has a number of guardians and children and one of the guardians acts as an administrator for that department. Children are assigned to one specific department. Generally each department has its own administrators, but in some cases a single administrator handles several departments.

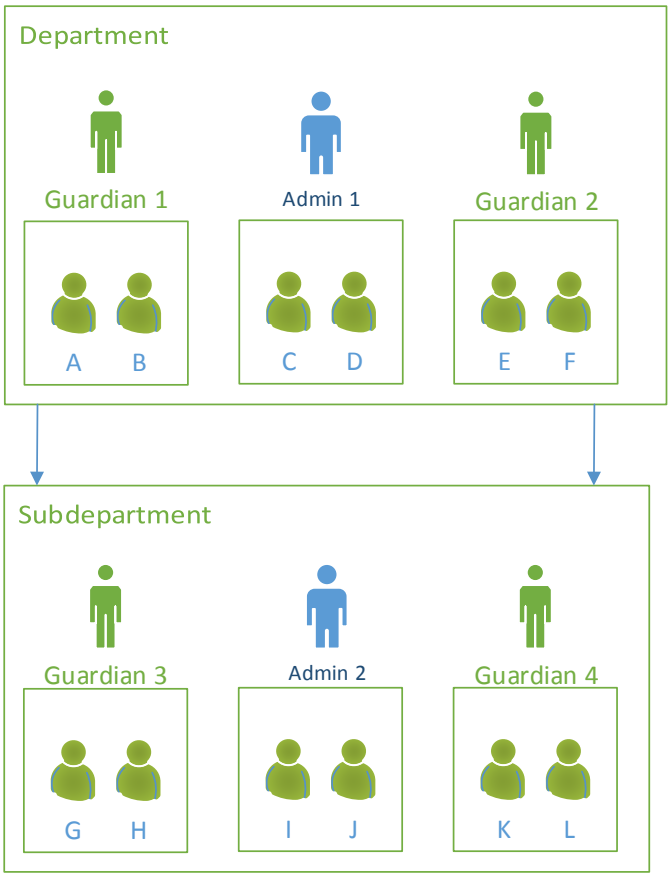

Figure 6.1: Overview of the people involved and department structure

The children have needs and demands that can vary greatly from one child to another. But common for all of the children is that they each have their own set of pictograms. The children generally have a resistance to change e.g. the taxi that drives them to the institution and picks them up again has to have a specific colour. This tendency can also occur with regard to preferences as some children insist on their pictograms being black and white with stick figures and

other prefer coloured images.
When these things are applied to what could be used in the database schema, there is a need to be able to represent *departments* with optional *subdepartments*. There needs to be a representation of *guardians* and *children*. The system should also be able to give guardians *administrator* rights to departments. The *pictograms* need to be included and it might be a good idea to be able to categorize the pictograms e.g. cereal and milk under breakfast items.

6.3 Requirements Analysis

It is a proven fact that miscommunication between developers and customers or users can lead to misunderstood, unnecessary or unwanted functionality.[6] As a result, development time is wasted on functionality that will not be used. The goal with the requirements analysis is to end up with a list of concrete requirements that will satisfy the customer's demands and will fulfil the problem statement without wasting time on unnecessary tasks.
The requirements have been collected from some of the contacts mentioned in the common report and from Ulrik Nyman, the semester coordinator. The Wasteland project is a bit peculiar in this context, because it handles the data behind the functionality in GIRAF and not so much of the functionality itself. The reason for Ulrik Nyman's inclusion is, that this is a student project and someone else will take over later on. And Ulrik Nyman will be involved in future development of GIRAF and he has some specific requirements in this context.

6.3.1 Contact Group Requirements

The contact persons held a lecture about how they use the various tools and techniques available to them and which pros and cons they each had. They have been involved in previous versions of GIRAF and have tested some of the existing functionality such as Wombat (see Chapter 1). They mentioned that some of their most useful tools were things such as timers that visualize the time spent and time remaining. They were interested in being able to take pictures with the tablets and assigning the pictures to one or several children in the department. They also requested that children were not dependent on a specific tablet and that one tablet could accommodate several children with their own specific preferences. The contact group mentioned that they sometimes take the children on excursions where there rarely is any Internet connection. They said that it would be nice if the tablets could be used in such a setting and the changes that were made would then only be saved locally until they got a Wi-Fi connection e.g. pictures taken on the excursion.
Ulrik Nyman had a few but more specific requirements. Such as install instructions that were simple and easy to understand and follow. He also specified that he preferred a small amount of well written and appropriately commented code over a large amount of uncommented and less structured code. Lastly he asked for some good documentation of the work done by each project group along with a list of functionality that would be well suited in the future.

6.3.2 Specific Applications

The various applications in the GIRAF system are divided into tools and games. Common for all applications is that they need to be able to distinguish between the different profiles. The games, for example, may need to be able to save settings and high scores for the individual child and some tools need to be able to access the child's pictograms. In these cases the applications need rights to add data to the database.

6.3.3 Admin Group

The Admin group is responsible for all desktop administration of the GIRAF system. Their goal is to allow administrators desktop access to the various children's profiles and allow them to create, edit and delete profiles. They also allow for administrators to make changes that affect entire departments, thus making it a lot easier to add some specific pictograms to all children of a given department.

6.3.4 Security

As of the summer 2013 the GIRAF project is still in development. The current login system consists of a QR-code that the guardians scan with the tablet's camera in order to log in. The QR-code contains the username and password for each individual guardian and the QR-codes can easily be copied and could present a security breach. When the system eventually gets released to the public, there needs to be a high degree of security implemented in GIRAF. The system will eventually contain personal and sensitive information and there are laws that regulate how this information should be managed.

6.3.5 List of Requirements

The activities mentioned in the previous sections have been evaluated and the requirements established from these are listed below:

- A central database
- A local database on each tablet
- Synchronization between the central and the local databases
- Android application that can control the synchronization
- An API that provides applications with easy database access
- Ability to use local database without Internet access.
- An install manual
- Good documentation

The requirements are fairly straightforward, however the last three requirements are a bit different from the rest. The ability to use the local database without Internet access is not essential, but would be nice if there is time as specified by one

of the contact persons. The two last requirements are Ulrik Nyman's requirements as there have previously been problems with poorly documented code and install instructions. This is a very important requirement as the GIRAF system will be further developed, either by students or professionals.

Chapter 7

Design

In this chapter the design of the database, synchronization between local and central database, an API for communicating with the server and modules for connecting to the database and serving clients is described.

7.1 Database Design

The database has to store all the information in the GIRAF system. To get an overview of the entire database, an entity-relationship diagram (ER diagram) has been constructed. Figure 7.1 illustrates the ER diagram for the database. The ER diagram uses a collection of basic objects called entities and their relationships to describe concepts in the real world and their individual relations. The group based the current diagram on the 2012 version, re-using some attributes for entities, but cleaning up relations and adding new content needed by other groups (e.g. pictogram category).

7.1.1 Profile

The profile entity includes both children and guardians, it holds information such as name, phone number, picture, e-mail, address and role (i.e. child, parent or employee). The roles are important in determining the privileges of the account. Profiles with the parent role can only administrate their own and their children's profiles, whereas profiles with the employee role on the other hand can administrate the profiles of the parents as well as the children.
Settings such as preferences with regards to background colour and the like, can be extremely important to an autistic person, and varies wildly from person to person, and thus these need to be stored in the database. They are saved in a blob because the data type is faster to use and because the data will not have to be manipulated through the database.
The phone number is saved as a varchar to allow users to save country codes (such as +45 for Denmark).
It is assumed that no address is longer than 256 characters.
The *guardian_of* relationship enables one profile to be guardian of several other profiles. This is because the children should not have the authority to administrate their own pictograms and settings, they rely on a guardian to handle all

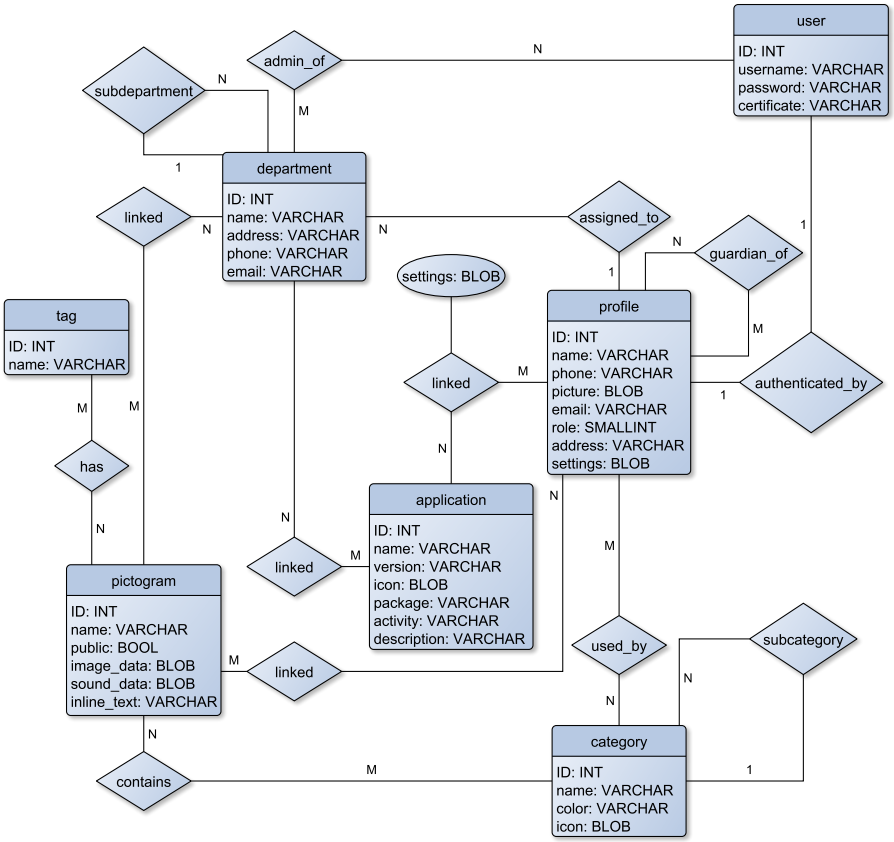

Figure 7.1: ER Diagram

of the customization in cooperation with them.
The settings of an application are saved in the relation to the profile in order to allow different users to customize the look of the application.

7.1.2 Application

The application entity holds information such as name, version, icon, package, activity and description. The package attribute signifies which Android package the application is included in and the activity attribute is the main activity of the application. Applications can be associated to both departments and profiles, depending on whether the application should be accessible to a single profile or all profiles associated with the department. As previously mentioned the settings are not stored as a part of the actual application, but in the relation to the profiles. This makes it easier customize the application for each individual profile.

7.1.3 Department

The department entity contains information such as name, address, phone and email. Departments have a number of profiles assigned to it i.e. employees and children.
The fact that applications and pictograms can be associated with departments signifies, that a department can allow all users affiliated with it to have access to specific applications and pictograms related to that department. This is done to ensure that all children, parents and employees can have access to common data, e.g. a picture of the department they are associated with.

7.1.4 Pictograms

The pictogram entity contains name, public, image data, sound data and some inline text. Pictograms can be a mixture of image, sound and text. The *public* attribute signifies whether the pictogram is public or restricted to a specific profile or department.
As of now, the image and sound data are supposed to be stored as blobs in the database, however in a production environment these data should be stored in files on the server and a system for uploading and downloading these should be implemented. This, however, is out of scope for the current project.

Categories

The pictograms can be arranged into categories each with its own name, colour and icon. An example could be pictograms of various food items being put in a category named food, which could have a subcategory called breakfast that could include items such as cereal and orange juice.

Tags

Each pictogram can have a series of tags. The tags can help categorize the pictograms.

7.1.5 User

In order to distinguish the various profiles and prevent unauthorized access to sensitive information in the system some kind of user authentication is needed. The user entity holds information such as username, password and certificate. The username and password will be used e.g. if an administrator needs to log into the system on a desktop computer. The certificate is a QR-Code that a guardian will scan with the tablet in order to log into the system. Each guardian will have their own personal QR-Code that they can use to administrate the children that they are responsible for.

7.2 Server API

When several different applications need access to data from the database, it is important to have a flexible API, especially when the applications need access for very different reasons, e.g. administration, games and pictures. Considering that several people have to work with it, and that new groups will be using the API later, simplicity is also a primary concern. Several decisions were made, all of which meant to support the simplicity and flexibility of the API.

7.2.1 Philosophy

Considering that the API should allow for all the necessary operations on a persistent storage system, it was decided that it should implement the Create, Read, Update, Delete (CRUD) actions, as these provide the minimal number of operations that allow for complete data manipulation. [14] As such, the CRUD actions comply with both demands of simplicity and flexibility. It was decided however, that the action for linking pictograms and applications to profiles and departments would be easier to use if it was placed in a separate action, in this case called “link”.

Another decision was made to keep the API design free of authorization elements. It is, for example, completely valid, for a normal user to request deletion of every profile in the database, and then it is up to the server to determine that they do not have the right to do this. This, again, keeps things simple, as it means that the same actions are available, whether the request comes from an administrative or user context.
Additionally, it does not create additional authentication work for the server implementations, since checking whether a user has a given right should be done each time in any case.

All or nothing

To keep things simple, calls should be kept all or nothing, that is, if it is not possible to complete the request fully, nothing should be done. An example of this could be a read call trying to retrieve details of several profiles, one of which is inaccessible to the authenticated user. In this case, the user receives no information, but an error message stating this. This is more important when updating or creating data in bulk, in which case it could be difficult for the client to know which parts of the request were completed, and which were not.

7.2.2 Data Serialization Format

When communicating on a network, it is important that the data can be serialized, that is, represented as a series of symbols. Two of the most well-known standards for serializing data between different platforms were considered, namely JSON and Extensible Markup Language (XML). The main difference, from an overall system point of view between these is that JSON is more light-weight, and arguably easier for humans to read, but has less ways of expressing data than XML.

This choice was discussed with multiple groups and in the end, JSON was selected. The readability of the format was valued highly, and it was argued that JSON would have enough power of expression for what this API would need. The fact that it is less verbose, and as such transferred faster over networks is also welcome.

7.2.3 Request Structure

For simplicity, the decision was made that requests of every type should have the same structure. The request should be in the form of a JSON object with three keys: auth, action and data. The auth key should contain another object, which contains the authentication information of the following request, whether logging in with a username and password, or using a QR certificate. The action key refers to a string, naming one of the CRUD (or link) actions. Finally, the data key, refers to an object where contents will differ depending on the action named. An example of requesting each profile that a user can access, can be seen in Listing 7.1.

```
1  {
2      "auth": {
3          "username": "john",
4          "password": "secret"
5      },
6      "action": "read",
7      "data": {
8          "type": "profile",
9          "view": "list",
10         "ids": null
11     }
12 }
```

Listing 7.1: Sample Request

7.2.4 Response Structure

As with the requests, all responses should have the same structure. The response is a JSON object containing four keys, in this case status, errors, session and data. The status key should contain a single string, describing what happened during the processing of the request, this could for example be “OK”, “SYNTAXERROR”, or “AUTHFAILED”. The errors key refers to an array of error messages, typically an empty array if everything went fine. The session key is only present if authentication was successful, and contains information about the authenticated user, like the ids of the user and profile if applicable, as well

as a session key for easy subsequent authentication. The data key can either refer to null, if an update or delete request was issued, or if there were errors in the request, a list of new ids if it was a create request, and finally the requested data for a read request. An example of a possible response to the request seen in Listing 7.1 can be seen in Listing 7.2

```
{
    "status": "OK",
    "errors": [],
    "data": [
        {
            "id": 521,
            "name": "John",
            "role": 1
        },
        {
            "id": 643,
            "name": "Mike",
            "role": 0
        },
        {
            "id": 1035,
            "name": "Joe",
            "role": 0
        }
    ]
}
```

Listing 7.2: Sample Response

7.2.5 Overview

The final API is simple, but powerful enough that it can do what it needs to do, partly due to the adherence to the CRUD principles. The full documentation for the API can be seen in Appendix A.

7.3 Server Application and Modules

For the server application itself, C++ was chosen as the implementation language for several reasons; it is extremely efficient while having access to high-level features such as classes, it has a large selection of well-documented third party libraries and all group members had experience with either the language or C, which it is based upon.
As it is well known that a modular code design, where each system references the others as little as possible, is desirable, designing the server application in such a way was an easy decision. It quickly became apparent that the server could logically be split into three modules: A connection module, a database module, and an API module, as well as a small amount of central code to bind these together. These modules should be created using a mix of imperative and object-oriented code, and will be described below.

7.3.1 Connection Module

The connection module will have the responsibility for any Internet communication. This module should include two classes: The first is a Connection class,

which represents an open connection, and allows data to be sent and received, as well as closing the connection. Instances of this class could represent both an incoming or an outgoing connection, though in the case of the server application, the incoming version will be the most relevant.
The second class is the Listener. This class should contain functionality to accept incoming connections on a given port, and make sure that an instance of the Connection class gets created to represent the client.
Apart from these two classes, the module also should contain a framework of unbound functions to set up and stop a Listener, while taking care of any multithreading needed, so that multiple clients can be handled concurrently. Additionally, the module needs to include several data structures, but these should not be manipulated by code outside the module, except for ServerInfo which may be referenced by the framework functions as it is used to represent a running server. A class diagram of the module's public interface can be seen in Figure 7.2.

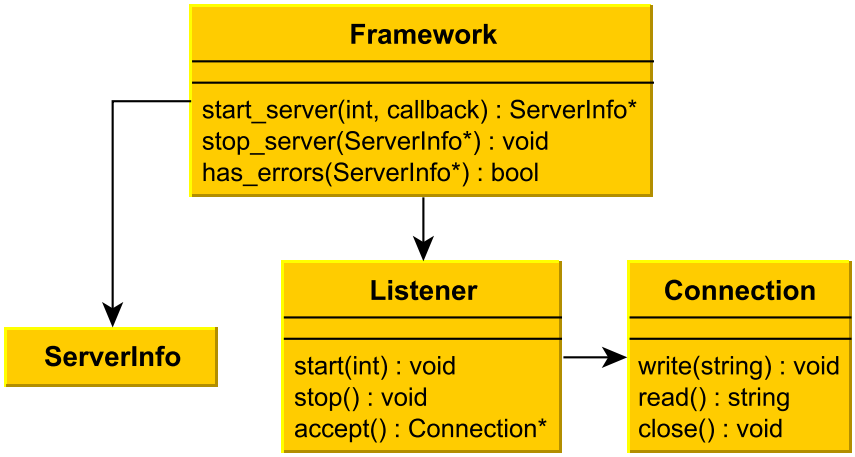

Figure 7.2: Connection Class Diagram

7.3.2 Database Module

The database module will be used to manage any connections to databases. While the actual SQL queries will be generated in the API module and passed to this one, this module will be responsible for sending them and returning any results to the caller. It will contain two classes.
Instances of the first, Database, will represent a set of settings for connecting and authenticating with a database. This instance can then be used to connect to, query and escape strings for that given database. The second class, QueryResult , is used to store whatever the database returns when queried. This result can then be read row by row. The rows should be stored in a common data type, where each field can be accessed in any order required. A container from the C++ Standard Template Library, like a map, would be a good candidate

for storing rows. The public interface of the database module can be seen in Figure 7.3.

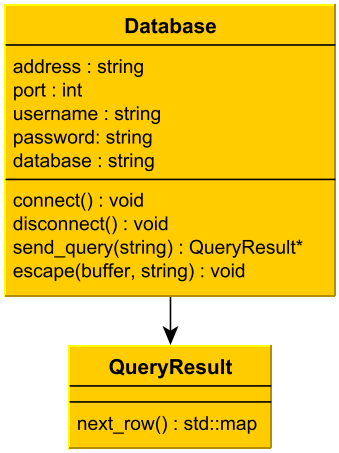

Figure 7.3: Database Class Diagram

7.3.3 API module

The API module will take care of handling each request made to the server. While it is where most of the important logic happens, it is also the module with the simplest public interface, with a single class having only one public function, as seen in Figure 7.5.
The private interface will be much more involved however. There will be a function for each action and each data type described in the API documentation (see Appendix A). This creates quite a few functions, as can be seen in Figure 7.4. Added to this will be functions to validate the structure of each type of request, as well as formatting the response and processing bulk data. For a comprehensive overview of these helper functions, see Chapter 8.

	User		Profile		Dept.		App.		Pict.		Cat.		Misc
	Li	De	Li	De	Li	De	Li	De	Li	De	Li	De	
Read	X	X	X	X	X	X	X	X	X	X	X	X	
Create	X		X		X		X		X		X		
Update	X		X		X		X		X		X		
Delete	X		X		X		X		X		X		
Link													X

Note: Li = list, De = details, X = An API call to implement

Figure 7.4: Necessary API Calls

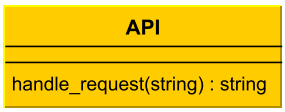

Figure 7.5: API Class Diagram

7.3.4 Overview

This setup results in three modules, where the only dependency will be from the API module to the database module, as shown in Figure 7.6, as the API module is responsible for executing SQL queries, and as such needs to reference a database. The use of callbacks in the connection library should prevent it from having any direct relation with the other modules, and while the database module could conceivably use the connection library to connect to a database, the protocols for this communication is already implemented in the library from MySQL, which makes it easier to use the connection functionality from said library instead.

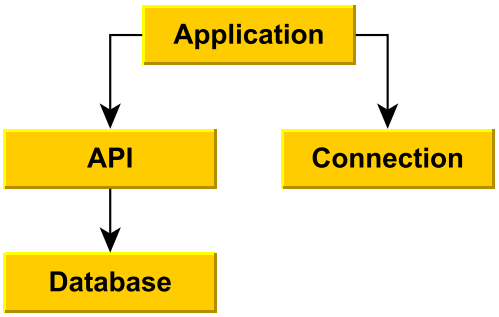

Figure 7.6: Module relations

7.4 Synchronization Design

The purpose of the synchronization is to maintain a local database on an Android device that can be kept in synchronization with the central database. The idea behind the synchronization is that many different Android devices should be able to access information about the same users. For this to work correctly, the data on different Android devices should be the same, otherwise the concept of a central database that stores all data will be moot. This means that the database on the Android device will include exact copies of relevant data that the central database includes, and as such the two databases must have the same tables. In addition to downloading data from the central database, an Android device will be able to create data itself. This data can then be uploaded to the central database to be kept in synchronization with many different devices.
The application that will synchronize the data between Android devices and the central database will run on the Android devices, and will thus be written in Java.

7.4.1 The Application

The application for synchronization, henceforth known as Puddle, is meant to be a replacement of the OasisLib application from previous years.

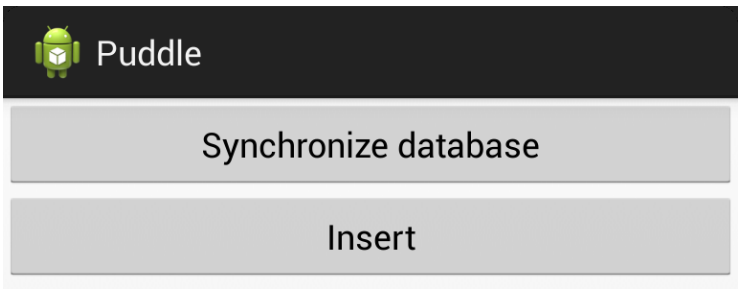

Figure 7.7: Overview of the Puddle Android application.

Figure 7.7 shows the Puddle application as it launches on an Android device. Puddle has two buttons, one for synchronizing with the central database, and one for inserting information into the local database. Puddle synchronizes manually with the central database, and needs to be launched manually. To synchronize, press the synchronize button, and Puddle will upload changes made to the local database, and then download from the central database. When the insert button is pressed, it inserts data into the database.

7.4.2 Creating a Local Database

Before it is possible to save to a local database on an Android device, it must first be created. Android only has native support for SQLite to create SQL databases, SQLite is therefore used for the local database in Puddle. The local database is not created until it is needed. This means that the local database will not be created until the synchronization with the central database is started.

The local database is not deleted, however, unless it is actively requested. On subsequent synchronizations there is no need to create the local database again.

7.4.3 Uploading and Downloading Changes

Changes made on an Android device should not be deleted when the two databases are synchronized. To avoid it, the updated data are first uploaded to the central database, before new data from the central database is downloaded to the Android device. This has the unfortunate side effect that if the same data was changed on both the central and local database between synchronizations, one has to take precedence over the other. It was decided that changes made on Android devices will always supersede changes made to the central database.
Before uploading changes made on the Android device, the changes must be found. To find these, a table has been added to the local database that does not exist in the central database. This table contains a timestamp with the time of the most recent synchronization with the central database. An additional attribute is added to all tables in the local database. This attribute also contains a timestamp, but this timestamp is updated when the row is inserted or changed. Every row that has a newer timestamp than the last synchronization is new or has been updated. These will be uploaded to the central database at the next synchronization. If there are no changes, this step will be skipped. Next, the data on the central database is downloaded to the Android device and its database is updated with this. Of course, only data that is newer than the timestamp of the last synchronization and is relevant for users of the device is downloaded.

Chapter 8

Implementation

In this chapter the implementation of the design of the modules, the API and the synchronization is described. It should be noted that there will be long listings in this chapter. This is to ease the understanding of the code for coming students building on top of this project.

8.1 SQL

The database is implemented in MySQL in accordance with the ER diagram. In Appendix B is a database schema showing the implementation.
On top of the database different views have been implemented to simplify the SQL needed to access data in the API calls. There are 8 views in total:

- profile_list , which computes a list of ids of the profile connected to the calling user, children this user is assigned guardian of, parents of these children, profiles the user is administrator of and created profiles.
- user_list which computes a list of ids and usernames of the caller's own user and users the caller is administrator of.
- department_list which computes a list of the ids and names of the department the user is connected to, departments the user is administrator of and sub-departments of the departments the user is administrator of.
- pictogram_list computes a list of ids, names, whether the calling user is the author and whether there is a direct link (a link not through department) of pictograms connected to the calling user's profile. The list includes the same info about pictograms linked to the department of the calling user, and the user's own created pictograms.
- application_list computes id, name and whether the calling user is the author of all applications connected to the calling user's profile, the calling user's department and created applications.
- application_details computes the settings as well as all the attributes on all applications connected to the calling user and the calling user's department.

- pictogram_extras computes the category and tags for all pictograms.
- category_list computes all ids, names and super category ids of categories connected to the calling user's profile.

As an example of the views implemented Listing 8.1 shows the view for application_list .

```
1  CREATE VIEW `application_list` AS
2      /* PROFILE */
3      SELECT `user`.`id` AS `user_id`, `application`.`id`, `
           application`.`name`, (0) `author` FROM
4          `user`
5          JOIN
6          `profile` ON `user`.`id`=`profile`.`user_id`
7          JOIN
8          `profile_application` ON `profile`.`id`=`
               profile_application`.`profile_id`
9          JOIN
10         `application` ON `profile_application`.`application_id`=`
               application`.`id`
11     UNION
12     /* DEPARTMENT */
13     SELECT `user`.`id` AS `user_id`, `application`.`id`, `
           application`.`name`, (0) `author` FROM
14         `user`
15         JOIN
16         `profile` ON `user`.`id`=`profile`.`user_id`
17         JOIN
18         `department` ON `profile`.`department_id`=`department`.`id`
19         JOIN
20         `department_application` ON `department`.`id`=`
               department_application`.`department_id`
21         JOIN
22         `application` ON `department_application`.`application_id
               `=`application`.`id`
23     UNION
24     /* CREATED applications */
25     SELECT `user`.`id` AS `user_id`, `application`.`id`, `
           application`.`name`, (1) `author` FROM
26         `user`
27         JOIN
28         `application` ON `user`.`id`=`application`.`author`;
```

Listing 8.1: Application List View

As can be seen, the view consists of the union of three different selects - one for the profile, one for department and one for created applications. To obtain the info about what applications are connected to a given profile and department, a series of joins are performed.
The structure is the same for all the views, albeit with differences in what is selected, the unions and what is joined.
These views are non-materialized views, i.e. they are not precomputed. The reason for this is that MySQL in itself does not provide materialized views. [4]

8.2 Database Module

The database module is a wrapper for functions from the MySQL Connector/C library and consists of two classes: a Database class responsible for connecting

to the database, escaping queries and sending them to the database, and a QueryResult, responsible for storing the result from the query and fetching rows, a type definition signifying a map of strings to strings.

8.2.1 Database Class

The Database class consists of a constructor for a database connection, which contains all the info needed to connect to a database and four public functions:

- send_query: Wraps the mysql_query function, which takes a connection pointer and a query and sends it to the database.
- connect_database: Wraps the mysql_real_connect functions, which uses the data given in the instantiation of the Database object and establishes a connection to the database.
- disconnect_database: Wraps the mysql_close function, which closes the connection.
- escape: Wraps mysql_real_escape_string, which escapes a string to prevent SQL injections.

The data given in the database constructor are saved, no matter how many times one may connect and disconnect with the same Database instance.

8.2.2 QueryResult Class

The QueryResult wraps the MYSQL_RES data type, which is what is returned from the database as the result of a query. The class has one public function, next_row, which fetches the next row in a result. It returns a row as an std::map.

8.3 Connection Module

The connection module is implemented using the Linux sockets and POSIX thread libraries. The two classes in the module, Connection and Listener, are mostly wrappers around socket functions, with various error handling and other convenience added. The framework deals with setting up threads. This functionality will be described in this section.

8.3.1 The Connection Class

The Connection class, as mentioned in Chapter 7 represents a TCP connection, either outgoing or incoming. As such it is implemented with two constructors, one taking no arguments which will initialize the instance so that it is ready to connect to a given host, as well as a constructor used by the Listener class that initializes the instance with the information needed to communicate with a client accepted by said Listener.
The class exposes three functions which are just wrappers around the Linux socket code, but includes error handling, namely connect_to_host, send and disconnect . The function for receiving, as seen in Listing 8.2 is a bit more involved.
After the initial setup, it blocks at line 14 with a call to poll until data is available. When data becomes available, it will enter a loop which reads from

the socket into a buffer, and places the contents of this buffer in a stringstream, a standard C++ class for building strings a section at a time. After reading it will check if more data is available from the socket, this time with a 100 millisecond time out, at line 31. It will continue reading until this poll times out. The reason that it is necessary to read the data this way, is the use of TCP. TCP is a stream socket, which means that it guarantees the arrival of data, but not that all data arrives at the same time. For this application, however, it is useless to have only part of a message, since a partial JSON object can not be decoded, so the function reads until it is reasonably sure all data has been received. In a production environment, the timeout may need to be increased.

```
1  char *Connection::receive()
2  {
3      if (!this->is_connected())
4      {
5          fprintf(stderr, "ERROR: Attempt to read from socket with no
               connection\n");
6          return NULL;
7      }
8
9      // Set up poll structure to listen for IN-data on stream
10     pollfd p;
11     p.fd = _connection_fd;
12     p.events = POLLIN;
13
14     poll(&p, 1, -1); // Wait for data
15
16     char *buffer = (char *)malloc(BUFFER_SIZE); // Chunk buffer
17     std::stringstream msg; // Message buffer
18
19     do
20     {
21         memset((void *)buffer, 0, BUFFER_SIZE);
22         int n = read(_connection_fd, buffer, BUFFER_SIZE - 1);
23         if (n < 0)
24         {
25             fprintf(stderr, "ERROR: Failed reading from socket.\n")
                   ;
26             break;
27         }
28         msg << buffer;
29         // See if more data is available, with 100ms timeout,
30         // this is necessary for stream (TCP) sockets.
31         poll(&p, 1, 100);
32     }
33     while (p.revents & POLLIN);
34
35     free(buffer);
36     std::string s = msg.str();
37
38     char *m = new char[s.size() + 1];
39     std::copy(s.begin(), s.end(), m);
40     return m;
41 }
```

Listing 8.2: Connection::Receive

8.3.2 The Listener Class

The Listener class is used to wait for clients and make sure that said clients can be handled by the application. It exposes three functions start, stop and accept_client . Each of these functions are direct wrappers of Linux sockets, except for the fact that accept_client automatically creates an instance of the Connection class to communicate with the client.

8.3.3 The Framework Functions

The main functionality of the framework is found in the public function run_server , as well as two functions that are not callable from outside the module: server_runner and client_runner, which serve as entry points for the different threads that will be started during execution. It is important to distinguish between the functions run_server and server_runner, and the similar names are a minor problem, considering that only run_server is visible at all outside the module.

Run Server

The run_server function, seen in Listing 8.3 starts by setting up a ServerInfo structure for the server. This structure contains several pieces of information that is useful for the framework to operate the server, and obtained at different points:

- The port the Listener should use. Provided as parameter to run_server.
- The callback to run when a client connects. Provided as a parameter to run_server.
- The thread the listener is allowed to block. Created in run_server on line 12.
- A mutex for controlling when run_server returns to the caller. Created in run_server on line 4.
- A pointer to the Listener instance for the server. Created in server_runner, explained later in this section.
- The error or success code for starting the Listener. Set in server_runner. explained later in this section.
- A flag for when to stop the server. Set when stop_server is called.

As this is being set up, a thread is started, with the server_runner as entry point and the ServerInfo instance as parameter at line 12. After this the function uses a mutex to block the calling thread, until the new thread has attempted to start the listener and the ServerInfo instance contains information about whether it was successful. This is done so the calling thread can check for errors immediately after run_server returns.

```
1  ServerInfo *run_server(unsigned int port, connection_callback
       callback)
2  {
3      ServerInfo *param = new ServerInfo(port, callback);
4      pthread_mutex_init(&(param->_lock), NULL);
5      if (pthread_mutex_lock(&(param->_lock)) != 0)
```

```
6       {
7           fprintf(stderr, "ERROR: Unexpected mutex error.\n");
8           return param;
9       }
10
11      // Start the server on a separate thread
12      if (pthread_create(&(param->_thread), NULL, server_runner, (
            void *)param) != 0)
13      {
14          fprintf(stderr, "ERROR: Failed to start client thread.\n");
15          pthread_mutex_unlock(&(param->_lock));
16          return param;
17      }
18      pthread_mutex_lock(&(param->_lock)); // Wait for success or
            error on Listener
19      pthread_mutex_unlock(&(param->_lock));
20
21      return param;
22  }
```

Listing 8.3: Framework - run_server

Server Runner

The server_runner is, as mentioned, the entry point of the thread that the server will run on. The code for this function can be seen in Listing 8.4. The first thing happening in this function is the creation and starting of a Listener instance, saving whether the Listener started successfully to the associated ServerInfo. After this is done, the mutex blocking the calling thread is released at line 11, allowing the main application to continue while the server is running.
If any errors has happened at this point, the thread will exit. If not, it will enter a loop, where it will attempt to accept clients until it is asked to stop by a call to stop_server. When looking at the code at line 24-29, between accepting the client and handling it, there is an additional check for whether the server has been asked to stop, that will break the loop if this is the case. The reason for this seemingly strange check is the blocking nature of accept_client. This block creates the necessity for stop_server to initiate a connection to the server, to ensure that it stops immediately. This check makes sure that particular connection is just discarded.
At this point, by line 33, the client will be handled. This is done by setting up a ClientInfo, a simple structure containing the desired callback, as well as a pointer to the Connection instance for communicating with the client. Then, a thread is started with client_runner as entry point and the ClientInfo as parameter.
After the accept loop is broken, the function stops the Listener, and exits the thread.

```
1   void *server_runner(void *param)
2   {
3       fprintf(stderr, "Running server\n");
4       ServerInfo *info = (ServerInfo *)param;
5
6       // Initialize the listener
7       info->_listener = new Listener();
8       info->_startcode = info->_listener->start(info->_port);
9
10      // Unlock calling thread now that the error code is ready
11      pthread_mutex_unlock(&(info->_lock));
```

```
12
13      if (server_has_errors(info))
14          pthread_exit(NULL);
15
16      // Accept clients until the server is stopped
17      while (!info->_stop)
18      {
19          // Attempt to accept a connection
20          Connection *connection = info->_listener->accept_client();
21          if (connection == NULL)
22              continue;
23
24          if (info->_stop)
25          {
26              // Do not handle connections after stop signal
27              connection->disconnect();
28              break;
29          }
30
31          // Attempt to start a thread for the client
32
33          ClientInfo *client = new ClientInfo(connection, info->
                _callback);
34          pthread_t client_thread;
35          if (pthread_create(&client_thread, NULL, client_runner, (
                void *)client) != 0)
36          {
37              fprintf(stderr, "ERROR: Failed to start client thread\n
                    ");
38              continue;
39          }
40          pthread_detach(client_thread);
41      }
42
43      fprintf(stderr, "Stopping server.\n");
44      info->_listener->stop();
45      pthread_exit(NULL);
46      return NULL;
47  }
```

Listing 8.4: Framework - server_runner

Client Runner

The client_runner function is very simple. It reads the callback from the ClientInfo instance and calls it with the Connection as parameter. After the callback returns, it cleans up and exits the thread.

8.4 Builder Functions

The so-called builder functions are implemented to facilitate ease of reading and writing in the API calls.

8.4.1 Fix

For all read calls to the API, some data from the database has to be altered to conform to the API standard either by renaming or fixing the type of the

data. In order for this to happen, two functions are implemented, fix_rename and fix_type, which as their names imply rename and change the type respectively.

8.4.2 Extractors

The API needs to extract data from JSON objects in order to build the queries to the database, and to prevent misuse of the queries some of these data need to be escaped and all need to be validated. This is done with three extractor functions:

- extract_string
- extract_int
- extract_bool.

In Listing 8.5 the extract_string function is shown.

```
1  int extract_string(char *buffer, Json::Value &object, const char *
       key, bool null, Database *escape_db)
2  {
3      if (object.isMember(key))
4      {
5          if (!object[key].isString()) return -1;
6          const char *raw_value = object[key].asCString();
7          char value[EXTRACT_SIZE];
8          memset(value, 0, EXTRACT_SIZE);
9
10         if (escape_db != NULL) escape_db->escape(value, raw_value);
11         else std::strncpy(value, raw_value, EXTRACT_SIZE - 1);
12
13         unsigned int length = std::min(EXTRACT_SIZE - 3u, (unsigned
               int)strlen(value));
14         std::strncpy(buffer + 1, value, length);
15         buffer[0] = '\'';
16         buffer[length + 1] = '\'';
17         buffer[length + 2] = '\0';
18     }
19     else
20     {
21         if (null) strncpy(buffer, "NULL", 5);
22         else return -1;
23     }
24     return 0;
25 }
```

Listing 8.5: Extract String

Each of the functions are similar in that they all validate that the given key is of the expected type (in Listing 8.5 this would be string) and if not they return an error. Apart from validating the type, extract_string also escapes the string using the database module, if a database is provided in the input.

8.4.3 Builders

These functions create a data type from another - e.g. building a JSON object from a row from the result of a database query, building an array from the result of a query, or build a string to fit with the SQL IN keyword. The builder functions are:

- build_object_from_row
- build_array_from_query
- build_simple_array_from_query
- build_in_string
- build_simple_int_vector_from_query
- build_simple_ind_map_from_query

In Listing 8.6 the code for the build_object_from_row function is shown.

```
1  int extract_string(char *buffer, Json::Value &object, const char *
       key, bool null, Database *escape_db)
2  {
3      if (object.isMember(key))
4      {
5          if (!object[key].isString()) return -1;
6          const char *raw_value = object[key].asCString();
7          char value[EXTRACT_SIZE];
8          memset(value, 0, EXTRACT_SIZE);
9
10         if (escape_db != NULL) escape_db->escape(value, raw_value);
11         else std::strncpy(value, raw_value, EXTRACT_SIZE - 1);
12
13         unsigned int length = std::min(EXTRACT_SIZE - 3u, (unsigned
               int)strlen(value));
14         std::strncpy(buffer + 1, value, length);
15         buffer[0] = '\'';
16         buffer[length + 1] = '\'';
17         buffer[length + 2] = '\0';
18     }
19     else
20     {
21         if (null) strncpy(buffer, "NULL", 5);
22         else return -1;
23     }
24     return 0;
25 }
```

Listing 8.6: Build Object from Row

This function is called when one wants to create a JSON object from the result of a query. It takes a row (the result of a query) and a fixture, which is a function pointer to a function which applies one or more of the fixes mentioned earlier.
The function loops through the row and directly translates it to a JSON object.

```
1  int extract_string(char *buffer, Json::Value &object, const char *
       key, bool null, Database *escape_db)
2  {
3      if (object.isMember(key))
4      {
5          if (!object[key].isString()) return -1;
6          const char *raw_value = object[key].asCString();
7          char value[EXTRACT_SIZE];
8          memset(value, 0, EXTRACT_SIZE);
9
10         if (escape_db != NULL) escape_db->escape(value, raw_value);
11         else std::strncpy(value, raw_value, EXTRACT_SIZE - 1);
```

```
12
13          unsigned int length = std::min(EXTRACT_SIZE - 3u, (unsigned
                int)strlen(value));
14          std::strncpy(buffer + 1, value, length);
15          buffer[0] = '\'';
16          buffer[length + 1] = '\'';
17          buffer[length + 2] = '\0';
18      }
19      else
20      {
21          if (null) strncpy(buffer, "NULL", 5);
22          else return -1;
23      }
24      return 0;
25  }
```

Listing 8.7: Build Array from Query

Listing 8.7 utilizes the database module to fetch a row from the result, and while there are more rows, calls the build_object_from_row and appends the JSON objects to a JSON array.

8.4.4 Validators

As the name implies, these functions validate that a given value (or set of values) exist in an array or a vector. The validator functions are:

- validate_array_vector
- validate_value_in_vector

validate_array_vector takes an array and a vector and loops through to determine if every value in the array exists in the vector.
validate_value_in_vector takes a single value and checks, if this value exists in the vector.
Both functions return boolean values of either true or false, depending on whether the value(s) were found or not.

8.5 API Calls

The API calls are very similar in structure, regardless of the data type and action they implement. They all start out by verifying that the data they have received is in accordance with what is expected and needed for the call to be performed, create the SQL statement to be executed, send it and prepare whatever return data is expected for the given call.

8.5.1 Read and delete calls

All read calls are first validated to ensure that they contain the required data, i.e. type, view and ids, and that the types are correct (e.g. ids must always be either null or an array). Afterwards it is determined whether the read call is of the list or details variety, and which data type is requested, and the call is performed.
The delete calls are almost exact copies of the read details calls, only the SQL-statements differ.

List

All calls requesting list data utilize the views implemented in the database for each data type. As an example, the read call for profile is shown in Listing 8.8.

```
1  Json::Value API::read_profile_list(Json::Value &data, int user,
       Json::Value &errors)
2  {
3      char query[API_BUFFER_SIZE];
4      snprintf(query, API_BUFFER_SIZE, "SELECT DISTINCT `id`, `name`,
           `role` FROM `profile_list` WHERE `user_id`=%d;", user);
5      QueryResult *result = _database->send_query(query);
6
7      Json::Value call_data = build_array_from_query(result,
           fix_profile_list);
8
9      delete result;
10     return call_data;
11 }
```

Listing 8.8: Read Profile List

As can be seen, the read list call for the profile data type, like all the others, receive the data, a user id requesting the data and a pointer to the errors array. A query is created and sent via the database module. The response from the database is then placed in an array and returned to the caller. For other read calls, the SQL statement to be executed (and subsequently the data extracted) differs. A different view will of course also be used.

Details

If the call is for a detailed view, the caller has to provide an array of ids for which details are needed.
As an example of what the details calls look like, the call for the data type profile is shown in Listing 8.9

```
1  Json::Value API::read_profile_details(Json::Value &data, int user,
       Json::Value &errors)
2  {
3      char query[API_BUFFER_SIZE];
4
5      snprintf(query, API_BUFFER_SIZE, "SELECT DISTINCT `id` FROM `
           profile_list` WHERE `user_id`=%d;", user);
6      QueryResult *result = _database->send_query(query);
7      std::vector<int> accessible =
           build_simple_int_vector_from_query(result, "id");
8      delete result;
9
10     if(validate_array_vector(data["ids"], accessible) == false)
11     {
12             errors.append(Json::Value("Invalid ID access"));
13             return Json::Value(Json::nullValue);
14     }
15     const std::string &st = build_in_string(data["ids"]);
16     snprintf(query, API_BUFFER_SIZE, "SELECT DISTINCT * FROM `
           profile` WHERE `id` IN (%s);", st.c_str());
17
18     result = _database->send_query(query);
19     Json::Value call_data = build_array_from_query(result,
           fix_profile_details);
20     delete result;
```

```
21
22      for (unsigned int i = 0; i < call_data.size(); i++)
23      {
24          snprintf(query, API_BUFFER_SIZE, "SELECT DISTINCT `child_id
                `FROM `guardian_of` WHERE `guardian_id`=%d;",
                call_data[i]["id"].asInt());
25          result = _database->send_query(query);
26          call_data[i]["guardian_of"] = build_simple_array_from_query
                (result, "child_id", V_INT);
27          delete result;
28      }
29
30      return call_data;
31  }
```

Listing 8.9: Read Profile Details

In lines 14-17 a list of profiles accessible to the calling user is compiled, and saved in a linked list. Each id requested is then checked against this list in line 19, and if the requested id is not in the list an error is added to the errors array, and the function returns.
If all the requested ids are accessible the ids are turned into a comma separated string in line 24, which is then inserted into the query in line 25. The query is sent to the database, and the response is added to the call_data object in lines 26 and 27.
Seeing as a call for profile details also requires info about which children a person is guardian of, this is fetched from the database and inserted into the call_data object in lines 31-37. Finally the data is returned to the caller.
For other data types, the verification will differ with respect to what views are used to compile accessible data, the SQL statement will of course also be different, with different data extracted, as well as different data returned to the caller.

8.5.2 Create and Update calls

These two actions are so similar that a single example will suffice to show the general idea. Like other API calls the data is first validated at top level, and the appropriate call for the data type is executed. In Listing 8.10 the create profile call is shown.

```
1   Json::Value API::create_profile(Json::Value &data, int user, Json::
        Value &errors)
2   {
3       char query[API_BUFFER_SIZE];
4
5       snprintf(query, API_BUFFER_SIZE, "SELECT DISTINCT `id` FROM `
            department_list` WHERE `user_id`=%d;", user);
6       QueryResult *result = _database->send_query(query);
7       std::vector<int> departments =
            build_simple_int_vector_from_query(result, "id");
8       delete result;
9
10      snprintf(query, API_BUFFER_SIZE, "SELECT DISTINCT `id` FROM `
            profile_list` WHERE `user_id`=%d AND `role`=2;", user);
11      result = _database->send_query(query);
12      std::vector<int> children = build_simple_int_vector_from_query(
            result, "id");
13      delete result;
```

```
14
15      for(unsigned int i = 0; i < data["values"].size(); i++)
16      {
17          Json::Value &object = data["values"][i];
18
19          int d = object["department"].asInt();
20          if (!validate_value_in_vector(d, departments))
21          {
22              errors.append(Json::Value("Illegal department"));
23              return Json::Value(Json::nullValue);
24          }
25
26          int r = object["role"].asInt();
27
28          if(r != 2)
29          {
30              if (!validate_array_vector(object["guardian_of"],
                    children))
31              {
32                  errors.append(Json::Value("Illegal profile(s)"));
33                  return Json::Value(Json::nullValue);
34              }
35          }
36          else
37          {
38              if (!object["guardian_of"].empty())
39              {
40                  errors.append(Json::Value("Child as guardian"));
41                  return Json::Value(Json::nullValue);
42              }
43          }
44      }
45
46      Json::Value call_data(Json::arrayValue);
47      std::vector<unsigned int> added_ids;
48      for(unsigned int i = 0; i < data["values"].size(); i++)
49      {
50          Json::Value &object = data["values"][i];
51
52          char name[EXTRACT_SIZE];
53          char email[EXTRACT_SIZE];
54          char address[EXTRACT_SIZE];
55          char phone[EXTRACT_SIZE];
56          char picture[EXTRACT_SIZE];
57          char settings[EXTRACT_SIZE];
58          int department;
59          int role;
60
61          int err = 0;
62          err += extract_string(name, object, "name", false);
63          err += extract_string(email, object, "email", true);
64          err += extract_string(address, object, "address", false);
65          err += extract_string(phone, object, "phone", true);
66          err += extract_string(picture, object, "picture", true);
67          err += extract_string(settings, object, "settings", true);
68          err += extract_int(&department, object, "department", false
                );
69          err += extract_int(&role, object, "role", false);
70          if (err != 0)
71          {
72              errors.append("Value error(s) in profile data object");
73              return Json::Value(Json::nullValue);
```

```
74          }
75
76          char query[API_BUFFER_SIZE];
77          snprintf(query, API_BUFFER_SIZE, "INSERT INTO `profile` (`
                name`, `email`, `department_id`, `role`, `address`, `
                phone`, `picture`, `settings`, `author`)"
78                                           "VALUES (%s, %s, %d, %d
                                                 , %s, %s, %s, %s, %
                                                d);", name, email,
                                                department, role,
                                                address, phone,
                                                picture, settings,
                                                user);
79          QueryResult *result = _database->send_query(query);
80          added_ids.push_back(_database->insert_id());
81          call_data.append(Json::Value(added_ids.back()));
82          delete result;
83
84          if(data.isMember("guardian_of"))
85          {
86              for (unsigned int i = 0; i < data["guardian_of"].size()
                    ; i++)
87              {
88                  int child = data["guardian_of"][i].asInt();
89                  int guardian = added_ids.back();
90                  snprintf(query, API_BUFFER_SIZE, "INSERT INTO `
                        guardian_of` (`guardian_id`, `child_id`)"
91                                                   "VALUES (%d, %d
                                                        );",
                                                        guardian,
                                                        child);
92                  QueryResult *result = _database->send_query(query);
93                  delete result;
94              }
95          }
96      }
97
98      return call_data;
99  }
```

Listing 8.10: Create Profile

In lines 5 to 13 the accessible departments and children are selected from the database to ensure that a user does not try to assign the profile to a department that they do not have the rights to add profiles to, make the profile a guardian of a child the creator does not have access to or make a child a guardian of someone. This check is performed in lines 15-44. In lines 50-74 the data to be inserted in the database is extracted and escaped with the extract_string and extract_int functions. The query is then formed and sent in lines 76-82, and the guardian_of table is updated in the same manner in lines 84-95. In line 98 the list of ids created in the database is returned.
For an update call what differs is the creation of and the keywords in the SQL statement. Seeing as a user may want to update some, but not all, data on a given data type (e.g. profile), all value objects could possibly be left null. Due to this, the data is only appended to the update SQL statement if there is any provided, which is validated with a simple check to see, if the first character in the object is a capital N, indicating that the data is a null value.

8.5.3 Link

Link is an action used to link and/or unlink profiles or departments with an application or a pictogram. Like all other calls, the request is first validated with regards to syntax and authentication of the user. If these succeed, the call proceeds to link and unlink as requested. The code is shown in Listing 8.11

```
1  Json::Value API::execute_link(Json::Value &data, int user, Json::
       Value &errors)
2  {
3      int profile = 0;
4      int department = 0;
5      char query[API_BUFFER_SIZE];
6      QueryResult *result;
7      row_t r;
8
9      if(data.isMember("profile"))
10     {
11         char query[API_BUFFER_SIZE];
12         snprintf(query, API_BUFFER_SIZE, "SELECT DISTINCT `id` FROM
               `profile_list` WHERE `user_id`=%d AND `update`=1;",
               user);
13         result = _database->send_query(query);
14         std::vector<int> accessible =
               build_simple_int_vector_from_query(result, "id");
15         delete result;
16
17         profile = data["profile"].asInt();
18         if (!validate_value_in_vector(profile, accessible))
19         {
20             errors.append(Json::Value("Illegal profile"));
21             return Json::Value(Json::nullValue);
22         }
23     }
24
25     if(data.isMember("department"))
26     {
27         snprintf(query, API_BUFFER_SIZE, "SELECT DISTINCT `id` FROM
               `department_list` WHERE `user_id`=%d AND `update`=1;",
               user);
28         result = _database->send_query(query);
29         std::vector<int> accessible =
               build_simple_int_vector_from_query(result, "id");
30         delete result;
31
32         department = data["department"].asInt();
33         if (!validate_value_in_vector(department, accessible))
34         {
35             errors.append(Json::Value("Illegal department"));
36             return Json::Value(Json::nullValue);
37         }
38     }
39
40     int actor_id = profile + department;
41     const char *actor = profile != 0 ? "profile" : "department";
42
43     if(data.isMember("link"))
44     {
45         for(unsigned int i = 0; i < data["link"].size(); i++)
46         {
47             Json::Value &object = data["link"][i];
48             char *settings = NULL;
```

```
49              char type[EXTRACT_SIZE];
50              _database->escape(type, object["type"].asCString());
51              int object_id = object["id"].asInt();
52
53              if(strcmp(object["type"].asCString(), "application") ==
                    0)
54              {
55                  if(object.isMember("settings") && profile != 0)
                        _database->escape(settings, object["settings"].
                        asCString());
56
57                  snprintf(query, API_BUFFER_SIZE, "SELECT `id` FROM
                        `application` WHERE `id`=%d", object_id);
58                  result = _database->send_query(query);
59                  r = result->next_row();
60                  delete result;
61
62                  if (r.empty())
63                  {
64                      errors.append(Json::Value("Illegal application"
                            ));
65                      return Json::Value(Json::nullValue);
66                  }
67
68              }
69              else
70              {
71                  snprintf(query, API_BUFFER_SIZE, "SELECT `id` FROM
                        `pictogram` WHERE `id`=%d", object_id);
72                  result = _database->send_query(query);
73                  r = result->next_row();
74                  delete result;
75
76                  if (r.empty())
77                  {
78                      errors.append(Json::Value("Illegal pictogram"))
                            ;
79                      return Json::Value(Json::nullValue);
80                  }
81              }
82
83              snprintf(query, API_BUFFER_SIZE, "SELECT * FROM `%s_%s`
                    WHERE `%s_id`=%d AND `%s_id`=%d;", actor, type,
                    actor, actor_id, type, object_id);
84              result = _database->send_query(query);
85              r = result->next_row();
86              delete result;
87
88              if (!r.empty())
89              {
90                  if (settings == NULL) continue;
91
92                  snprintf(query, API_BUFFER_SIZE, "UPDATE `%s_%s`
                        SET `settings`='%s' WHERE `%s_id`=%d AND `%s_id
                        `=%d;", actor, type, settings, actor, actor_id,
                         type, object_id);
93                  result = _database->send_query(query);
94                  delete result;
95              }
96              else
97              {
98                  if (settings == NULL)
```

```
99                      snprintf(query, API_BUFFER_SIZE, "INSERT INTO
                            `%s_%s` (`%s_id`, `%s_id`) VALUES (%d, %d);
                            ", actor, type, actor, type, actor_id,
                            object_id);
100                 else
101                     snprintf(query, API_BUFFER_SIZE, "INSERT INTO
                            `%s_%s` (`%s_id`, `%s_id`, `settings`)
                            VALUES (%d, %d, '%s');", actor, type, actor
                            , type, actor_id, object_id, settings);
102                 result = _database->send_query(query);
103                 delete result;
104             }
105         }
106     }
107
108     if(data.isMember("unlink"))
109     {
110         for(unsigned int i = 0; i < data["link"].size(); i++)
111         {
112             Json::Value &object = data["link"][i];
113             char type[EXTRACT_SIZE];
114             _database->escape(type, object["type"].asCString());
115             int object_id = object["id"].asInt();
116
117             snprintf(query, API_BUFFER_SIZE, "DELETE FROM `%s_%s`
                        WHERE `%s_id`=%d AND `%s_id`=%d;", actor, type,
                        actor, actor_id, type, object_id);
118             result = _database->send_query(query);
119             delete result;
120         }
121     }
122     return Json::Value(Json::nullValue);
123 }
```

Listing 8.11: Link and Unlink

In lines 9-38 accessible profiles and departments are compiled into lists and the requested ids are checked against these lists. In line 40 an actor_id is set - this is the id of either a profile or a department, and is computed as a simple addition of the two values profile and department, due to the fact that one of these values will be 0 by default (see line 3 and 4).
In line 41 an inline if-statement checks whether the profile integer is 0, and if it is not, it will set the actor string to be "profile". If it is, it will set the string to be "department".
Lines 43-106 are executed if one or more links are to be created, in which case it is first determined if the API is dealing with a request for linking of an application in line 53. If indeed it is an application, the data object is examined to determine if any settings need to be updated on a profile for this particular application. Should this be the case, these are escaped. This is done in line 55. A list of accessible applications for the caller is compiled in lines 57-60. The requested id is then validated against this list in line 62.
Should the request turn out to be for a pictogram, the ids are simply validated against a list of accessible ids in lines 71-80.
In lines 83-86 an SQL query for checking if a link already exists between a profile or department and application or pictogram. Should a link already exist, it is first determined if there are settings to update in line 90, and if so these are updated in lines 92-94. If a link does not already exist, it is created in lines 97-104.

Lines 108-121 is executed if there is a request for unlinking applications or pictograms. For each requested link to be deleted, an SQL statement is sent in lines 117-119.

8.6 Synchronization

The synchronization between the central database and a local database on an Android device is handled by the Puddle application. This application runs on the Android device, and is programmed in Java.

8.6.1 Main Activity

The main activity in Android is the activity that is started when the application is launched. The main activity in Puddle includes the user interface for the application as well as a reference to the database in the class PuddleApplication. MainActivity also defines the buttons for inserting data into the local database and triggering synchronization.

8.6.2 Connection

The connection from Puddle to the central database is handled by MySQL Connector/J [9]. Connector/J will given an IP address, username and password to connect to a remote MySQL database. From there it is possible to send SQL queries to manipulate the central MySQL database.

8.6.3 SQLite Database

The local database in Puddle is handled by the Database class. The Database class includes functions for creating the database, inserting and retrieving updated rows.
Creation of the SQLite is done by the DbHelper class seen in Listing 8.12, an extension of the SQLiteOpenHelper [10], which is an Android helper class designed to manage database creation and version management. Line 2 specifies the name of the final database file and line 3 specifies the version number of the database. onCreate functions in Android run when the class activity is first started. Lines 11-23 run when the database is first accessed. The createUser string on line 13 creates an SQL statement that is executed on line 23. The example in Listing 8.12 creates the user table in the database. Each individual table has an SQL statement created by the DbHelper.

```
1  private class DbHelper extends SQLiteOpenHelper {
2      public static final String DB_NAME = "giraflocal.sqlite";
3      public static final int DB_VERSION = 1;
4      private static final String TAG = "DbHelper";
5
6      public DbHelper() {
7          super(context, DB_NAME, null, DB_VERSION);
8      }
9
10     @Override
11     public void onCreate(SQLiteDatabase db) {
12         Log.i("Database onCreate", "Creating Database");
```

```
13          String createUser = String.format("CREATE TABLE IF NOT
                EXISTS " +
14                  "user" +
15                  " (id            INT(11)         NOT NULL," +
16                  "username        VARCHAR(64)     NOT NULL    UNIQUE,
                        " +
17                  "password        VARCHAR(256)    NULL," +
18                  "certificate     VARCHAR(512)    NULL        UNIQUE,
                        " +
19                  "timestamp  INT(14)," +
20                  "PRIMARY KEY (id)" +
21                  ");");
22
23          db.execSQL(createUser);
```

Listing 8.12: Creating the SQLite database

Android applications consist of activities that run when they are active. These activities can for example be the main screen started when the application is started or the settings menu. Because database and network calls cannot be made on the User Interface (UI) thread in Android, they need to be made from an AsyncTask [7] and not an activity. The class PuddleApplication extends the Android class Application and will, unlike activities, run the entire time that Puddle is running. PuddleApplication (Listing 8.13) contains an instance of the database created in the Database class, and makes it possible to access the local database from AsyncTasks. To instantiate PuddleApplication, the class has been added to the ¡application¿ tag in the file AndroidManifest.xml. The Manifest is the file that tells the Android device how to run the application.

```
1  public class PuddleApplication extends Application{
2
3      private Database db = null;
4
5      @Override
6      public void onCreate() {
7          super.onCreate();
8          setDatabase(new Database(this));
9      }
10
11     public Database getDatabase() {
12         return db;
13     }
14
15     public void setDatabase(Database db) {
16         this.db = db;
17     }
18 }
```

Listing 8.13: Creation of Database in PuddleApplication

8.6.4 Downloading From the Central Database

When pressing the synchronize button in Puddle, two things happen, the classes UploadTask and DownloadTask are called. DownloadTask connects to the central database, downloads all data and inserts it into the local database. The DownloadTask is an extension of the Android class AsyncTask. AsyncTask creates a separate thread for DownloadTask because as mentioned access to network and databases is not allowed on the UI thread in Android. Listing 8.14 shows the download

from the central database for the user table. Line 1 and 2 sets up a new connection to the database. Line 4 inserts a timestamp to the table last_sync in the database, this timestamp is also added to every row inserted from the central database. This is done to be able to upload new rows when the databases are synchronized again. Line 6 selects all rows from the user table in the central database. The while loop on lines 9-16 selects a single row and passes it to the corresponding insert function in the Database class. The other tables are handled in a similar manner.

```
1  Class.forName("com.mysql.jdbc.Driver").newInstance();
2  Connection con = DriverManager.getConnection(MainActivity.DB_URL,
       MainActivity.DB_USER, MainActivity.DB_PASS);
3  Statement st = con.createStatement();
4  db.insertSync(currentTime);
5  Log.i(TAG, "Starting download");
6
7  ResultSet downloadUser = st.executeQuery("SELECT * FROM user");
8  String userId, userUsername, userPassword, userCertificate;
9  while(downloadUser.next()) {
10     userId = downloadUser.getString("id");
11     userUsername = downloadUser.getString("username");
12     userPassword = downloadUser.getString("password");
13     userCertificate = downloadUser.getString("certificate");
14     Log.i(TAG, "downloadUser " + userId + userUsername +
           userPassword + userCertificate + currentTime);
15     db.insertUser(userId, userUsername, userPassword,
           userCertificate, currentTime);
16 }
17 downloadUser.close();
```

Listing 8.14: Connecting to and downloading from the central database

Listing 8.15 shows the insertion of data into the local database. Line 4 requests write access to the database, line 6 and 7 creates an SQL statement that is executed on line 9.

```
1  public void insertUser(String id, String username, String password,
       String certificate, String timestamp){
2      Log.i(TAG, "Inserting user " + id + " " + username + " " +
           password + " " + certificate + " " + timestamp);
3
4      SQLiteDatabase db = dbHelper.getWritableDatabase();
5
6      String sql = "INSERT OR REPLACE INTO user (id, username,
           password, certificate, timestamp) " +
7              "VALUES ('" + id + "', '" + username + "', '" +
                   password + "', '" + certificate + "', '" +
                   timestamp + "')";
8
9      db.execSQL(sql);
10 }
```

Listing 8.15: Inserting in the database

8.6.5 Uploading Updates to the Central Database

As with the DownloadTask, the UploadTask in Listing 8.16 begins by connecting to the central database on line 1 and 2. Line 4 gets updated rows from the getUpdated function shown in Listing 8.17. The while loop on lines 6-21 prepares each updated row by creating an SQL query, and executes the query on line 20.

```
1  Class.forName("com.mysql.jdbc.Driver").newInstance();
2  con = DriverManager.getConnection(MainActivity.DB_URL, MainActivity
       .DB_USER, MainActivity.DB_PASS);
3
4  Cursor uploadUpdatedUser = db.getUpdated("user");
5  String userId, userUsername, userPassword, userCertificate;
6  while(uploadUpdatedUser.moveToNext()) {
7      userId = uploadUpdatedUser.getString(0);
8      userUsername = uploadUpdatedUser.getString(1);
9      userPassword = uploadUpdatedUser.getString(2);
10     userCertificate = uploadUpdatedUser.getString(3);
11
12     user = con.prepareStatement("INSERT INTO user (id, username,
           password, certificate) VALUES(?, ?, ?, ?) " +
13         "ON DUPLICATE KEY UPDATE id=VALUES(id), username=VALUES(
               username), password=VALUES(password), certificate=
               VALUES(certificate)");
14     user.setString(1, userId);
15     user.setString(2, userUsername);
16     user.setString(3, userPassword);
17     user.setString(4, userCertificate);
18     user.addBatch();
19     Log.i("uploadUpdatedUser", "getting " + userId + userUsername +
            userPassword + userCertificate + user);
20     user.executeBatch();
21 }
```

Listing 8.16: Uploading changes to the central database

getUpdated() in Listing 8.17 first requests read access to the database on line 2. it then selects the timestamp of the last synchronization from the table last_sync on line 4-11. On line 15 every row that has a newer timestamp than the last synchronization is selected and is returned to the UploadTask on line 17. The getUpdated() function returns updated rows from one table at a time, the table needs to be specified when the function is called.

```
1  public Cursor getUpdated(String table) {
2      SQLiteDatabase db = dbHelper.getReadableDatabase();
3
4      String getLastSync = "SELECT * FROM last_sync;";
5
6      Cursor cursor = db.rawQuery(getLastSync, null);
7      String last_sync = "";
8      if(cursor != null) {
9          cursor.moveToFirst();
10         last_sync = cursor.getString(1);
11     }
12
13     Log.i("Get Updated", "Last sync = " + last_sync);
14
15     String sql = "SELECT * FROM " + table + " WHERE timestamp > " +
            last_sync + ";";
16
17     return db.rawQuery(sql, null);
18 }
```

Listing 8.17: Retrieving updated rows

8.6.6 Known Limitations of the Current Version

One limitation is that an update to the central database will be overwritten by an update from an Android device, even if the update to the central database is newer than the update on the Android device. This is a direct consequence of timestamps only being added on the local database. If timestamps were to be added to the central database as well, it would be possible to compare timestamps before inserting updates into the central database. This would make it possible to always keep the most recent change.
Another limitation is that the entire database is downloaded to each individual Android device. In a real world scenario, it would be smarter to limit the download to users and profiles associated with the device and download data for new users as needed. The current solution does not scale very well, and would cause the local database to grow unnecessarily large if the system were to be widely used. This problem could be avoided in several ways, e.g. if the central databases included timestamps on all additions and updates. This way, Puddle could synchronize only the changes newer than last synchronization.
A third limitation is that the local SQLite is unencrypted, and can be accessed by anyone who wishes to. This is of course a security risk if the system were to be used in a production environment, because anyone can get access to sensitive user data.

Chapter 9

Test

In this chapter the tests performed during development are described

9.1 Unit Tests

Unit tests have been written for all modules of the server using the BOOST Unit Test framework.
The unit tests for the modules test that everything gets initialized properly, and that the functionality handles both expected and unexpected input adequately, thus at least a test that should fail and one that should pass is written for each functionality included in the final product. An example of a BOOST unit test for the connection module can be found in Appendix C.
Furthermore the unit tests, specifically those of the API module, have been the basis of the integration tests of the API design. All the functionality is tested via unit tests to ensure that they conform to the design.
All in all, the group wrote 3 test suites, one for each module, containing a total of 11 test cases with 110 tests.
The Java implementations were not unit tested due to time constraints that did not allow the group to acquaint themselves with another unit testing framework. They were however tested in implementation, and given correct input they provide the expected functionality.

9.2 Acceptance Test

For acceptance testing we relied on the Admin group in the multiproject. They used the API and reported back with errors and functionality that they either had trouble understanding or that did not do what it was supposed to do.
Throughout their use they reported back through e-mail, phone and written notes, and here is a list of some of the bugs and improvements suggested by them:

- In the documentation, it should be clarified that files should be base 64 encoded to prevent escaping errors. This was fixed.
- The initial file-size limit of 2048 bytes, an arbitrary number set during implementation for testing purposes, was too small to handle 400x400 pixel

JPEG-files. The Admin group compiled the source code and empirically tested various limits on a local machine. They reported back, that a 2MB file-size limit worked. This limit was subsequently changed in the code to adhere to their results.

- There were problems with large files, but files of size 12.5kB and under worked. This was not clarified further by the test group and thus could not be fixed.
- There is currently no way to see a child's guardians without logging in. This was discussed with the group, and it was determined that this was indeed what should happen.
- Currently some create calls return a profile object error where other errors should be returned. This should of course be fixed, but was reported too late in the process for the group to be able to implement the fix for this.
- Some update calls return with success, but nothing in the database is updated. The group was unable to reproduce this error, and thus unable to fix it within the deadline. The problem was added to the list of known bugs found in Subsection 11.3.1.

Overall, the API in its current non-production form, fulfilled their basic needs and expectations after reviewing the design and documentation.

Chapter 10

User Manual

In this chapter, the instructions for installation of the current version of the project is explained.

10.1 Installation Instructions

Below is the install instructions for the server application and Puddle.

10.1.1 Hardcoded Information

It should be noted, that the IP, port and name of the database is hardcoded into the constructor for the API class in the serverside code. In Puddle the IP address, name of the central database, username and password for the central database, as well as insertion of information when pressing the Insert button is hardcoded. For Puddle are hardcoded information is found in the MainActivity.

10.1.2 Prerequisites for the server application installation

The system running the serverside application is expected to run on a Linux system with MySQL installed.

- MySQL Connector/C
  - Download from [8]
  - Place headers in usr/local/include
  - Place libs (libmysql.so etc) in usr/local/lib
- JsonCPP
  - Install instructions found at [12]
  - Place headers in usr/local/include
  - Place libs (renamed to libjson.so, libjson.a) in usr/local/include
- Add line /usr/local/lib to etc/ld.so.conf and run sudo ldconfig
- Unit tests require BOOST to be installed.[2]

10.1.3 Building the Program

- Download the code from [13].
- In the terminal cd to source/server
- run make all
- run ./serverapp

The server is now running. Type stop to exit.

10.1.4 Running Unit Tests

- In the terminal cd to source/server
- run make test
- run ./test_connection, ./test_database or ./test_api.

The database unit test requires a database called giraf accessible by user giraf@localhost identified by 123456.
Connection unit test requires permission to open listening sockets and requires TCP ports 1238 and 1239 to be free.

10.1.5 Prerequisites for the Puddle Android application

- Android SDK.
- Development IDE. Eclipse has been used, but IntelliJ IDEA and Android Studio should work.
- Java Development Kit.
- Connector/J library. Included in the library folder of the project. Can also be downloaded from [9].

10.1.6 Building Puddle Android Application

Import the project into your IDE, and build it. Importing in Eclipse can be done by pressing File -¿ Import -¿ Existing Projects into Workspace -¿ Select root directory and navigate to the Puddle folder.

10.1.7 Running the Puddle Android Application

Puddle requires Android version 3.0 or higher to run.

10.1.8 License

The report content is freely available, but publication (with source), only with credit to the authors.
GIRAF Database's source code is released under the GPLv3 [3] open source license. This means that you are free to inspect the source code at any time or contribute to the project yourself.

Online Availability

The installation guide and the latest, commented version of the code can also be found online [13].

Chapter 11

Reflection

In this chapter we will conclude the report and reflect on the project and working in a multiproject. We will also give a status of the project and a list of future work.

11.1 Conclusion

We have set up a central server with a database containing the information needed in the GIRAF system. We have also implemented a local database, however this is not accessible by applications on an Android device, due to the fact that this would require Puddle to be a library, which due to time constraints had to be left as future work. We have created an API to facilitate access and CRUD actions on the central database and tested the implemented functionality with unit tests. We have made basic two-way synchronization between a local database and the central server. There is still need for further development, especially on the synchronization.
We have used SCRUM for the project development.

11.2 Project Status

Currently, the API is implemented and tested with basic functionality and there is a central database containing all information needed to the best of our knowledge. There is a local database, however, as mentioned, it is not accessible by applications at this moment. There is basic two-way synchronization between the central database and the local database. Everything is currently unencrypted. There are still some problems in the code which need to be fixed.

11.3 Future Work

This section is divided into two lists, one containing known issues in the project, which should be fixed when the project is further developed and one suggesting new functionality to add to the project.

11.3.1 Known Issues

- On some update calls the server returns an OK, but no SQL is executed.
- If a user can access several profiles there is no way to see which settings belong to which profile for any given application.
- SQL errors can crash the server because null-errors in some query results go unhandled.
- There is no way to make a user department administrator without directly accessing the database.
- There are no SQL transactions in the API which means race conditions are a risk.

11.3.2 New Functionality

- Encryption and hashing should be added to the database and the connections.
- It should be possible to create a copy of a pictogram if an update is made to one that has multiple links.
- A search function should be implemented to allow the public setting on pictograms to let the pictogram show up in searches.
- The synchronization should be updated to have support for selective download from the database.
- A library for Android applications should be made to allow them to access the local database.
- The local database should not be allowed to exceed a certain size.
- Synchronization should be automatic instead of manual.
- A thread pool should be implemented to prevent thread overflow on the central server.
- Database information should not be hardcoded.
- Sessions should be implemented. Currently the sessions are not used for authentication, which was part of their intended purpose.
- Currently the views only accommodate selecting by user id, not by profile id. This means that when a guardian is logged in, even if the tablet is switched to the child's profile (which is a projected functionality of the GIRAF system), all pictograms, applications etc. that the guardian can access will also be accessible to the child. This should be fixed by adding a profile id column to the different views where users need it.

11.4 Project Evaluation

During the semester we had an excellent supervisor, which helped us keep the project on track.
The primary goal of the project was initially synchronization between two databases, a local tablet-database and a central server, but this was severely inhibited by the fact that the central database was virtually non-existent. This meant that we had to spend time implementing a server and a communication API before the expected project, i.e. the synchronization could be started. This led to a very basic synchronization functionality, more a proof of concept than anything else. The project would have benefited from an early prototype of the server application and the synchronization application being distributed among the groups in the multiproject, to see if there were errors or misconceptions in the design. However, due to the fact that a vast majority of the functionality had to be implemented to achieve a working prototype, there was no time to do this.
The workflow was heavily influenced by the fact that this was a multiproject, which meant that the sprints were global. This sometimes clashed with the desired workflow of our project.
The multiproject communication was quite good, with meetings every week and good communication between the different subprojects relying on each other. It would have been good to have the common vision ready as early as possible.
The common development method was SCRUM, which for the multiproject worked. The sprints were generally of adequate length, considering the time frame of the multiproject, however a few sprints were too short and could have been merged.
The committees worked very well, especially at the beginning, in unifying the overall design, stories etc.

11.4.1 Recommendations for next year

- Make sure the basis for each project proposal is done, i.e. ensure that the Launcher, the databases and everything is finished before building on top of them.
- Make sure everyone knows what is implemented.
- Have a common vision early on. Ensure that everyone knows exactly what it is.
- Committees are a great way to let all groups in on important decisions without involving every single person.
- Communicate. Have weekly meetings, visit each other's group rooms, send e-mails.
- Socialize.

Part III

Appendices

Appendix A

API Documentation

Wasteland API Documentation

API Request

Any requests to the Wasteland server should be a single JSON object in the following format:

```
{
  "auth": request_auth,
  "action": request_action,
  "data": request_data
}
```

Request Authentication

`request_auth` is a JSON object containing authentication information of the requester. Depending on the method of request, the content will vary.
Authentication as an administrator:

```
{
  "username": STRING,
  "password": STRING
}
```

Authentication using a QR ID-card:

```
{
  "certificate": STRING
}
```

Authentication using a session code:

```
{
  "session": STRING
}
```

Request Action

`request_action` is a string describing what the request wants to accomplish. Possible values are:

`"create" OR "read" OR "update" OR "delete" OR "link" OR null`

A null value signifies that the request does not want to access the database, but only check if authentication was successful (eg. to get a session code).

Request Data

`request_data` is an object describing the specifics of the request. The possible contents of this varies depending on the value of

`request_action`

Create

```
{
  "type": data_type,
  "values": [value_object, ...]
}
```

Read

```
{
  "type": data_type,
  "view": "list" OR "details",
  "ids": [INT, ...] OR null
}
```

The "view" value determines what should be read. A list of accessible objects to the requesting user, or details about specific objects
"ids" should be null if "view" is "list". If view is "details" it should be an array of integer ids of the objects required.

Update

```
{
  "type": data_type
  "values":
  [
    {
      "id": INT,
      "value": value_object
    },
    ...
  ]
}
```

Link

```
{
  "profile": INT,
  "department": INT,
  "link":
  [
    {
      "type": "pictogram" OR "application",
      "id": INT,
      "settings": STRING
    },
    ...
  ],
  "unlink":
  [
    {
      "type": "pictogram" OR "application",
      "id": INT,
    },
    ...
  ]
}
```

Exactly one of `department` or `profile` should be set, not both.

`settings` is only used if `profile` is set, and `type` is `"application"`

Delete

```
{
  "type": data_type,
  "ids": [INT, ...]
}
```

API Response

The Wasteland server responds to any request with a single JSON object.

```
{
  "status": status_code,
  "errors": [STRING, ...],
  "data": response_data,
  "session": session_info
}
```

Status code

`status_code` is a string telling how the request went. If unnsuccessful, more details will be written in the `errors` array. Possible values are:

`"OK"` Everything went fine.

`"SYNTAXERROR"` The request did not conform to the json syntax

`"BADREQUEST"` Keys or types in the request was wrong.

`"AUTHFAILED"` The authentification details were wrong.

"ACCESSDENIED" The user did not have access to the action or ids requested, or requested ids were not found.

Response Data

The type and value of data depends on the request_action of the request, and whether any errors were found.

Errors were found

null

Actions delete, update and link

null

Action create

[INT, ...] A list of ids of the objects created.

Action read

- If view is details: [value_object, ...]
- If view is list: [light_value_object, ...]

Session Info

session_info is an object containing information about the currently authenticated user and session. It looks like this:

```
{
  "user": INT,
  "profile": INT OR null,
  "session": STRING (NOT YET IMPLEMENTED)
}
```

If the request was not successful, this object will be replaced by a null-value.

Data Types

All image and sound files are expected to be base 64 encoded.
Depending on what is being accessed, different parameters are available.
Updating some fields might have different access rights. (TODO: describe clearly)

OPTIONAL means that create calls does not need to provide this value

RESPONSE ONLY means that create and update calls should not provide this value, but read calls will return it

REQUEST ONLY means the value can be updated and created, but never read.

CONSTANT < means that the value can only be changed on create

"light_value_object" is always response only, in list views

Users

"type": "user"

```
"value_object": {
  "id": RESPONSE ONLY INT,
  "profile": INT,
  "username": CONSTANT STRING,
  "profile": CONSTANT INT,
  "password": REQUEST ONLY OPTIONAL STRING,
  "certificate": REQUEST ONLY OPTIONAL STRING
}
```

Note: At least one of password or certificate must be defined.

```
"light_value_object": {
  "id": INT,
  "username": STRING
}
```

Profiles

```
"type": "profile"
```

```
"value_object": {
  "id": RESPONSE ONLY INT,
  "name": STRING,
  "email": OPTIONAL STRING,
  "department": INT,
  "user": RESPONSE ONLY INT,
  "role": INT,
  "guardian_of": OPTIONAL [INT, ...],
  "address": STRING,
  "phone": OPTIONAL STRING,
  "picture": OPTIONAL STRING,
  "settings": OPTIONAL STRING
}
```

```
"light_value_object": {
  "id": INT,
  "name": STRING,
  "role": INT
}
```

Departments

```
"type": "department"
```

```
"value_object": {
  "id": RESPONSE ONLY INT,
  "name": STRING,
  "address": STRING,
  "phone": STRING,
  "email": STRING,
  "subdepartments": OPTIONAL RESPONSE ONLY [INT, ...],
  "topdepartment": INT
}
```

```
"light_value_object": {
  "id": INT,
  "name": STRING
}
```

Pictograms

```
"type": "pictogram"
```

```
"value_object": {
  "id": RESPONSE ONLY INT,
  "name": STRING,
  "public": BOOL,
  "image": OPTIONAL STRING,
  "sound": OPTIONAL STRING,
  "text": OPTIONAL STRING,
  "categories": OPTIONAL [STRING, ...],
  "tags": OPTIONAL [STRING, ...]
}
```

```
"light_value_object": {
  "id": INT,
  "name": STRING,
  "categories": [STRING, ...],
  "tags": [STRING, ...]
}
```

Applications

```
"type": "application"
```

```
"value_object": {
  "id": RESPONSE ONLY INT,
  "name": CONSTANT STRING,
  "version": STRING,
  "icon": STRING,
  "package": STRING,
  "activity": STRING,
  "settings": OPTIONAL STRING,
  "description": OPTIONAL STRING
}
```

```
"light_value_object": {
  "id": INT,
  "name": STRING
}
```

Categories

```
"type": "category"
```

```
"value_object": {
  "id": RESPONSE ONLY INT,
  "name": STRING,
  "colour": STRING,
  "icon": OPTIONAL STRING,
  "topcategory": OPTIONAL INT
}
```

```
"light_value_object": {
  "id": INT,
  "name": STRING
  "topcategory": INT
}
```

Appendix B

Database Schema

A → signifies a foreign key, and the underlined attribute signifies a primary key.
NULL means that the attribute can be null.

profile		
id	INT(11)	
name	VARCHAR(64)	
phone	VARCHAR(11)	NULL
picture	BLOB	NULL
email	VARCHAR(64)	NULL
role	SMALLINT	
address	VARCHAR(256)	
settings	BLOB	NULL
user_id →	INT(11)	
department_id →	INT(11)	
author →	INT(11)	NULL

user		
id	INT(11)	
username	VARCHAR(64)	
password	VARCHAR(256)	NULL
certificate	VARCHAR(512)	NULL

department		
id	INT(11)	
name	VARCHAR(64)	
phone	VARCHAR(11)	
picture	BLOB	NULL
email	VARCHAR(64)	
role	SMALLINT	
address	VARCHAR(256)	
super_department_id →	INT(11)	NULL
author →	INT(11)	NULL

pictogram		
id	INT(11)	
name	VARCHAR(64)	
public	TINYINT	
image_data	BLOB	NULL
sound_data	BLOB	NULL
inline_text	VARCHAR(64)	
author →	INT(11)	NULL

tag		
id	INT(11)	
name	VARCHAR(64)	

category		
id	INT(11)	
name	VARCHAR(64)	
colour	VARCHAR(11)	
icon	BLOB	NULL
super_category_id →	INT(11)	NULL

application		
id	INT(11)	
name	VARCHAR(64)	
version	VARCHAR(32)	
icon	BLOB	
package	VARCHAR(256)	
activity	VARCHAR(64)	
description	VARCHAR(1024)	
author →	INT(11)	NULL

admin_of		
user_id →	INT(11)	
department_id →	INT(11)	

profile_pictogram		
profile_id →	INT(11)	
pictogram_id →	INT(11)	

department_application		
department_id →	INT(11)	
application_id →	INT(11)	

profile_application		
profile_id →	INT(11)	
pictogram_id →	INT(11)	
settings	BLOB	NULL

pictogram_tag		
pictogram_id →	INT(11)	
tag_id →	INT(11)	

pictogram_category		
pictogram_id →	INT(11)	
category_id →	INT(11)	

guardian_of		
guardian_id →	INT(11)	
child_id →	INT(11)	

Appendix C

Unit Test Example

```
1  #define BOOST_TEST_MODULE connection
2  #include <boost/test/included/unit_test.hpp>
3
4  #include "framework.h"
5
6  BOOST_AUTO_TEST_SUITE(connection_lib)
7
8  BOOST_AUTO_TEST_CASE( base )
9  {
10     BOOST_CHECK( BUFFER_SIZE >= 2 );
11 }
12
13 BOOST_AUTO_TEST_CASE( classes )
14 {
15     Connection c;
16     BOOST_CHECK_EQUAL( c.is_connected(), false ); // Initialization
17     fprintf(stderr, "Unit test: Expecting logged error.\n ");
18     BOOST_CHECK_EQUAL( c.connect_to_host("127.0.0.1", 1238), 1); //
           Connection should fail
19     fprintf(stderr, "Unit test: Expecting logged error.\n ");
20     BOOST_CHECK_EQUAL( c.send("Hello World!"), 1); // Sending
           should fail, error should be printed
21     Listener l;
22     BOOST_CHECK_EQUAL( l.start(1238), 0 ); // Listening should
           succeed
23     BOOST_CHECK_EQUAL( c.connect_to_host("127.0.0.1", 1238), 0); //
           Connection should succeed
24     Connection *c2 = l.accept_client();
25     BOOST_CHECK_EQUAL( c.send("Hello World!"), 0); // Sending
           should succeed
26     BOOST_CHECK_EQUAL( strcmp("Hello World!", c2->receive()), 0);
           // Should be same message
27     c2->disconnect();
28     c.disconnect();
29     BOOST_CHECK_EQUAL( c.is_connected(), false ); // Disconnected
30 }
31
32 int test(Connection *c)
33 {
34     c->send("Hello World!");
35     c->receive();
36     c->disconnect();
37     return 0;
```

```
38  }
39
40  BOOST_AUTO_TEST_CASE( framework )
41  {
42
43      ServerInfo *i = run_server(1239, test);
44      BOOST_CHECK_EQUAL(server_has_errors(i), false); // No errors
            should be detected
45      Connection c;
46      BOOST_CHECK_EQUAL( c.connect_to_host("127.0.0.1", 1239), 0); //
            Connection should succeed
47      BOOST_CHECK_EQUAL( strcmp("Hello World!", c.receive()), 0); //
            Should be same message
48      BOOST_CHECK_EQUAL( c.send("Hello World!"), 0); // Sending
            should succeed
49      c.disconnect();
50      stop_server(i);
51  }
52
53  BOOST_AUTO_TEST_SUITE_END()
```

Bibliography

[1] American Accreditation HealthCare Commision. Autism, May 2012. http://www.ncbi.nlm.nih.gov/pubmedhealth/PMH0002494/.

[2] Boost Community. Boost, 2013. URL http://www.boost.org/.

[3] Free Software Foundation. Gnu general public license, 2007. URL https://github.com/Zucka/girafAdmin/blob/master/LICENSE.md.

[4] FromDual. Materialized views with mysql, 2013. URL http://www.fromdual.com/mysql-materialized-views.

[5] Robert Hundt. Loop recognition in c++/java/go/scala, 2011. URL https://days2011.scala-lang.org/sites/days2011/files/ws3-1-Hundt.pdf.

[6] Craig Larman. *Agile and iterative development: a manager's guide*. Pearson Education, Inc., 2004. ISBN 0-13-111155-8.

[7] MySQL. Asynctask documentation, 2013. URL http://developer.android.com/reference/android/os/AsyncTask.html.

[8] MySQL. Download connector/c, 2013. URL http://dev.mysql.com/downloads/connector/c/.

[9] MySQL. Download connector/j, 2013. URL http://dev.mysql.com/downloads/connector/j/.

[10] MySQL. Sqliteopenhelper documentation, 2013. URL http://developer.android.com/reference/android/database/sqlite/SQLiteOpenHelper.html.

[11] Pyramid Droup Management Services. Picture your student learning. Website, Visited 21.03.2013. http://pecs.com.

[12] StackOverflow. Building jsoncpp (linux) - and instruction for us mere mortals, 2011. URL http://stackoverflow.com/questions/4628922/building-jsoncpp-linux-an-instruction-for-us-mere-mortals.

[13] SW603F13. Installation guide, 2013. URL https://github.com/Ezphares/giraf_database/blob/master/README.md.

[14] Wikipedia. Crud, 2013. URL http://en.wikipedia.org/wiki/CRUD.