# Make Giraf Stable Again

Web API, Database, and Administration Panel

**Group:**
SW613F18

**Supervisor:**
Ulrik Mathias Nyman

June 13, 2018

AALBORG UNIVERSITY
STUDENT REPORT

**Department of Computer Science**
Aalborg University
http://cs.aau.dk

**Title:**
Make Giraf Stable Again

**Theme:**
Web API, Database, and Administration Panel

**Project Period:**
Spring 2018

**Group:**
SW613F18

**Participants:**
Morten Rask Andersen
Asger Horn Brorholt
Anton Christensen
Christian Mønsted Grünberg
Mathias Ibsen
Mathias Pihl

**Supervisor:**
Ulrik Mathias Nyman

**Pages:**
89

**Date of Completion:**
June 13, 2018

**Number of Copies:**
1

**Abstract:**

Giraf (Graphical Interface Resources for Autistic Folk) is a collection of applications aimed at helping people diagnosed with autism spectrum disorder. The WeekPlanner App is a part of the Giraf application environment and helps to plan and visualise activities in a week schedule. The focus of this project is to continue the development of the WeekPlanner in a multi-project with five other groups. Our responsibility in this project is to develop the backend API and database to support the functionality of the app. In parallel with this, we are assigned the task of developing a new Giraf Administration Panel for administrating departments and users of Giraf. In four sprints, we plan, design, and develop parts of the API to support the user stories in each sprint. This includes increasing the stability and functionality of the API by improving and adding unit tests, integration tests, API documentation, bug fixing, refactoring, and new features. The project resulted in three stable releases of the Week Planner application, an administrative web page, and a custom development method called IWWP.

Morten R.A.

Morten Rask Andersen
mran15@student.aau.dk

AsgerHB

Asger Horn Brorholt
abrorh15@student.aau.dk

Anton Christensen

Anton Christensen
achri15@student.aau.dk

Christian Grünberg

Christian Mønsted Grünberg
cgranb15@student.aau.dk

[illegible]

Mathias Ibsen
mibsen15@student.aau.dk

Mathias pihl

Mathias Rohde Pihl
mpihl15@student.aau.dk

# Preface

This report documents the work done by group sw613f18 during the sixth semester at Aalborg University. The project involves five other groups, each responsible for a part of the Giraf multi-project. The project has spanned from February to May 2018 and has been supervised by Ulrik Mathias Nyman.

## Reading Guide

The illustrations and figures used throughout this report have been made by group sw613f18 unless otherwise stated. Excluded sections of code have been replaced with an ellipsis ([. . . ]). Citations will be made using the number notation where the number refers to a source in the bibliography.

Advice for future developers of the Giraf project, more specifically next year's students, can be seen in chapter 10 and can be skipped if the reader does not find this relevant.

We reference the wiki on Phriction several times. If this is unavailable for future semesters, a PDF-version is available in the electronic appendix, in the file `Wiki.pdf`.

## Source Code

The entire code base for the last release of the API, including code for the admin panel can be found in the Git repository for the API at `https://gitlab.giraf.cs.aau.dk/Server/web-api`.

## Collaboration

Section 3.2 and 7.2.3 has been written together with group sw610f18 and sw609f18, respectively. The documentation for next years groups which is described in section 7.5 has been coordinated among all Giraf groups.

## Abbreviations

Listed in table 1 are the abbreviations used in the project along with their meaning.

| **Abbreviation** | **Meaning** |
|---|---|
| ADHD | Attention Deficit Hyperactivity Disorder |
| API | Application Programming Interface |
| ASD | Autism Spectrum Disorder |
| CLI | Command Line Interface |
| DTO | Data Transfer Object |
| ECTS | European Credit Transfer and Accumulation System |
| EF | Entity Framework |
| ERD | Entity-relationship Diagram |
| Giraf | Graphical Interface Resources for Autistic Folk |
| HATEOAS | Hypermedia As The Engine Of Application State |
| ID | **Id**entifier / **Id**entification |
| IWWP | Iterative Waterfall With Pipelining |
| JSON | JavaScript Object Notation |
| JWT | JSON Web Tokens |
| MVP | Minimal Viable Product |
| PO | Product Owner |
| REST | Representational State Transfer |
| SQL | Structured Query Language |
| UML | Unified Modelling Language |
| URI | Uniform Resource Identifier |
| XML | Extensible Markup Language |

Table 1: Abbreviations used throughout the report

# Contents

# List of Figures

# Chapter 1

# Summary

This report documents the work of group sw613f18 in the 2018 AAU Giraf multi-project. Giraf is an environment of applications, primarily aimed at helping children diagnosed with ASD or other psychiatric development disorders. The project is split into six groups, each group responsible for a specific area, where we, group sw613f18, were tasked with developing the server-side API as well as updating and maintaining the database.

At the time of takeover of the project, no Giraf application was fully functional, with the exception of the Voice Game. The last year's Giraf multi-project decided to change the server-side API to .NET-core instead of continuing development on the existing Java server-side, as well as changing the API endpoints. These changes were unfortunately not fully integrated with the front-end applications. This year we decided to focus solely on one application, to ensure it would work properly and live up to the expectations of the stakeholders. The stakeholders prioritised the WeekPlanner application and wanted the project to focus on making it usable and stable.

The following sections summarise what was done in each of the four sprints of the project.

## Sprint 1

This sprint describes the initial user stories created in collaboration with the stakeholders of the project and the state of the backend as it was left by the previous Giraf multi-project. The primary focus, besides implementing the functionality of the shared user stories, was updating all unit tests to match the code again.

We assume the reason behind the sub-optimal state of the code at the time of handover was partly caused by a lack of a proper Git Workflow, as it seems the last Giraf multi-project seldom used more than a single Git branch. Therefore, to ensure that next year's project groups will have at least one working stable version of the different components, we decided to start using a well-defined Git Workflow and described this and other practices for version management in a configuration management plan. Furthermore, we changed the response types to a well-typed format and added Swagger [1] to document the API.

## Sprint 2

Since the majority of the first sprint was spent on getting to know the code repositories and refactoring the code itself, the multi-project did not complete the necessary functionality of the user stories for this sprint. Therefore, the second sprint continued with the same user stories. Since the backend functionality of these user stories was more or less complete, we instead created technical stories, with the purpose of improving the code base. Examples of these technical stories are implementing pictogram search and pagination for quicker and easier pictogram access. In addition to improving unit tests, integration tests were also added to ensure that the related units were also working in combination with each other and to ensure that we also test dependencies like retrieving data from an actual database. A code coverage analysis was conducted to ensure that an acceptable amount of the endpoints was tested.

## Sprint 3

The third sprint prioritised improving the structure of the week functionality, such as making it unique on user, week number and year instead of using an identifier for each. It also included optimising the idea of week templates such that a template was not dependent on an actual instance of a week, but instead inherited from a base class.

After two sprints without a release, we decided it was time to get something published on the Google Play Store. A discussion with the group responsible for the development method used in the project, led to a decision on a set of features necessary before a release could be published. This discussion also led to a consensus on the need to re-think the current software development model, as it had never been formalised. A new software development model was created with the goal of ensuring frequent releases.

## Sprint 4

The scope of the final sprint of the project was chosen to be smaller than the average sprint, partly because the groups had to focus on non-development work such as documentation and report writing. We decided to prioritise documentation of the project, as we wanted to help the upcoming Giraf multi-project groups and minimise the time they would have to use to understand the project. One of the main features we implemented in this sprint is an admin panel aimed at helping the customer to perform administrative actions such as managing the citizens and guardians in their department.

## Conclusion

As stated by the stakeholders at the beginning of the project, the primary focus was to continue working on the WeekPlanner application, ensuring stability and making it usable. First and foremost, it was important for all this year's project groups that the customers were able to get a working product in their hands, they could download and use in their daily lives. We succeeded in making three stable releases, in addition to defining a software development model that worked better than the one initially used, including better configuration management.

# Chapter 2

# Introduction

Giraf (Graphical Interface Resources For Autistic Folk) is a collection of applications aimed at helping young people diagnosed with autism spectrum disorder. The collection includes applications to ease the everyday lives of the users but also includes educational games and administrative tools. Giraf also seeks to help nonverbal autistic children, which is a specific kind of autism where the person is either not verbal at all or is limited to a few words or sentences. An example of an application specifically targeted to nonverbal autistic children is the Voice Game which teaches children to regulate the volume of their voice.

## 2.1 About Giraf

The entire Giraf application environment includes a number of applications. Among these apps, only the Voice Game and WeekPlanner are ready for use at the moment, as the other applications are outdated and not working. The official Giraf website is located at `http://giraf.cs.aau.dk`. The 2018 Giraf multi-project groups, as well as the stakeholders of the project, decided to prioritise the focus on the WeekPlanner application. The purpose of this application is to replace the physical week schedules already used at Børnehaven Birken (Kindergarten Birken). We, group sw613f18, have primarily been tasked with development and stability of the backend of the WeekPlanner application as well as making the configuration management plan for Giraf i.e. establishing the conventions used for controlling the different components of Giraf. The Giraf project has been further developed this year with support from Kindergarten Birken and the Centre for Autism and ADHD at Aalborg Municipality.

Below follows a list of terminology used in Giraf and their corresponding meaning.

| Terminology | Meaning |
|---|---|
| Citizen | A child from a kindergarten |
| Guardian | Caretakers i.e. kindergarten, employees, or parents |
| Settings | A GirafUser's personal settings for the Giraf applications |
| Department | Institutions e.g Børnehaven Birken |
| Pictogram | Image representing an object, concept, or action |
| Activity | A pictogram representing an activity on a week day in a week schedule |
| WeekSchedule | A schedule for a citizen containing a list of days that each contains a list of activities |
| Phabricator | Web application for managing software projects |
| Phriction | A wiki consisting of useful articles regarding the Giraf multi-project on Phabricator |
| Børnehaven Birken | A kindergarten for children with symptoms of psychiatric development disorders. Also one of the primary stakeholders of the 2018 Giraf multi-project |

Table 2.1: Giraf terminology used throughout the report

The Giraf multi-project groups and their responsibilities can be seen in table 2.2.

| Group | Responsibility |
|---|---|
| sw608f18 | Frontend |
| sw609f18 | Application and Product Owner (PO) |
| sw610f18 | Frontend and Scrum Master |
| sw611f18 | Server |
| sw612f18 | Server |
| sw613f18 | Backend |

Table 2.2: Giraf groups and their responsibility

## 2.2 Project Goal

The primary goal of this year's Giraf project is to make the WeekPlanner stable so it is in a usable state. Making the WeekPlanner stable was important for the stakeholders, especially Kindergarten Birken who stated that the application is an essential tool which would substitute their current physical activity boards (see figure 2.1), which they currently use daily.

## 2.3 WeekPlanner

The WeekPlanner is a part of the Giraf application suite with the purpose of replacing the acticity boards used in e.g. Kindergarten Birken. As mentioned in table 2.1, Kindergarten Birken is a kindergarten for children with symptoms of psychiatric development disorders such as ASD, ADHD, Tourette syndrome, and other related disorders [2]. The current activity board is physically located in the kindergarten and takes up a lot of space, as each child in the kindergarten has their own full 7-day board. The boards schedule the day of each child by ordering multiple pictograms, indicating which activity the child should perform, starting when they arrive at the kindergarten and until they leave. An example of an activity board can be seen in figure 2.1. An example of how the activity board looks in the WeekPlanner application before the 2018 project can be seen in figure 2.2. Figures 2.1 is from the introductory slides found in [3], and figure 2.2 is from the official website found at [4].

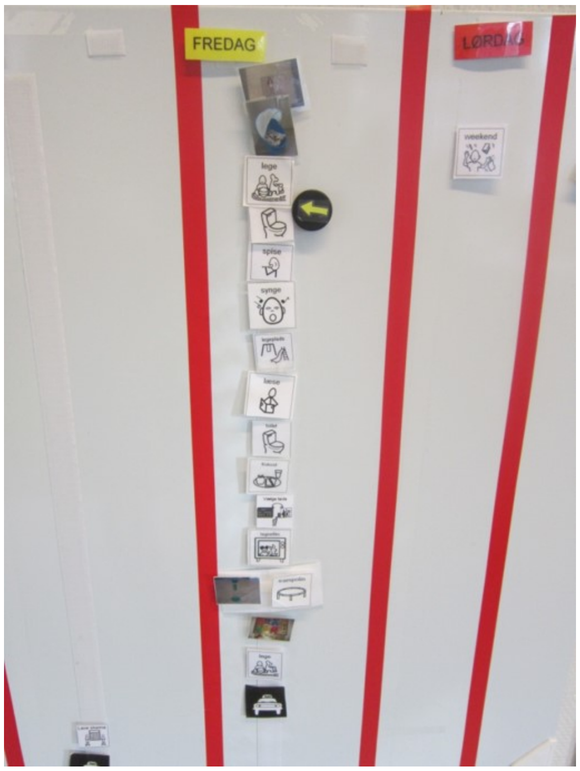


Figure 2.1: A picture showing the physical activity board at kindergarten Birken

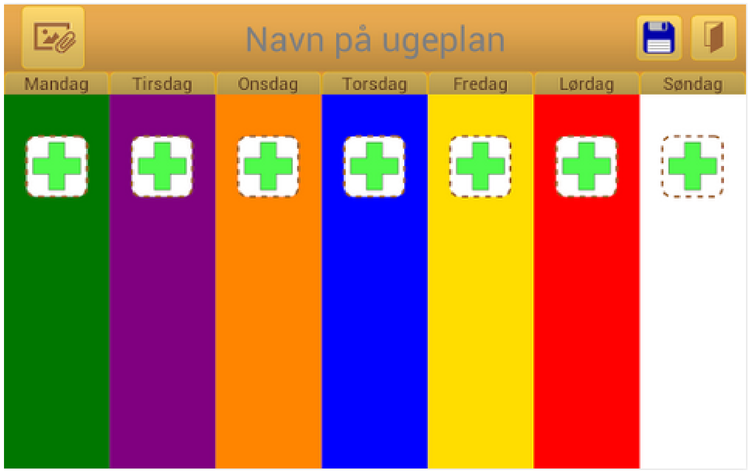


Figure 2.2: A picture showing the WeekPlanner app before the 2018 Giraf multi-project

## 2.4 API

An API is a software interface that allows for communication between software components. Consider the example in figure 2.3 where several components communicate directly to a common database where each client executes queries directly on the server. This gives a tightly coupled design where each client is directly coupled to the database, making it hard to maintain when the database is altered. Furthermore, there is no uniform communication to the database and code has to be maintained across many clients. This architecture is also not desirable for security reasons if for instance a company wants to control the queries sent to the database.

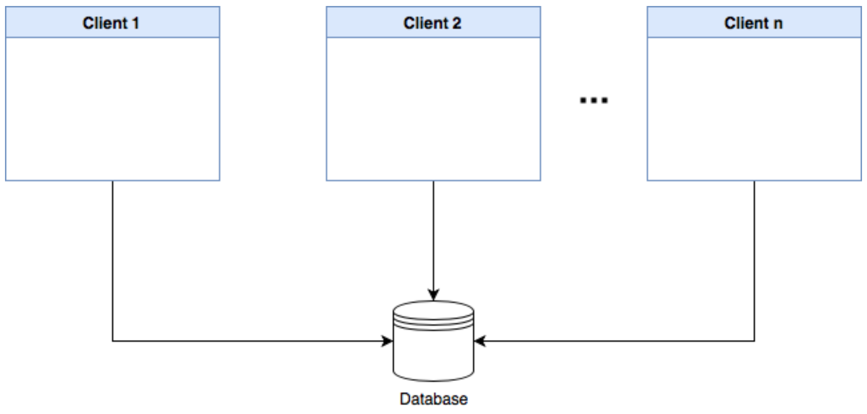


Figure 2.3: Many clients directly accessing the same database

As illustrated in figure 2.4 an API acts as an interface for the communication to the database which gives a more loosely coupled architecture, where the API is responsible for all queries sent to the database. This has the advantage of higher stability, as all checks can be made in the API and not with the clients.

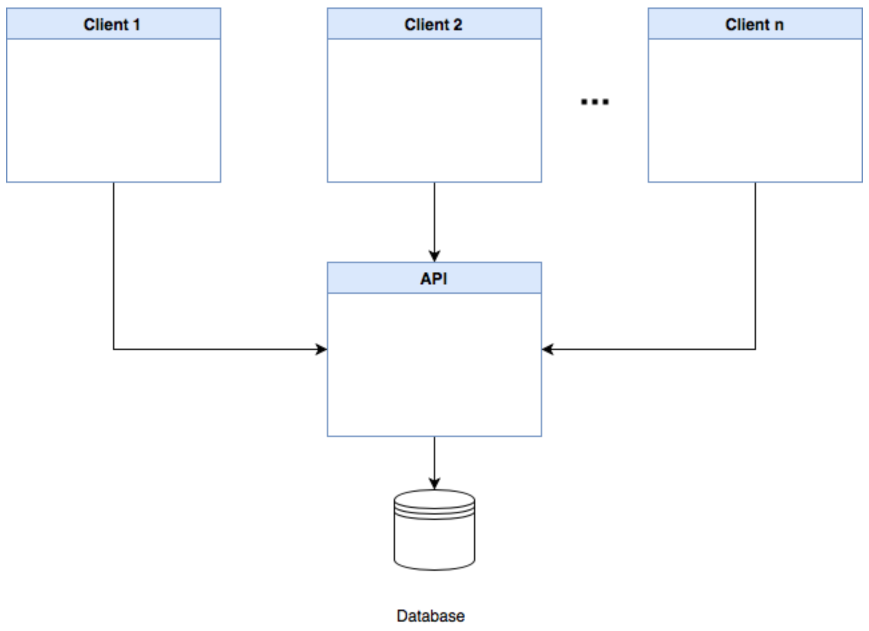


Figure 2.4: Many clients accessing database through an API

To communicate with the API we will be using REST which is a set of constraints used for communication between components [5].

### 2.4.1 REST

REST stands for Representational State Transfer and was first phrased by Roy Thomas Fielding in his paper *Architectural styles and the design of network-based software architectures* and is a standard that defines a uniform interface for communication between components [5]. In many ways, REST has become a standard for making modern web applications and was defined by Fielding to reflect properties that he believed modern web applications should have [5, p.76-77].

An important aspect of REST is the separation of client and server. By making this separation we get a more loosely coupled design where the tests of storing and retrieving data is separated from the task of displaying the data to a user. This improves the portability of the system as new user-interfaces can be defined and use the data on the server independently of other clients, which allows the client and server to be developed more independently [5, p.78]. Furthermore, each request to the server should be stateless, meaning that the request should contain all necessary information for computation by the server i.e. information for retrieving data from storage [5, p.78-79]. Because of this, REST does not recommended to use data from the web session as this is not stateless.

An important term in REST is a resource which is any "[. . . ] information that can be named [. . . ]" [5, p.88] such as in the context of Giraf, e,g. a pictogram or a user. A resource identifier is then any identifier that uniquely identifies a resource [5, p.90].

REST uses representations to perform actions on resources and for sending responses back to the clients [5, p.90-92]. A representation can for instance be in the form of JSON or XML which can be used to represent a resource.

To allow for easier communication between client and server; REST defines a uniform interface based on the following four constraints defined by Fielding [5, p.82]:

- Unique identification of a resource
- Manipulation of resources by using resource representations
- Self-descriptive messages
- Hypermedia as the engine of application state

In this project, we will mostly be implementing an API based on the first three constraints. The last constraint, also called HATEOAS, specifies that each response contains relevant URLs. However, as explained in [6], this is rarely used in modern REST APIs and will not be a focus to support throughout the API. The reason for this, is that HATEOAS, as explained in [6], is used to define the control logic of the API in form of links which the clients can use to make requests. These requests have the benefits that they do not break the client if, for instance, the URLs change. However, as described in section 4.3.5, we will use Swagger for documenting the API and generating the client API, meaning that any changes to the backend will be documented on the Swagger UI and reflected in the generated code. We will, however, use HATEOS where it makes sense, which for instance is providing URLs for the images associated with pictograms.

As we do not completely follow all constraints originally phrased by Fielding in [5], we will refer to the API implemented in the remainder of the report as a REST-like API. As a general rule of thumb, the following naming convention will be used to specify the URL to the different endpoints in the API: `host/resource/identifier`

The different part of the URL is as follows:

**`host`** denotes the base URL to the website that hosts the API

**`resource`** denotes any information that can be named

**`identifier`** denotes a unique identifier for identifying a specific resource

Now that we have described the purpose of this year's Giraf multi-project, as well as our role in the multi-project setting and the kind of API we will develop and maintain, we will in chapter 3 describe the process model that we have used and how it changed through the semester.

# Chapter 3

# Development Process

The purpose of this chapter is to describe the development process used both internally in the group and in the multi-group environment during this year's Giraf project. The chapter describes how the process model evolved during the different sprints and why these adaptations were made.

## 3.1 Initial Process Model

Figure 3.1 illustrates the initial process used by all the groups in the Giraf project. Before each sprint begins there is a planning phase where the user stories to be included in the next sprint are planned by the product owner group along with a prototype for these user stories.

The sprint then begins and the different user stories are split into smaller tasks which are distributed to the different groups depending on their responsibility areas. Some groups such as the server groups, do not directly implement tasks related to user stories, but more technical stories such as implementing continuous integration, build scripts and so forth. When a task is finished, meaning it has been implemented and tested, it is integrated into the system.

This process continues until the sprint ends. During a sprint, scrum meetings occur twice a week, where at least one representative from each group is attending. During these meetings each group discusses the following questions:

- What have we done since the last meeting?
- What will we work on until the next meeting?
- Do we have any problems that we need help with?
- Have we made any decisions that can affect the other groups?

At the end of a sprint, an acceptance test with the stakeholders is conducted. The purpose is to show the current state of the project and gain feedback, as well as define new tasks to develop in subsequent sprints. Based on this feedback, the PO group makes new user stories and creates a prototype along with the tasks to be implemented in the next sprint, as well as any issues that might need to be addressed from the tasks developed in the current sprint.

After the acceptance test, the sprint review and retrospective are then held among all the Giraf project groups, where the purpose of the review is for the PO group to discuss the acceptance test, problems and to present the prototype for the new sprint, as well as estimate the time to implement new user stories. Afterwards, the retrospective is held where the process is discussed in terms of what worked well, and what did not. The process is then changed according to this discussion. Based on the total number of sprints, an additional increment of this process is either taken or the project ends if the current sprint is the last sprint.

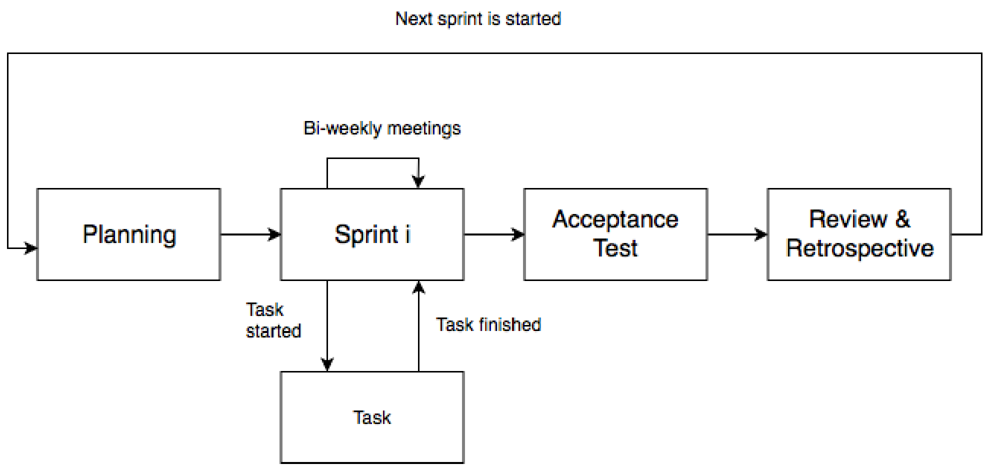


Figure 3.1: Scrum model used for the Giraf project (bi-weekly meaning twice a week)

## 3.2 Adapting the Process Model

When planning the third sprint, we wanted to include a large number of user stories, compared to previous sprints, to ensure all participants had something to work on. This, however, resulted in many half-finished features at the end of that sprint. That, and the fact that we had no plan for when to stop and prepare the sprint for a release, meant that there was a chance that no version of the software would ever be released in any sprint. This might have been mitigated by setting aside a week at the end of each sprint to prepare a release. However, if we at the final sprint, missed the deadline, the work for that sprint would have been lost, and as such, it would be preferable to divide a sprint into several smaller releases.

In collaboration with the other groups, we revised our plan for giving the customers as much value as possible. We introduced weekly releases as a cornerstone of our inter-group development process. The idea is for the product owners to create a list of releases, where each release contains a minimal, but functional, set of features. Each successive release should, in general, include all features from the previous release, and additionally have extra functionality. If the requirements and wishes from the customers change between releases, this property might not always be satisfied. A release is naturally not finished until the project has been released and is available on the relevant app-stores.

While a release makes sense in itself, it is important for a proper execution plan to be in place, especially when several teams are involved. Figure 3.2 shows the new schedule that all groups must adhere to in order for the release plan to function.

This semester we decided to make use of horizontal development in spite of it also making use of agile software development. This is widely considered wrong when working agile, as "Working in thin vertical slices is the keystone habit for agile software development." [7]. The reasoning behind this choice of horizontal development is, that the developers of the project are inexperienced in working together with other teams, as well as inexperienced in regards to a lot of the software frameworks used in the project. The Giraf multi-project is also very time-limited, as only 15 ECTS are assigned for it. If we chose vertical development instead, every group had to spend time getting to know each framework in each 'slice' of the architectural layers, which would be beneficial in regards to knowledge, but would mean a significant overhead in terms of time spent for each person to get familiar with all relevant code-bases.

A disadvantage of the horizontal development method is bottle-necking where one group can slow down the entire development process. We worked around this by using pipelining: Even though one step in the assembly line is a bottleneck (a team being behind schedule), we can reallocate resources (whole teams or team members) from one component to another if needed. This proved useful, as we began with two designated server teams, but as the frontend teams began bottle-necking the project, we reallocated one of the server teams to the frontend.

We will refer to this new software development method as *iterative waterfall with pipelining* or IWWP for short. A graphic illustrating our use of IWWP can be seen in figure 3.2.

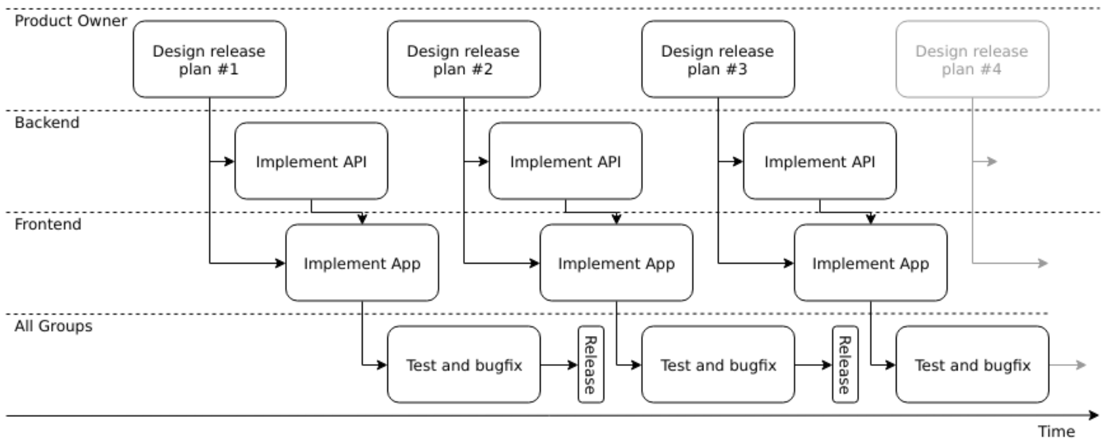


Figure 3.2: Iterative Waterfall with Pipelining (IWWP)

First, the PO group creates the release plan which includes:

- A set of coherent and complete features
- A functioning prototype showing all features to be included in the release

Once release #1 is defined, the backend is designed. The backend, in collaboration with the different groups, discuss the changes needed for the backend to support the features laid out by the product owner. All developers discuss which changes should be made to the system as it currently is. This entails 'hiding' features which is outside of the current release-scope and establishing which endpoints are needed in the backend.

Each group then implements the features necessary to satisfy the release plan. When a group has finished implementing all necessary features, they create a new release branch from the develop branch. New features for release #2 can now be implemented on the develop branch and bugs on the release branch can be fixed without being tainted by the develop branch.

The backend group should, as quickly as possible, perform the required changes to their release branch and notify the frontend groups once finished. The frontend groups will finish the implementation of the release while using the functionality of the backend on the corresponding release-branch, to ensure that everything works end-to-end.

Once the groups are satisfied with the quality of the release, they deliver it to the PO group for a final stress/acceptance-test. If the PO group find any problems, these should be fixed and the process repeated. Otherwise, the applications will be published to the relevant app-stores.

An example of a release schedule can be seen in figure 3.3.

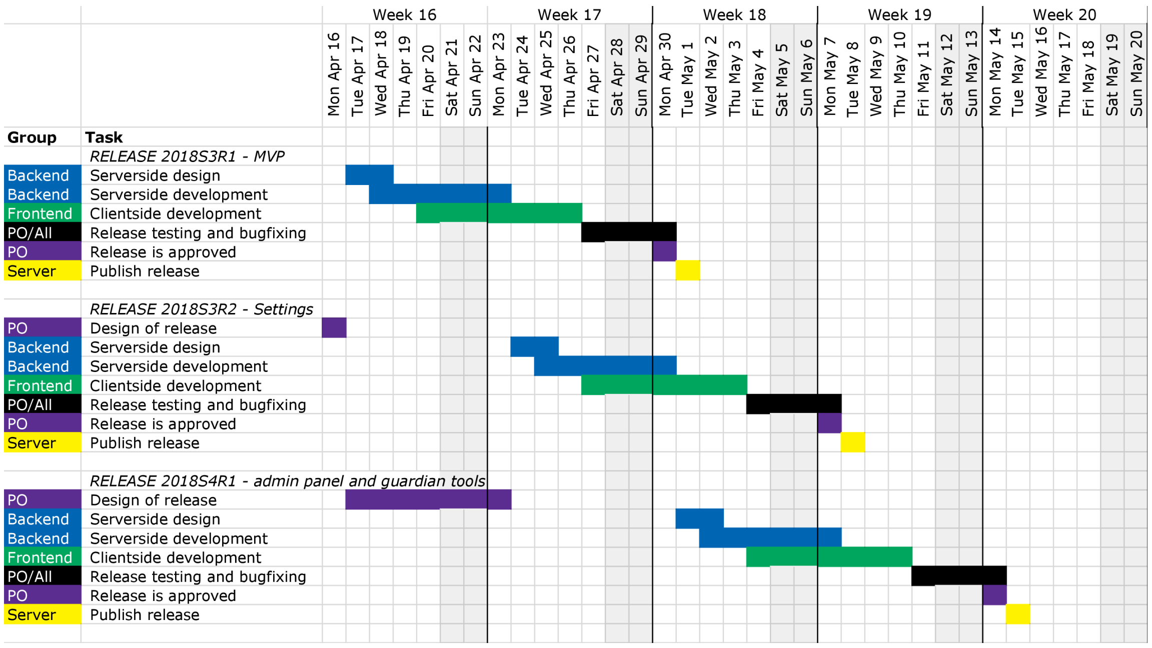


Figure 3.3: Gantt chart for releases according to the IWWP model

Shorter release cycle lengths are preferred, as they ensure that fully functional code gets pushed to a release-branch more often. This in turn ensures that the released code is often updated to the newest features. This is good for morale, as everyone, including the customer, can see progress. It also ensures that as little code as possible will be abandoned on a release branch after the last sprint has ended. However, short iterations may result in more time spent on things like preparing a release for publishing on the different app stores and conducting acceptance tests with the stakeholders.

We propose two methods for setting deadlines and the length of a release cycle. One option is fixed schedules, e.g. having a release published every Thursday, and performing tests of the code over the weekend. A fixed schedule helps every project member remember their relevant deadlines.

A second option is to define the length of the release cycle in terms of man-hours. This approach fits well with the weekly changing schedules of Aalborg University lectures.

We chose to use a fixed schedule, as IWWP was first implemented in sprint 3 after we finished most of our lectures, meaning there would be few distractions from the project work.

Figure 3.4 shows how the IWWP model fits into the overall process model and as illustrated, the only thing that has changed from the previous process (see figure 3.1) is the software development method i.e. how software features are implemented. Figure 3.4 illustrates that one or more releases according to the IWWP model might be produced in each sprint.

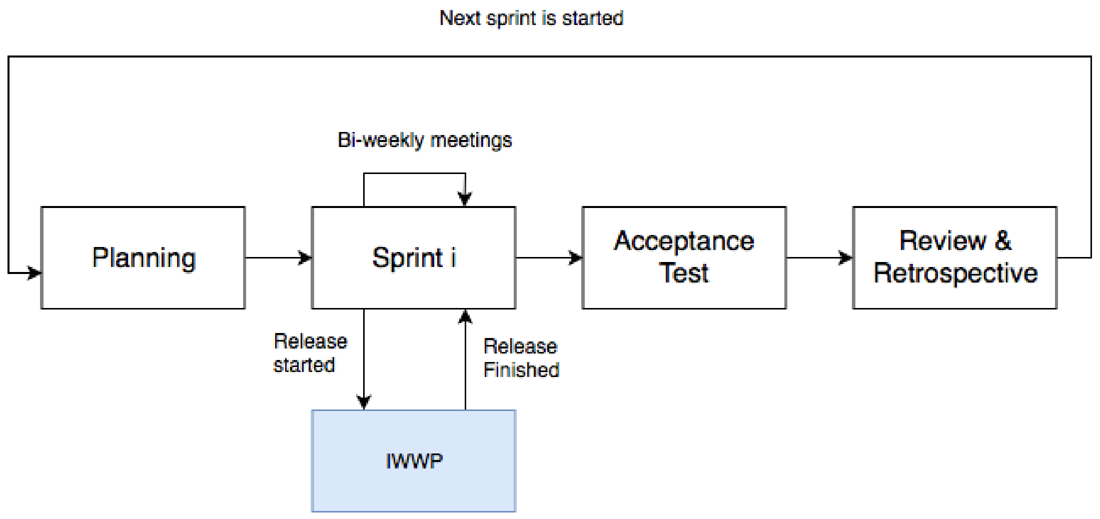


Figure 3.4: The new process model (bi-weekly meaning twice a week)

Now that we have described the process model used in this year's Giraf multiproject we will in chapter 4 to 7 describe the four sprints with respect to the work that our group did in the Giraf multi-project.

# Chapter 4

# Sprint 1

This chapter describes the initial sprint of the Giraf project. In section 4.2, we describe the overall goal of the sprint in the form of user stories formulated by the Giraf project groups in collaboration with two representatives from Kindergarten Birken, which can be found in appendix C.2.

Based on these user stories, our team, responsible for the backend, will examine the initial state of the code in section 4.1, and, with this in mind, we create additional backend-specific tasks to complete in this sprint. These are shown in section 4.2.2 and in section 4.3, we describe how the functionality of these tasks are implemented in the project.

## 4.1 Initial State of the Backend

The initial state of the backend, as we received it, can be seen in appendix B. The API was mostly working and we were quickly able to get it running by restoring NuGet packages and downloading .NET Core 1.1.1.

To produce an initial backlog of tasks for the API, we examined the existing code. The primary problems were discovered in the unit-tests, which had not been updated and could therefore not be compiled.

Additionally, the error responses from the API were vague, and in some cases not representative of the errors that occurred. Furthermore, the .NET Core version used was outdated. An example of a typical API error code was `HTTP 400 (Bad Request)`. This error could e.g. look as follows: `return BadRequest("No username specified.");` when the user forgot to specify a username. However, it should not be the responsibility of the backend to generate strings that might be shown to the user. Furthermore, you will have to check the class type of each response to determine whether or not one made a successful request. A better approach would instead be to define a response class and have different fields like error codes, a success flag, and data fields that are universal for all endpoints.

## 4.2 Planning

In this sprint the focus is to implement the tasks described in appendix C.1. Examples for the user- and tech-stories implemented in this sprint is given below in section 4.2.1 and 4.2.2, respectively.

### 4.2.1 User Stories

In the early phases of this sprint, the PO group prioritised the different user stories based on the needs of the stakeholders and decided that a guardian should be able to log in to the system, choose a citizen to manage, as well as create, save, and read week schedules. The full definition of the user stories can be seen in appendix C.1.

### 4.2.2 Tasks to Complete in the First Sprint

In order to make backwards compatibility easier to maintain in the future, we want to make the URLs of all API endpoints to include a version number, so that older versions can still be made available.

Moreover, we want to look into a better way than wiki pages for documenting the endpoints of the API for the other groups to use.

The purpose of this sprint is also to test and document the backend of Giraf, hence the database and the .NET Core API with respect to the tasks below:

- Improve responses when errors occur.
- Restore the unit tests to working order.
- Update .NET to the newest version.
- Include API version number in endpoint URL.
- Set up a system to document endpoints.

#### Time Span

At the first sprint planning meeting on the 12th of February, it was agreed that the time span of the first sprint was from 12th of February to 7th of Marts. This corresponds to 22 half days of 4 hours each, totalling 88 hours per person.

## 4.3 Development

The purpose of this section is to describe the primary tasks completed by group sw613f18 in sprint 1. These tasks were selected and described in section 4.2, and the following subsections will describe the development of these tasks.

### 4.3.1 Organising the Git Repository

In one of the initial stand-up meetings with the other project groups, it was mentioned that a configuration management plan would be useful to establish consistent guidelines across the groups regarding version control and Git workflow.

Initially, the repository contained only the master-branch. In earlier projects, we have had success working with the GitFlow Workflow [8], which describes creation of branches for releases, hotfixes, current development, and features. We implemented this model in the repository early in sprint 1 to split up work and avoid 'breaking the build' when pushing non-working code.

This means that we can have both a stable release on a `release`-branch that can be used to deploy, along with a latest build available to other groups for development on a `develop`-branch, which will be used in the next release. Finally, we have different `feature`-branches where a new feature can be implemented, before being merged into the current `release` branch. The purpose of `feature`-branches is to implement features that typically involve breaking the build temporarily, before pushing the ideally error-free code to the current `release` branch or `develop`, which should only have working code.

With few changes to the GitFlow Workflow which make it more useful when working with IWWP, we created a configuration management plan. The plan was presented to the other Giraf groups which all agreed to follow it. The configuration plan can be seen in appendix A.

### 4.3.2 Calling the API

Given the user stories of this sprint, we determined the API endpoints that are required to support the relevant functionality. The code base was fetched, built, and run, and calls to said endpoints were made using Postman [9]. The database was inspected to ensure that updates were made to it as required.

These calls suggested that the API already supported the following functionality:

- Log into the system
- Show all week plans for the user currently logged in
- Show specific week plan (specific ID)
- Create a new week plan
- Edit a week plan
- Show all users associated with the user currently logged in
- Show specific user (specific ID)

Notably, it was not possible to view week plans and related information for users other than the one currently logged in, regardless of whether they have the edit rights. Instead, if a guardian wished to manage the week plan of a citizen, they had to log in as that citizen (without the password being required, as they have edit-rights) in order to make the relevant changes, whereupon they could log back into their own account to continue working. This is illustrated in the sequence diagram shown in figure 4.1.

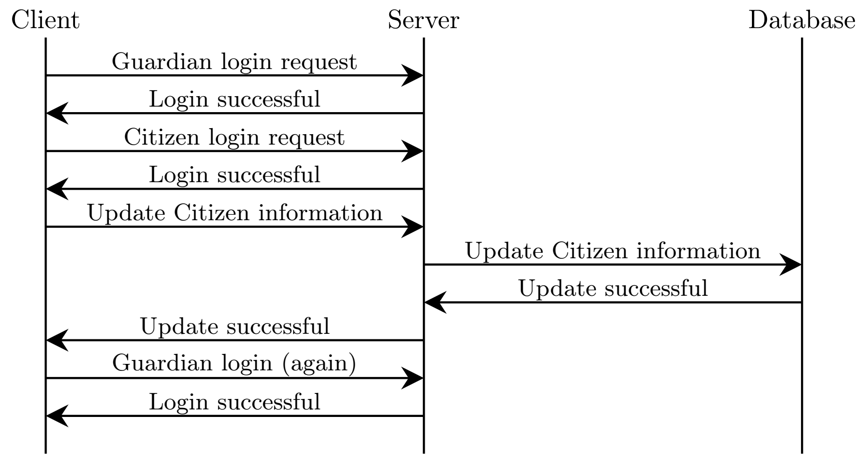


Figure 4.1: Sequence diagram showing how a guardian has to navigate the system to change a given citizen's information

Figure 4.1 is a simplified version of this use case. Some details are omitted, such as looking through the database to accept the login details and checking that the guardian is a guardian of the given citizen.

Having to navigate the system this way just to update the information of a citizen is arguably a major flaw in the API, but one that the current implementations across the Giraf project currently relies on. As such it is not a goal of sprint 1 to improve upon this system. The ideal functionality would be to give guardians access to view week plans and related information for every user they are a guardian of, without requiring them to log in again. Functionality for this is implemented in sprint 4 which is described in chapter 7.

### 4.3.3 Updating .NET Core

We updated .NET Core from 1.1.1 to the latest version, 2.1.4, as of 2018-03-07. This required some updates to e.g. how the web-host is built and run, how the database is initialised, and how logging is set up [10]. It also required us to notify any other group this update may concern, to ensure they would not have unexpected troubles with an outdated version.

### 4.3.4 Changing Response Types

When we received the codebase, most of the API endpoints had the `IActionResult` as response type. The problem with this is that it enables the same endpoint to return data structures of different types. This makes it difficult for the consumers of the API to predict and parse the response from an endpoint. This required type-casting and -checking when determining the result of a call.

For this reason, we decided that the type of the response objects should be changed throughout the code into a class named `Response`. This class contains a field stating whether the request was successful, a enum of error codes, as well as a generic data field for DTOs.

We also defined a class, `ErrorResponse`, that, by convention, is used when an error occurs. This class inherits from `Response` and differs only in its constructor.

Because the response types were changed across the entire API we had to refactor all of the unit tests in addition to the controllers. The response types were changed early in the first sprint and the relevant project groups were informed of this change during a Scrum stand-up meeting.

Listing 4.1 shows the base class, `Response`, that we implemented for responses. This class contains a flag `Success` indicating whether or not a request was successful, a possible `ErrorCode`, and an additional field, `ErrorProperties`, for storing information related to error codes such as which properties in a request caused an error. From the `Response` class, we implemented the generic `Response` class shown in listing 4.2. This class includes a generic field `Data` for storing the data returned by the API. Similar to the generic `Response` class, we also implemented a generic `ErrorResponse` class, which is used when a request fails e.g. when invalid credentials are provided during login.

```
public class Response
{
    public Response()
    {
        Success = true;
        ErrorCode = ErrorCode.NoError;
        ErrorProperties = new string[0];
    }
    public bool Success { get; set; }
    [JsonIgnore]
    public ErrorCode ErrorCode { get; set; }
    public string[] ErrorProperties { get; set; }
    [EnumDataType(typeof(ErrorCode))]
    [JsonConverter(typeof(StringEnumConverter))]
    public ErrorCode ErrorKey
    {
        get
        {
            return ErrorCode;
        }
    }
}
```

Listing 4.1: The base class for Responses

```
public class Response<TData> : Response where TData : class
{
    public Response(TData data, params string[] missingProperties)
    {
        Data = data;
        ErrorProperties = missingProperties;
    }
    public TData Data { get; set; }
}
```

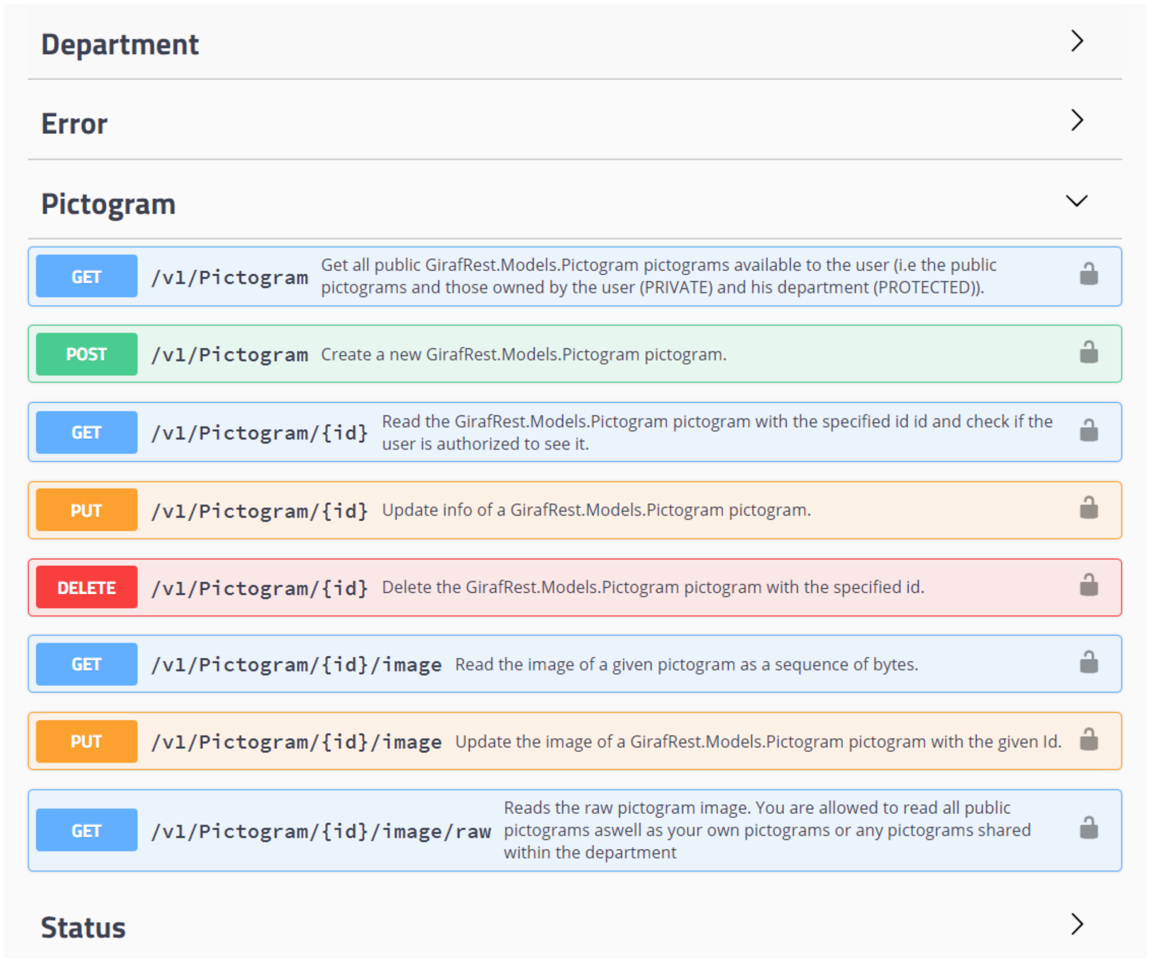
Listing 4.2: The generic Response Class

### 4.3.5 Swagger

Swagger is a framework of API developer tools for the OpenAPI Specification (OAS) which defines an interface that allows humans and computers to easily understand a given API, without access to the source-code or other documentation [1]. We decided to implement Swagger as middle-ware in our API to document the API to the Giraf groups responsible for the client-side applications. Swagger also provides the possibility of automatically generating client-side APIs in different programming languages, for calling our API.

To add Swagger to the API, we added a NuGet package for Swagger and configured the API to use Swagger as a service. After having successfully set up Swagger then the Swagger UI could be accessed by navigating to `<host>/swagger` e.g. on `localhost` on port 5000: `http://localhost:5000/swagger/`.

At the Swagger UI, the different models and end-points of the API are documented, as seen in figure 4.2.



GET `/v1/Pictogram` Get all public GirafRest.Models.Pictogram pictograms available to the user (i.e the public pictograms and those owned by the user (PRIVATE) and his department (PROTECTED)).

POST `/v1/Pictogram` Create a new GirafRest.Models.Pictogram pictogram.

GET `/v1/Pictogram/{id}` Read the GirafRest.Models.Pictogram pictogram with the specified id id and check if the user is authorized to see it.

PUT `/v1/Pictogram/{id}` Update info of a GirafRest.Models.Pictogram pictogram.

DELETE `/v1/Pictogram/{id}` Delete the GirafRest.Models.Pictogram pictogram with the specified id.

GET `/v1/Pictogram/{id}/image` Read the image of a given pictogram as a sequence of bytes.

PUT `/v1/Pictogram/{id}/image` Update the image of a GirafRest.Models.Pictogram pictogram with the given Id.

GET `/v1/Pictogram/{id}/image/raw` Reads the raw pictogram image. You are allowed to read all public pictograms aswell as your own pictograms or any pictograms shared within the department

Figure 4.2: Shows some endpoints from the Swagger UI

By using the Swagger UI, it is possible to access each endpoint of the API, see a description of the endpoints, examples of how to use them, and even use the UI to make requests to the API, eliminating the need for third-party programs like Postman [9].

### 4.3.6 Supporting other Groups

As explained earlier in section 4.3.5, we decided to implement Swagger as a middle-ware in our API.

This gives us the possibility of automatically generating the client-side Java API, which we suggested to the frontend groups, who are responsible for maintaining the current client-side API. Generating the client-side API has the advantage that changes to the API are automatically reflected in the client-side API when generated.

However, the client groups suggested sticking with the currently implemented client-side API, which, at first glance, seemed stable. Furthermore, the client-side API used the Volley HTTP library [11], which Swagger initially had some errors generating due to syntax errors in the generation of enums. This was however eventually fixed.

Based on this, we decided to move forward and help the different groups in Giraf with small tasks such as running the API locally, fixing bugs, and changing the launch-settings of the API for the server groups.

At the end of the sprint, the groups working on the application and client-side API came to the conclusion that all client-side code was unstable and would take a long time to fix. Because of this, the client-side groups wanted to investigate the possibilities of writing the WeekPlanner app from scratch in Xamarin [12] instead of continuing maintenance. This would also open up for the possibility of making the WeekPlanner application cross-platform.

A vote was held on the recommendation of the frontend groups, and a majority decided upon starting anew using Xamarin.

Due to this request, we used Swagger-codegen [13] to generate a client-side API in C# with the HTTP library RestSharp [14] and instructed the groups in how to use this generated code to call our API as well as helping the client groups deserialising different API responses.

## 4.4 Test

Initially, when we took charge of the backend, there were 227 unit tests using the xUnit framework. However, the test project did not compile. This was due to late changes in the API from the previous group that had not been carried over into the test project.

After we updated the test project and got the tests to compile, 44 tests failed, 28 of which were because of a bug in the test helper method `InitializeTest` in `AccountControllerTest.cs`, where an exception was thrown in all cases it was called.

When we chose to change all the response-types of the main project, all tests obviously no longer compiled, and therefore, we had to rewrite every test. Refactoring and changing the tests helped us discover many errors in the API, especially missing null-checks.

After the unit tests were fixed, it was decided that the unit tests should run before each commit to a `develop`, `release` or `master` branch.

## 4.5 Conclusion

Through testing, we discovered that a majority of the user stories were already supported in the API, and therefore, we spent our time fixing and creating new unit tests, refactoring code, and fixing bugs. Furthermore, we created technical stories for new features, such as documenting the API by integrating Swagger and changing the response types used throughout the API.

# Chapter 5

# Sprint 2

In this chapter, we will describe the second sprint of the Giraf project. The primary focus of this sprint is to assist the front-end groups in developing the WeekPlanner application, by ensuring that the backend is stable and can accomplish the tasks needed in correspondence with the user stories listed in appendix C.2.

## 5.1 Planning

Because it had been decided to recreate the WeekPlanner app from scratch, the user stories from sprint 1 remained incomplete and were therefore copied directly over as the user stories for sprint 2.

However, as described in chapter 4, the backend functionality required for the user stories had, for the most part, been completed. For this reason, we created our own technical stories, intended to increase the reliability, stability, and readability of the backend code base.

### 5.1.1 Technical Tasks

A complete list of the technical stories produced by the backend group can be seen in appendix C.2, while some of these tasks can be seen below:

- Change local database to use MySQL instead of SQLite.
- Use token authentication instead of cookie authentication.
- Implement pagination and search functionality for Pictograms.
- Add functionality to retrieve pictograms as PNG images.
- Write scripts for automated integration tests.

The majority of the technical stories are chosen to improve the reliability of the backend, e.g. changing from SQLite to MySQL on the local database, so that it matches the database language of the production database, as SQLite does not support some common database operations like dropping foreign keys. Adding an endpoint and pagination for searching pictograms was necessary, as it would otherwise request every pictogram from the database, which was slow and unnecessary.

Some technical stories had the purpose of fixing functionality the group found unreasonable, e.g. not having a many-to-many relation between citizens and guardians or having to call two separate endpoints to create a pictogram.

While these are the main tasks for this sprint, we also created a list of minor, yet still relevant tasks to complete. These could e.g. be created upon discovery of a bug, or by request of another project group. Below is a view of a selected set of these minor tasks, while a complete list can be found in appendix C.2.

- Create indexes for tables for faster lookup.
- DTOs should only return necessary information.
- Add various endpoints, e.g.: for getting all citizens in a given department.
- Clean up Phriction for old and irrelevant documents.

### Time Span

The sprint planning meeting of the second sprint was held on the 8th of March. The second sprint spanned until the 11th of April, corresponding to 34 half days of work, which gives us $34 \cdot 4 = 144$ working hours per person.

## 5.2 Development

In section 5.1, we listed the technical stories to be solved in this sprint. We will now describe some of these technical stories, i.e. why we changed the local database to a MySQL database, why and how we implemented token authentication, as well as the algorithm used for implementing pagination and search. Besides implementing the features described in this section we also implemented many other features and bugfixes which are listed in appendix C.2.

### 5.2.1 Database Maintenance

There exist different approaches for maintaining a database in ASP.NET using EF. One approach is to manually create all the tables, relations, and constraints by writing a set of SQL statements. Based on an existing database, it is then possible to use a database first approach to generate C# classes for the API in order to interface with the database.

Another approach is the code first approach which is used in the Giraf API.

#### Code First

In the code first approach, we program the model classes first, and based on these classes, we can use EF [15] to generate a migration. A migration describes the update of tables, relations, and constraints in the database, for the changes made since the previous migration, as well as how to undo the migration i.e. reverting the changes, when going back to an earlier version of the software.

The advantage of using migrations is that it allows us to update the schema of the database without reconstructing the entire database. This is crucial in a production setting where the database stores actual information for the end users. In the production database we want to keep as much of the data as possible, even though we modify columns of the tables. Later in section 7.2.8 we give an example of how to manually write such a migration, when changes to the model-layer is made.

The code first approach is not limited to a specific database system, but some database systems are not fully supported yet. In the beginning of this sprint, we used a SQLite database for running the API locally on our development machines, and a MySQL database for production. To do this, we had to maintain migrations for both SQLite and MySQL. During an update of the model classes we encountered a problem with the use of the SQLite database. When we added a new migration for the SQLite database we discovered that some operations, like dropping a foreign key of a table, was not supported [16].

Therefore, we decided to stop using SQLite and instead use MySQL, both locally for testing purposes, and in production. The advantage of this, is that we now only need to maintain migrations for MySQL, and that we use the same database system for development as we do in production. This ensures that we remember to add and test migrations for the MySQL database and decreases the risk of problems in the production setting. A disadvantage of this is that developers of the API now have to install and setup a MySQL server of their own. This is not a major problem as MySQL can be installed on both Windows, Mac, and Linux with ease. The `README.md` file in the repository for the API contains more detailed information for developers on how to setup and run the API locally.

### 5.2.2 Token Authentication

Another goal for this sprint is to replace cookie authentication with token authentication. The reason for this, is that we want a stateless API, meaning that we do not want to manage the state of a user's session on the server. This is to better comply with the notion of a REST-like API as explained in section 2.4.1. Instead of using cookie authentication, we want to let the client manage the state of the user by using tokens. This also enables the application to obtain and store multiple tokens to enable e.g. switching between using the application as a guardian or a citizen without needing to request the API to get a new token.

We chose to use JWT [17] because it integrates the ASP.NET Core Identity library which was already used for the cookie authentication. This simplified the process of integrating token authentication, because it was mainly a configuration task.

Figure 5.1 shows the process of getting data (e.g. week schedules) using token authentication.

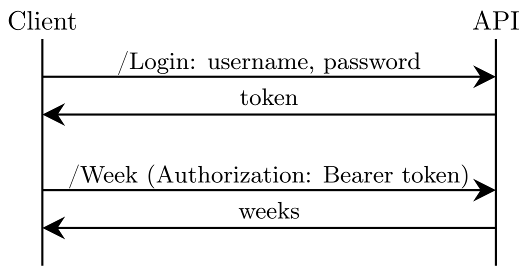

Client
API
/Login: username, password
token
/Week (Authorization: Bearer token)
weeks

Figure 5.1: Sequence diagram showing the accession of weeks using token authentication

To obtain a token, the client first authenticates the user with their `username` and `password`. If the `username` and `password` of the given user are valid, the API responds with a token. The client then stores this token, containing signed information used to identify the user. In each subsequent request to the API, the client passes the token in the `Authorization` header using the Bearer authorisation schema [17].

### Token vs Cookie Authentication

Cookie authentication is similar to token authentication in the sense that the content of the cookie can be seen as a token. The main difference is how the cookie is handled by the client. When using cookie authentication, a login request to the API will return a `Set-Cookie` header back to the client. Some clients, such as browsers, will handle this implicitly and automatically store the cookie and send it with each subsequent request. This is different from token authentication, where the client explicitly decides when and which token is sent with each request. Another difference is that a cookie is associated with a domain, which is typically the domain from which the cookie was set. This describes when the client should include the cookie, depending on which domain the client is requesting. A token is not associated to a domain and can be shared between domains if necessary.

### Splitting Up the Giraf API

Using token authentication enables us to later split up the API into smaller and more coherent services. This could e.g. be that we divide the Giraf API into an Account API for user and role management and another API for the Weekplanner-specific endpoints.

Figure 5.2 shows how the token authentication would work across such two APIs.

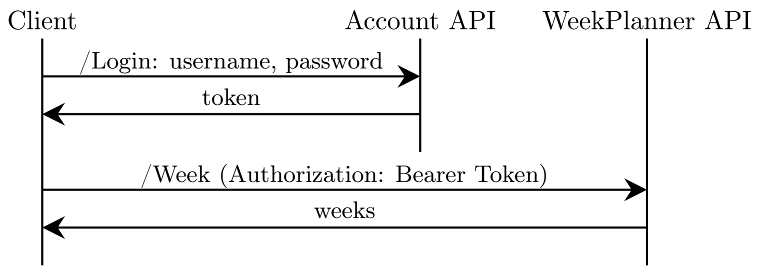

Client
Account API
WeekPlanner API
/Login: username, password
token
/Week (Authorization: Bearer Token)
weeks

Figure 5.2: Sequence diagram showing the accession of weeks using token authentication across two APIs

In the approach where the API is split into multiple APIs, as shown in figure 5.2, the two APIs will share a private key used to ensure that only tokens generated using this key is valid. This ensures that a user cannot simply generate his or another user's token.

If the Account API does not share a database with the Weekplanner API, the token needs to be self-contained i.e. contain information about which resources and actions the user has access to in the Weekplanner API. This can be obtained by using JWT, where each token contains three parts: header, payload, and signature [17].

The header contains meta information like which type of encryption algorithm is used to sign the token.

The payload contains the information about the user like id, roles, privileges etc. [17].

Finally, the signature is generated by the encryption algorithm, using the secret private key which then can be used to validate the token [17].

## 5.2.3 Pictograms: Pagination and Search

The endpoint `GET: /pictogram` retrieves the entire pictogram collection which contains, for each entry, the image data as a base64 encoded string. Since the production database currently contains more than 10.000 pictograms, we proposed two simple and effective ways of making the results less data-intensive:

1. We could limit the amount of pictograms the `GET: /pictogram` endpoint returns.
2. Redact the image data from the `GET: /pictogram` endpoint and create a new endpoint like `GET: /pictogram/{id}/image` which would return the image data for any single pictogram.

The first change is achieved through pagination. This means that all requests to `GET: /pictogram` will either explicitly or implicitly (by server-side defaults) indicate a page number and a number of elements per page i.e. `GET: /pictogram?page=3&pageSize=12`.

Secondly, applications using the API might want to search for pictograms by their title. Instead of having every application which uses the API implement their own search algorithm, we thought it best if such an algorithm at least be available through the API itself. We chose to add search functionality through the same endpoint as pagination e.g. `/pictogram?query={search_query}`.

### Implementing Search

This section describes the implementation of the search functionality. One approach is to filter all items according to some condition e.g. the search query must be a substring of the item. Another approach is to sort all items according to how similar they are to the search query. While there are still other aproaches, we will chose the second approach and use the Levenshtein distance [18] as our metric of similarity.

The Levenshtein distance, or edit distance, of two strings is the number of insertions, substitutions, and deletions of single characters required to turn one of the strings into the other. When talking about a modified cost model, what me mean is we change how much the edit distance increases when doing either a insertion, substitution, or deletion.

There are several well known implementations of Levenshtein distance. A dynamic programming implementation is the Wagner-Fischer algorithm.[19]

We use the format $(i_x, s_y, d_z)$ to describe a modified Levenshtein cost model with insertion cost of $x$, substitution cost of $y$, and deletion cost of $z$.

We implemented search by using the Wagner-Fischer algorithm to order pictograms by their title's Levenshtein distance to the query string. Later, we modified the costs of inserting, substituting, and deleting to be $(i_1, s_{50}, d_{50})$ instead of $(i_1, s_1, d_1)$.

We make the assumption that users will usually search with short query strings even if looking for somewhat specific things. As an example, if the user searches for '*leaf*', we might expect the search to return the more specific '*withered Ash leaf*'.

This section describes some of the considerations we made, as well as arguments for and against the implemented solution. We do not claim the implemented solution to be the best of the ones discussed here. We simply did not prioritise discovering a better approach until the last few weeks of the project.

To get an indication of the weights of insertion, substitution, and deletion, we had other Giraf students sort a set of words in the order they would expect a good search system to sort them. The example search word was '*cat*'. Table 5.1 shows the ordering the seven students gave. Table 5.2 shows the ordering we will use as a target for a good search method.

The averaged position is calculated by taking the average of the position number of each word in each row. If a student did not include a word in their list, it is given a position of 12.

| | **Student 1** | **Student 2** | **Student 3** | **Student 4** | **Student 5** | **Student 6** | **Student 7** |
|---|---|---|---|---|---|---|---|
| 0 | Cat | Cat | Cat | Cat | Cat | Cat | Cat |
| 1 | Bobcat | Catalyst | Cataclysm | Bobcat | Hat | Bobcat | Bobcat |
| 2 | Hat | Cataclysm | Catalyst | Hat | Catalyst | Hat | Catalyst |
| 3 | Craft | Hat | Bobcat | | Cataclysm | Craft | Cataclysm |
| 4 | Hatter | | Abdicate | | Bobcat | Catalyst | Complaint |
| 5 | Catalyst | | Hat | | Abdicate | Cataclysm | Hat |
| 6 | Cataclysm | | Hatter | | Hatter | Center | Craft |
| 7 | Complaint | | | | Craft | Complaint | Hatter |
| 8 | | | | | Complaint | Hatter | Center |
| 9 | | | | | ABC | ABC | Tent |
| 10 | | | | | Center | Abdicate | ABC |
| 11 | | | | | Tent | Tent | Whig |
| 12 | *Others* | *Others* | *Others* | *Others* | Whig | Whig | Abdicate |

Table 5.1: Student orderings for the query '*Cat*'

| **Averaged position** | **Student 1** |
|---|---|
| 0 | Cat |
| 1.57 | Bobcat |
| 2.85 | Hat |
| 4 | Catalyst |
| 4.57 | Cataclysm |
| 7.85 | Craft |
| 7.85 | Hatter |
| 8.85 | Complaint |
| 9.5 | Abdicate |
| 10.28 | Center |
| 10.85 | ABC |
| 11.28 | Tent |
| 11.85 | Whig |

Table 5.2: Averaged student ordering for the query '*Cat*'

From table 5.2 we see some obvious tendencies. '*Cat*' should be first as it is exactly what was searched for. '*Whig*' should be last as it has not a single character in common with the query string.

Words which contain the query, like '*catalyst*', '*cataclysm*', and '*bobcat*' are generally placed high on the list. The exception here being '*abdicate*'. This may be because the pronunciation of '*abdicate*' is different from the word '*cat*'. '*Hat*' on the other hand may have been placed so high, exactly because it has similar pronunciation when compared to '*cat*'.

Words that contain all the letters of the query string but with letters inserted in between the letters of the query string, are generally placed in the middle.

Lowest on the list are words where the longest sub-string of the query string that can be found in the word is one or zero characters.

### Comparing Levenshtein Models

Table 5.3 compares the students ordering to three different variations of the Levenshtein distance. One of them being the standard $(i_1, s_1, d_1)$ model. A problem with this is that it favours short words, which is not preferable in our case.

A problem with $(i_1, s_{50}, d_{50})$ is that it favours words with more characters inserted into the query string, like '*complaint*', over words with just a single substitution like '*hat*'.

The model $(i_1, s_{4^x}, d_{50})$ tries to mitigate that, by allowing some substitutions. The idea is that the first substitution costs $4^0$, The second substitution costs $4^1$ and so on. This means that words like '*hat*' will get placed high while '*whig*' still gets heavily penalised. We still have the problem of words like '*craft*' and '*complaint*' getting placed higher on the list than we would like.

| **Student ordering** | | **Levenshtein distance** | | **Punishing Levenshtein distance** | | **Increasing punishment distance** | |
|---|---|---|---|---|---|---|---|
| Average of student scores | | $(i_1, s_1, d_1)$ | | $(i_1, s_{50}, d_{50})$ | | $(i_1, s_{4^x}, d_{50})$ | |
| 0 | Cat | 0 | Cat | 0 | Cat | 0 | Cat |
| 1.57 | Bobcat | 1 | Hat | 2 | Craft | 1 | Hat |
| 2.85 | Hat | 2 | Craft | 3 | Bobcat | 2 | Craft |
| 4 | Catalyst | 3 | ABC | 5 | Catalyst | 3 | Bobcat |
| 4.57 | Cataclysm | 3 | Tent | 5 | Abdicate | 4 | Hatter |
| 7.85 | Craft | 3 | Bobcat | 6 | Cataclysm | 5 | Catalyst |
| 7.85 | Hatter | 5 | Catalyst | 6 | Complaint | 5 | Abdicate |
| 8.85 | Complaint | 5 | Abdicate | 50 | Hat | 6 | Cataclysm |
| 9.5 | Abdicate | 4 | Center | 53 | Center | 6 | Complaint |
| 10.28 | Center | 4 | Hatter | 53 | Hatter | 8 | Center |
| 10.85 | ABC | 4 | Whig | 101 | ABC | 21 | ABC |
| 11.28 | Tent | 6 | Cataclysm | 150 | Tent | 22 | Tent |
| 11.85 | Whig | 6 | Complaint | 200 | Whig | 22 | Whig |

Table 5.3: Comparison of different orderings

The increasing punishment distance, as seen in table 5.3 gets us some of the way, but since we have already identified some of the important characteristics of what results are relevant, we can formulate a set of rules that dictate how to order the results. These rules are given below and consists of 4 rules and a number sub-rules for ordering. The first search results that are returned are all the results that start with the query string. If multiple results has this property the results are first ordered from shortest to longest and if they have the same length they are ordered lexicographically. After that we find all results which contain the query as a substring and likewise have a set of sub-rules for ordering. This set is then appended to the list of results which match the first rule etc.

1. Results which start with the query string:

   (a) Order shortest to longest.

   (b) Order lexicographically.

2. Results which contain the query string:
   (a) Order shortest to longest.
   (b) Order by how many characters come before the query string.
   (c) Order lexicographically.
3. Results which contain some characters of the query string in the correct order:
   (a) Order by the length of the longest common sub-string.
   (b) Order shortest to longest.
   (c) Order lexicographically.
4. Results which have no common characters with the query string:
   (a) Order lexicographically.

Table 5.4 shows the ordering of the words from table 5.1 according to the rule based method along with the rule number for which any specific word complies with.

| Rule | Word |
|---|---|
| 1 | Cat |
| 1 | Catalyst |
| 1 | Cataclysm |
| 2 | Bobcat |
| 2 | Abdicate |
| 3 | Hat |
| 3 | Hatter |
| 3 | ABC |
| 3 | Tent |
| 3 | Craft |
| 3 | Center |
| 3 | Complaint |
| 4 | Whig |

Table 5.4: Rule-based ordering

As explained above, we initially chose to implement what we call the punishing Levenshtein distance or $(i_1, s_{50}, d_{50})$, as we wanted to punish misspellings and deletions but wanted words that contained the search query as a sub-string to be prioritised highly. We later compared our chosen model to other orderings which indicated that the rule based ordering might be a better search method for Giraf. However, we did not have time to test the performance of the other approaches in depth, and therefore we leave the decision up to next year, leaving the current implementation as $(i_1, s_{50}, d_{50})$. This section documents our ideas on how search could be implemented without the need for external services like Elasticsearch [20].

### 5.2.4 Miscellaneous Tasks

In this sprint, we also implemented all the small tasks described in appendix C.2.2. Examples of these tasks include creating indexes on tables and enabling support for cross origin resource sharing (CORS). Some of these tasks, such as refactoring DTOs and making better authentication checks will also carry over to other sprints as these tasks are continuous.

### 5.2.5 Supporting other Groups

In this sprint we had a lot of interactions with both the server groups and the groups developing the WeekPlanner app. This was due to the groups now having finished the initial phases of understanding the domain and actually starting to write code and use the API. An example of an interaction could be that the server groups requested that instead of the API throwing exceptions when giving a wrong connection string, the error should be caught and pretty printed.

Furthermore, we changed the API such that the value specifying which environment to use no longer has to be specified in a `launch-settings` file but could be directly exported from the CLI. We also helped the server group correctly configure the settings file with respect to the newly implemented token authentication.

We also collaborated with the two front-end groups tasked with developing the WeekPlanner app in this sprint. We helped the front-end groups with fixing bugs, creating new features, and seeding the database with new data. They requested that we changed some of the error-codes, as these were inaccurate and sometimes confusing. Consider the example in listing 5.1 that checks if a given username exists and returns an `InvalidProperties` error if no user was found. The problem here is that the error code is confusing if for instance a username is provided, but does not exist in the database. In this example, the user would get an `InvalidProperties` error, even though the username is a valid username (It is simply not registered). This particular problem was fixed by simply changing the error code to `InvalidCredentials`, rather than `InvalidProperties`.

```
var loginUser = await _giraf.LoadByNameAsync(model.Username);
    if(loginUser == null) // If username is invalid
        return new ErrorResponse<GirafUserDTO>
                                (ErrorCode.InvalidProperties,
                                   ↪ "username");
```

Listing 5.1: Example of confusing ErrorCode

We also developed support for getting images as their raw format (png) instead of base64 encoded, as well as support for getting all citizens under a department based on requests from the front-end groups. Furthermore, we helped the front-end groups using the API, including tasks such as properly deserialising responses and correctly converting a base-64 image into a displayable image object.

With respect to the collaboration in this sprint, one can say that we had a supporting role, where our primary role, beside the technical stories, was to ensure that the front-end and server groups could carry out their tasks easier and that any bugs or lacking features was fixed or implemented as quickly as possible.

## 5.3 Test

One of the primary focuses this year is to make the WeekPlanner stable. To ensure high stability, each component must be properly tested. In this section, we describe how we added integration tests in order to test the API, as well as the improvements made to the unit tests to ensure that the endpoints work as expected.

### 5.3.1 Integration Test

The purpose of integration tests is to test the dependencies that our unit-test does not test, such as access to the real database and authorisation. This type of test has the advantage that we are able to discover additional run-time errors like for instance database constraints.

In *The Art of Unit Testing* [21], Roy Osherove defines integration tests as follows:

"Integration testing is testing a unit of work without having full control over all of it and using one or more of its real dependencies, such as time, network, database, threads, random number generators, and so on."

Here, a 'unit of work' is a use case within the system and can be everything from a single method to a series of methods and classes that serve some purpose.

Initially, we tried to use Python without looking into specialised testing libraries, but instead implemented our own integration-test framework. We quickly found problems like catching exceptions mid-test, test-dependencies, and proper error-codes to be difficult without extensive development of our own framework. Therefore, we looked into existing frameworks and chose Integrate [22], which is available using pip [23]. We then rewrote the tests from our own framework and continued work.

Some of the things we test are the token-authentication and that the different access-levels work as expected. We also test the connection to the database, to ensure that the actual database, and not only the mock database, works as intended. This proved valuable as the integration-tests found issues that the unit tests did not like for instance authorisation errors. Many endpoints first checked if the requested item was present before testing the supplied JWT-token, so a malicious person could gather a lot of information about e.g. which IDs are in use in the database and which users are signed up. We also found errors where endpoints returned more data than needed or incorrect data.

An example of a test that uncovered one of these authorisation issues can be seen in listing 5.2 where the test makes a GET request to the `Department/{ID}/citizens` endpoint and ensures that we return the `NotFound` error-code, which indicates that the authorised user does not have access to this endpoint, or that it does not exist. This endpoint would return `DepartmentNotFound` if a department with that ID did not exist, which revealed information about the database to unauthorised users.

```python
@test()
def callGET_Department_id_citizensNoAuthNoBody(self, check):
    "Testing authorization of GET /v1/Department/{id}/citizens"
    response = requests.get(Test.url + 'Department/0/citizens').json();
    check.is_false(response['success'])
    check.equal(response['errorKey'], "NotFound")
```

Listing 5.2: Integration test that checks the returned errorkey

### 5.3.2 Unit test

At the start of this sprint, most unit tests only checked the state of `Success`-flag and not the returned data for errors. An example of such a unit test can be seen in listing 5.3.

```
[Fact]
public void UpdateDay_InExistingWeek_Ok()
{
    var wc = initializeTest();
    _testContext.MockUserManage
                .MockLoginAsUser(_testContext.MockUsers[USER_0]);
    var day = _testContext.MockWeeks[USER_0_DAY]
                          .Weekdays[(int)DayOfWeek.Monday];
    day.Elements.Add(new WeekdayResource(day,
       ↪ _testContext.MockPictograms[PUBLIC_PICTOGRAM]));
    var res = wc.UpdateDay(day.Id, new WeekdayDTO(day)).Result;
    Assert.True(res.Success);
}
```

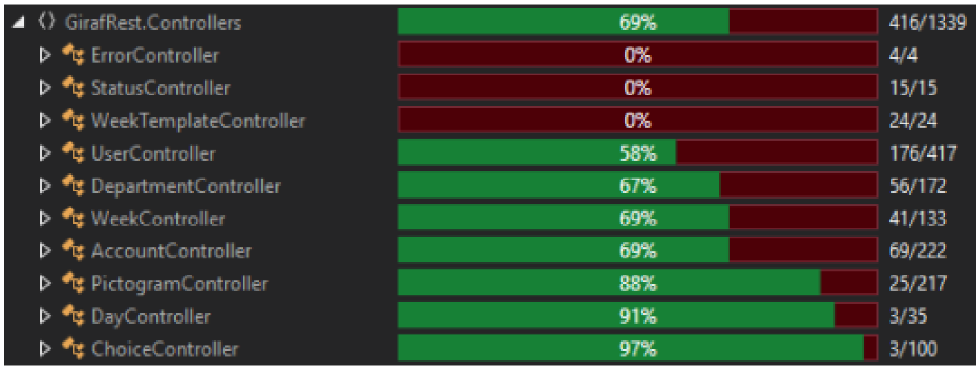
Listing 5.3: Unit test example that only checks if response is successful

As tests should exist for finding bugs and wrong results, we updated all unit tests to include checks for correctness of the affected data. This helped us to discover additional bugs in the API. For instance, when updating a day in a week, the different resources like choices and pictograms were not added to the weekday.

### 5.3.3 Test Coverage

As we have used some time in this sprint and last sprint to improve and write new unit tests, it is interesting to see how much of the code that these tests actually cover. We are mainly interested in the code in the various `controller` classes, as this is the code that gets executed whenever request to the API is made. To make a test coverage analysis, we used the 'dotCover' tool from JetBrains [24].

As seen in figure 5.3, 69% of the code blocks in the controller code is covered in the unit tests. No tests currently exist for testing the role and error controllers as these controllers have not been directly part of the user stories covered in this or the previous sprints, and has therefore not been prioritised. The exact commit on which the test coverage is performed can be found in [25].



| | | |
|---|---|---|
| GirafRest.Controllers | 69% | 416/1339 |
| ErrorController | 0% | 4/4 |
| StatusController | 0% | 15/15 |
| WeekTemplateController | 0% | 24/24 |
| UserController | 58% | 176/417 |
| DepartmentController | 67% | 56/172 |
| WeekController | 69% | 41/133 |
| AccountController | 69% | 69/222 |
| PictogramController | 88% | 25/217 |
| DayController | 91% | 3/35 |
| ChoiceController | 97% | 3/100 |

Figure 5.3: Code coverage of unit tests on all controller components

## 5.4 Conclusion

As stated in section 5.1, the overall goal of the second sprint did not differ a lot from the overall goal of the first sprint. As the functionality of the backend already supported the user stories for this sprint, we created our own technical stories. These stories were created primarily to ensure stability and intended functionality requested by other groups, or decided upon ourselves.

Changes for the backend in this sprint included changing the development database from using SQLite to MySQL, and using EF Code First, as described in 5.2.1. Another task was switching from cookie authentication to token authentication, using JWT. This was done to refrain from having the server manage the state of the user, but instead place this responsibility on the client. As seen in appendix C.2.2, this sprint included several minor miscellaneous tasks such as setting up specific endpoints, creating indexes on tables for faster look ups, and changing DTOs to only return necessary information.

Another focus in this sprint was to improve the testing aspect of the backend. This included updating unit tests, so we did not merely check the response type and error code, but also the actual data returned from the response. Integration tests were also developed using a Python script executing a series of requests. The integration tests were able to find errors and bugs that unit tests could not, as unit tests were not able to check connection to the production database, or properly test token authentication.

The state after this sprint is that the backend supports all the user stories described in C.1 and that we managed to implement the technical tasks and miscellaneous tasks described in C.2.

Figure 5.4 shows the general architecture after sprint 2, where a component with an outgoing arrow is dependent on the component the arrow points to. The primary focus of the frontend groups in this sprint was to rewrite the WeekPlanner application to be cross-platform. The state of the WeekPlanner is now that users are able to log in and add, create, and modify week schedules.

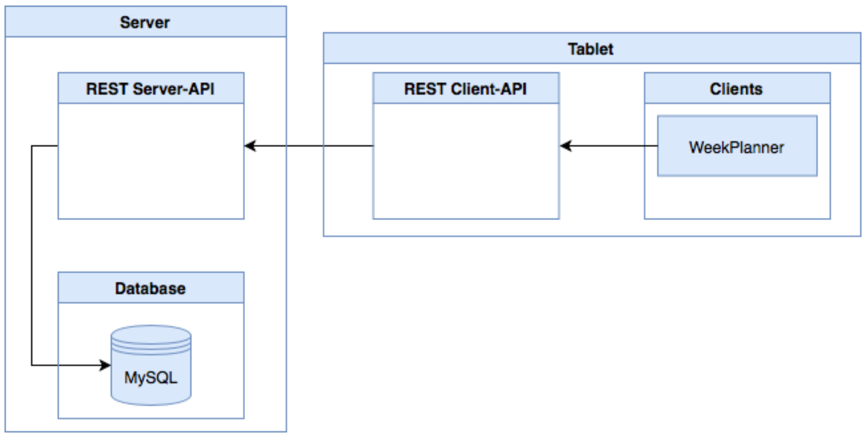


Figure 5.4: The general architecture after sprint 2

# Chapter 6

# Sprint 3

This chapter describes the third sprint of the Giraf project. The purpose of this sprint is to implement support for user settings, support week templates in the database as well as add support for activities and choices.

## 6.1 Planning

This section presents the user- and tech-stories in the backlog relevant for the backend in the third sprint of the Giraf project.

Most of the user stories in sprint 2 were completed, but not all got accepted by the stakeholders due to minor bugs and issues, primarily in the WeekPlanner app which means that some tasks from the sprint 2 backlog are carried over to sprint 3.

### 6.1.1 Tasks

The tasks relevant for the backend with respect to both user stories and technical stories can be seen in appendix C.3 and include tasks such as implementing user-specific settings and modifying the database structure for week schedules, week templates, and choices.

#### Time Span

The third sprint was planned on the 12th of April, and lasted until the 7th of May. We had 25 available half-days to work on the project in this sprint, which gives us $4 \cdot 25 = 100$ hours per person.

### 6.1.2 Process Model

Many of the issues that occurred in sprint 2 were due to the small amount of time used testing the app, as front-end features were integrated the evening before the acceptance test with the users. Due to these issues, we took the initiative to improve the development process used, and created the IWWP process model described in section 3.2.

In the new process model, two releases were scheduled for this sprint and requirement specifications for each release were written. The specifications are to find at [26] and are named `2018S3R1` and `2018S3R2`, respectively.

## 6.2 Development

This section presents the implementation of some of the user- and technical stories mentioned in section 6.1.

### 6.2.1 Migrating old production-database

As new requirements were discovered, the database structure of the backend was changed, and so the production database had to be updated in order to work with the newest version. The standard way to do this would be to update the migration scripts generated by EF, so that data from the production database could automatically be converted into the new structure. However, considering the vast amount of migration files left unchanged from the previous semester, and the fact that nothing was currently in production, we decided to write a SQL-script for porting the relevant data and starting over with migrations. The script is named `migrate.sql` and can be found on the Git repository at `https://gitlab.giraf.cs.aau.dk/Server/web-api/blob/2018S4R1/scripts/migrate.sql`.

### 6.2.2 Restructuring Entities for Weeks

Before this sprint, there where no attributes on a `Week` schedule describing when the schedule is for. To be able to determine which week schedule is the current and which week schedule is next, we alter the database structure to support that week schedules can be uniquely identified based on the user, year, and week number.

Figure 6.1 shows the ERD diagram related to week schedules after sprint 2 had been completed. As illustrated, a `Week` can either be an ordinary `Week` or `WeekTemplate`, where a `WeekTemplate` is a generic `Week` that defines the structure for creating new `Weeks`. A `Week` contains seven `Weekdays` and is related to a `User`. A `Weekday` contain a list of `WeekdayResources`, which represent either `Pictograms` or `Choices`.

With the addition of weekyear and weeknumber attributes, the ERD diagram in figure 6.1 has two major problems. The first is that `WeekTemplates` and `Weeks` can no longer be related the way they are, as a `WeekTemplate` should not have a year and week number and not be related to a user. The second is the way that the group from previous years decided to model `Choices` in `Week` schedules. As defined in chapter 2, a `Choice` is an option between one or more `Activities`. As illustrated in figure 6.1, `Choices` are currently related to `Weekdays` through the `WeekdayResource` entity, which is a pivot table for describing the many to many relation between `Weekday` and `Resources`.

This allows for a `Week` to have many `Pictograms` and `Choices`, but since a `Choice` contains a relation to `ChoiceResources`, which is the pivot table for modelling a many-to-many relation between `Choices` and `Frames`, this gives rise to a recursive structure where a `Choice` may contain another `Choice` and even contain itself.

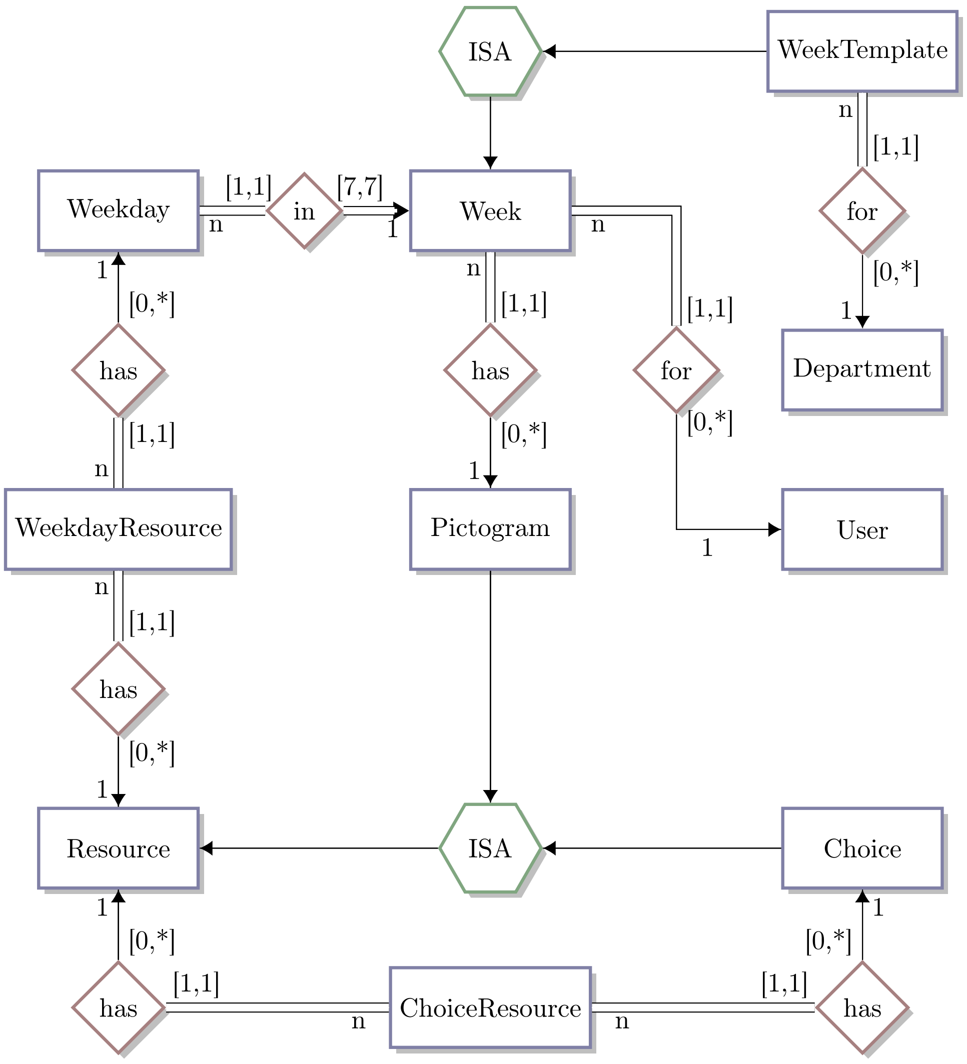


Figure 6.1: Entity Relationship Diagram for Weeks after sprint 2

Because of the problems highlighted above, we decided to restructure the database; the result can be seen in the ERD diagram in figure 6.2.

Here, `WeekTemplates` and `Week` schedules now both inherit from a `WeekBase` entity which contains all the common relations. Furthermore, only a `WeekTemplate` has a title while a `Week` contains a year and week number. While a `WeekTemplate` is associated with a `Department`, a `Week` is associated with a specific `User`.

To model `Choices`, we deleted the `Choice` and `ChoiceResource` entities due to the problems with this structure. Instead, we opted to create an `Activity` entity which is related to a `Weekday` and contains a number that represents its ordering within the day.

This structure has two advantages over the old; it allows us to support `Choices` without having to create extra entities, as a `Choice` can simply be modelled as two or more activities on a weekday with the same order index. At the same time, an order index allows us to guarantee that the `Activities` are retrieved in the order that they were created, which is something we could not do in the old database structure.

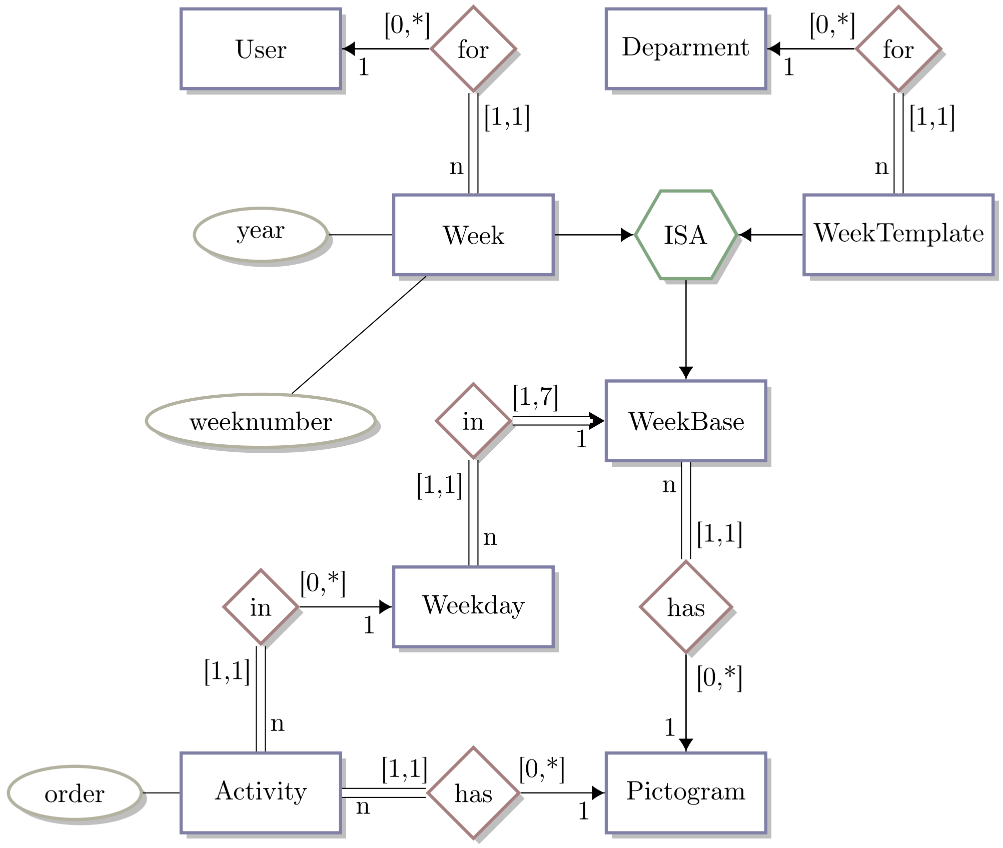


Figure 6.2: Entity Relationship Diagram for Weeks after sprint 3 including new attributes

### 6.2.3 Marking activity as done

A goal of this sprint was letting `Citizens` mark the `Activities` in their `Week` schedules that they have completed, in order to get an arrow to point to the one they should currently be doing. At the same time, the PO group asked for the additional option of marking an `Activity` as cancelled.

We added a new attribute to the `Activity` table, as well as adding support for setting and getting this attribute in the existing endpoints.

### 6.2.4 Implementing user settings

The PO group asked for several new individual settings in order to meet the different requirements of citizens. The new list of settings, as agreed upon by all frontend groups, became:

- Whether to dynamically change the tablet's orientation, or stay locked.
- Whether and how to display completed activities.
- Whether and how to display cancelled activities.
- Default style of the timer.
- Default timer value.

- Number of activities to show at once.
- Number of days to show at once.
- Whether to show everything in grey scale.
- General colour theme to use.
- Colour theme for week schedules.

To implement the above settings, we discussed two possible approaches. Both approaches involved a separate `Settings` table in the database, which would have a relation with a `User`. The first approach was to have a simple table with three columns; `id`, `key`, and `value`. This approach has the benefit that it allows the clients to easily add new settings by simply adding a new key-value pair and implementing support for this through an endpoint in the API. Furthermore, it would be easy to rename keys through endpoint support.

However, it has the disadvantage that it is too generic, which makes it difficult to do validation checks for the clients, because everything is stored as strings as well as preventing the database system from ensuring that the same information is not stored in different key-value pairs.

The second approach is to use value types like enums, integers and so fourth to hold the relevant information for the settings. This has the advantage that we, in the backend, can make different validation checks before storing the data in the database. The main disadvantage is that it is more difficult to extend, as it requires a new column in the settings table for each new setting.

After a discussion with one of the front-end groups, we chose the second approach, as this is easier for them to work with, because the DTO generated by Swagger signifies the exact different attributes of the settings and their respective type. The first approach would only have been better if each `User` should be able to have different settings or if they were to change a lot. However, this is not the case.

Initially, colour-theme for `Week` schedules was implemented as a simple enum, as this was how the initial requirements for the release specified the task. However, one day before the deadline for the release, the PO group requested that it should be possible to change the colour of each day of a week as a setting instead of one colour for the whole week. The default colour of the days should correspond to those in figure 2.2.

To implement this, we added a `WeekDayColour` entity which contains an attribute for storing the colour as a hex-string. An enum which indicates the day of the `Week` and a foreign key to the `Setting` entity. Furthermore, we added a simple regular expression to check if a given string is a valid hex-string: `#[0-9a-fA-F]{6}`.

### 6.2.5 Logging Requests to the API

During a sprint meeting it was requested that a logging system should be implemented for requests sent to the API.

The purpose of the logging system is to implement some basic foot-printing for the users of the API. For instance, the logging system should log if a guardian attempts to fetch information about a citizen. As no specific requirements were given from the stakeholders at this time, we decided to implement a simple logging system that logged to a file, which later could be extended to a more advanced system that logged to a database.

To implement the logging system, we created a `LogFilter` class which inherits from `IActionFilter`. `IActionFilter` is an ASP.NET interface that defines methods to be executed before and after controller endpoints are called [27]. By implementing the `LogFilter` as an `IActionFilter` it allows us to log information after each request to the API. In the current implementation of the API, guardians are allowed to log in as citizens and change their information. One could say that guardians are allowed to impersonate a citizen. From a logging perspective this is a problem as there is no way of telling who is actually sending requests on behalf of the user.

To solve this problem of not being able to log if a user is impersonating another user, we added an `impersonatedBy` claim to the JWT user token, which states who is actually making the requests. A pseudo code implementation of the logger is seen given in algorithm 1 as well as an example output in listing 6.1

**Algorithm 1** LogFilter for logging request to the API

```
procedure OnActionExecuted(context)
    path ← "Logs/log" + year + month + day
    userId ← GetUserIdFromContext(context)
    byId ← GetImpersonationClaim(context.Claims)
    user ← FindUserInDatabase(userId)
    by ← FindUserInDatabase(byId)
    url ← GetUrlFromContext(context)
    verb ← GetVerbFromContext(context)
    error ← GetErrorFromContext(context)
    PlaceFileLock()
    File.Append(path, "{Timestamp}; {by}; {user}; {verb}; {p}; {error}; {byId}; {userId}")  ▷ write to file
    RemoveFileLock()
```

```
2018-05-17T12:23:39.8923273Z; Graatand; Kurt; GET; /v1/Week/2018/20;
    ↪ NoError; e5d4130a-95f1-4917-9397-a8d220d3ab84; 9fe4612d-5a74-4b93
    ↪ -b27b-5aa7d7994bf4
```

Listing 6.1: Example of a line written by our `LogFilter`, where Graatand is impersonating Kurt.

## 6.3 Conclusion

The goal for this sprint was primarily to implement user specific settings, like support for customising colours on week schedules. Furthermore, an internal goal was to restructure week-schedules and activities in the database to give a more sound database structure.

Status after this sprint is that we managed to implement all the tasks seen in appendix C.3.2 and C.3.3, which means that the API and database now, among other things, support user settings, week templates, and choices as well as unique identification of week schedules based on user, week number, and week year. Furthermore, we also included a simple `LogFilter` that logs all requests to the API.

# Chapter 7

# Sprint 4

This chapter describes the fourth and final sprint of the 2018 Giraf multi-project. As seen in 7.1.1, the time span for this sprint was shorter than usual. This is because other sprints were prolonged, either to finish features before an acceptance test was conducted, or simply because the stake-holders had to postpone the acceptance test meeting.

## 7.1 Planning

During the planning meeting for sprint 4, it was decided that this sprint should only contain a single release, due to the time available. Additional time should be used for documenting our work to next year's students.

The full specification of user stories planned for this sprint can be seen in appendix C.4 and the wiki-page for the release, which contains deadlines and requirements, can be found at `http://web.giraf.cs.aau.dk/w/releases/2018s4r1/`.

### 7.1.1 Tasks

The purpose of sprint 4 is to increase guardian accessibility in regards to the WeekPlanner app.

There is a need for handling the administrative part of the application such as creating, updating, and deleting users, guardians, and departments and password recovery/reset. To refrain from having to fit this in the existing app, it was decided to create a web-based administration panel instead. This required expanding the `AccountController` to support these administrative tasks. Also included in this sprint is the technical backend related tasks (described in C.4.3), such as query optimisation.

#### Time Span

The final sprint of the project spans from the 8th of May to the 17th of May, which was decided to be the day of code stop, in order to have enough time to finish the written report. This corresponds to 7 working days, which equals 56 hours per person.

## 7.2 Development

In this section, we start out by giving an overview of the implementation of the Giraf administration panel, which we continue with a description of the changes made in the API. This includes changing the authentication controller to support the features needed in the administration panel as well as some query optimisation.

### 7.2.1 Giraf Administration Panel

As part of this sprint, we were assigned the task of developing an administration panel to manage citizens, guardians, and departments. The purpose of the administration panel is to enable the customers and developers to setup and manage users, without calling endpoints directly. This is essential to initiate the use of the WeekPlanner application.

For guardians, the admin panel should enable them to add, edit, and delete citizens within their department as well as adding another guardian to their department. To support multiple departments, the admin panel should enable a superuser to add, edit, and delete departments. Finally, the admin panel should also enable all users to change or reset their password.

The administration panel will be implemented as a web based application. We will try to implement a responsive user interface meaning that the application can be used on devices with different screen sizes. This will enable both mobile and desktop use. We will implement the administration panel using Vuetify.js because it enables us to implement a responsive UI. Vuetify.js also comes with a variety of UI components like buttons, text fields, and data tables [28]. Another advantage of Vuetify.js is that it builds on top of Vue.js which supports two-way data bindings between the model received from the API and the properties displayed in the UI. To manage users and departments, the administration panel will consume the API and call the `AccountController`, `UserController`, and `DepartmentController`.

#### Login Page

Before we can enable a user to manage other users through the admin panel, we first need to authenticate the user. This is achieved by adding a login page, where users enter their credentials as shown in figure 7.1.

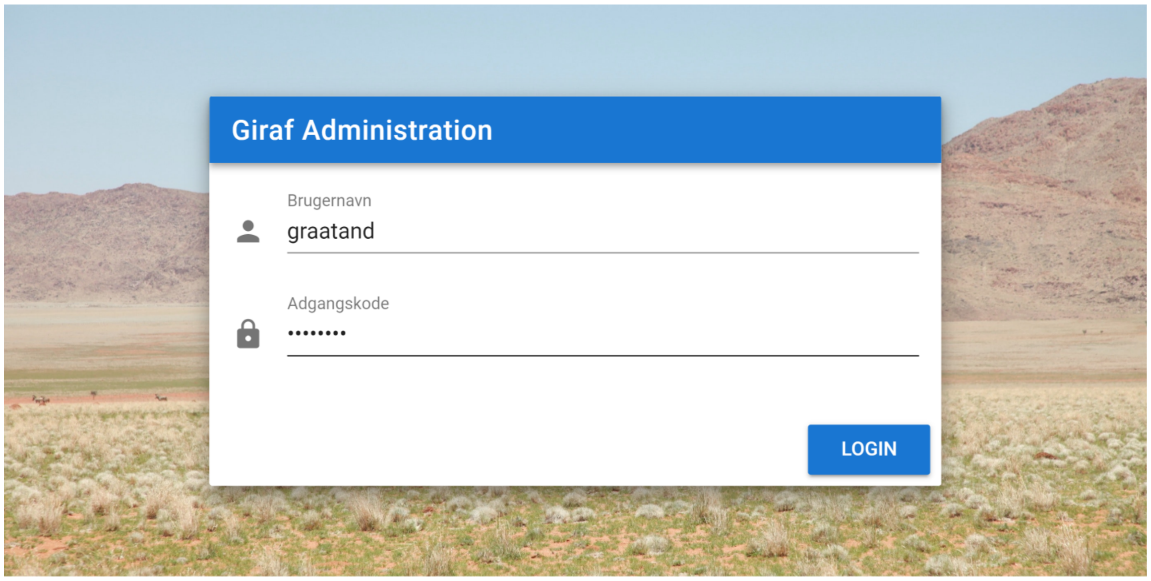


Figure 7.1: Log-in page in the Giraf Administration Panel

The login form is implemented based on the example provided in [29]. Below the login form, we added an information box to inform users if they have entered invalid credentials or try to log in as a user which is not authorised to access the admin panel.

### Managing Citizens as a Guardian

After a guardian has been authenticated by using the login page, the citizen page is shown. The citizen page can be seen in figure 7.2, which shows the list of all citizens within the same department as the currently authenticated user. On top of the list, we added a search bar to make it easier for users to find a specific citizen.

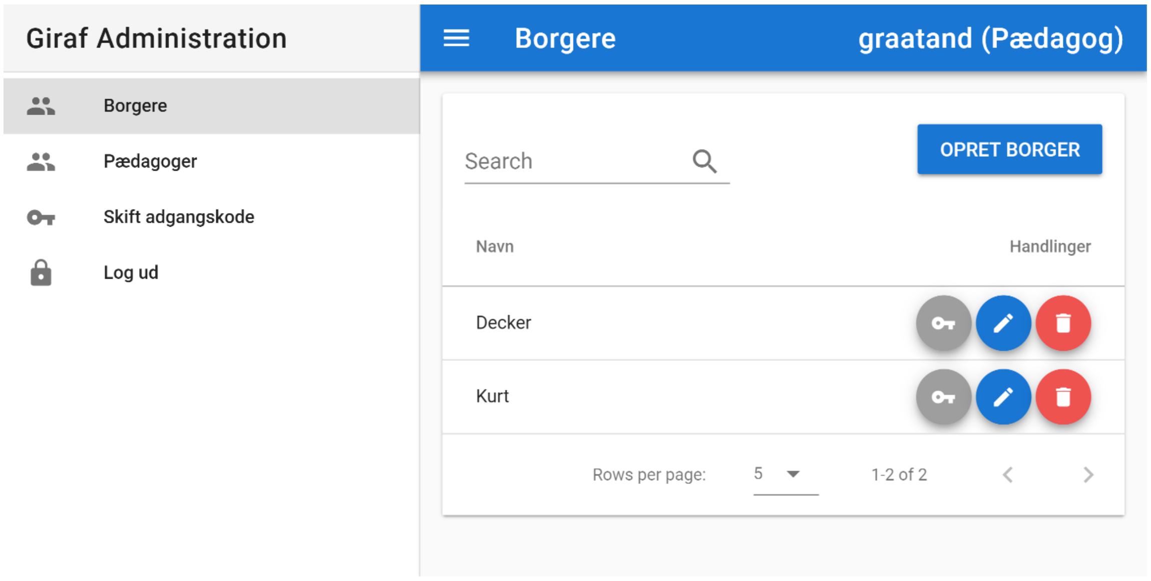


Figure 7.2: Citizen page showing the list of all citizen within the department of the authenticated user

To the right of the search bar, there is a button to add a new citizen to the system. Pressing this button will present the dialogue shown in figure 7.3.

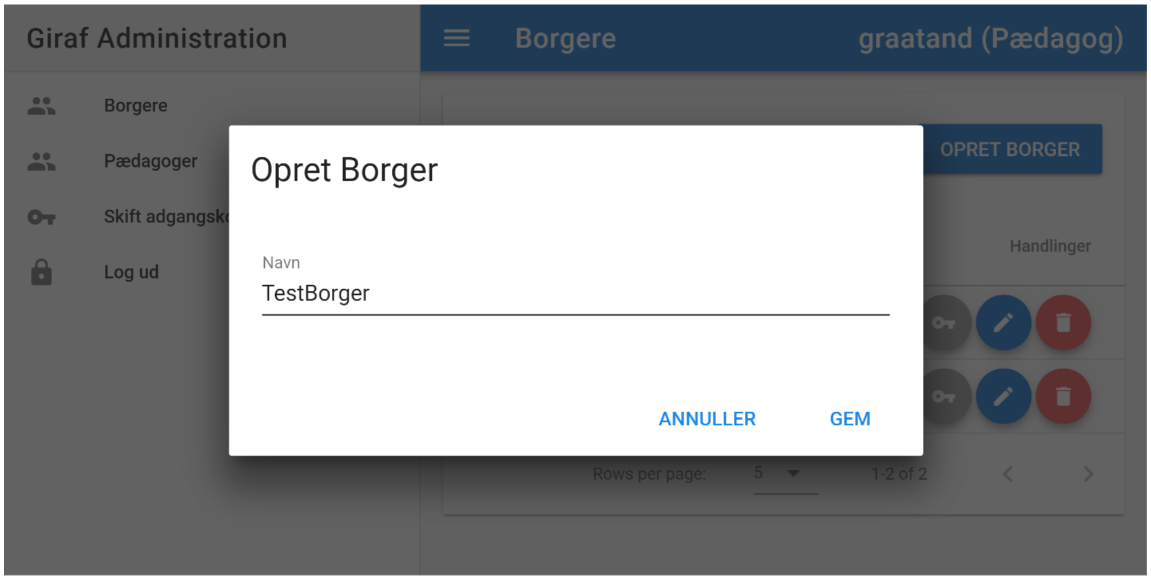


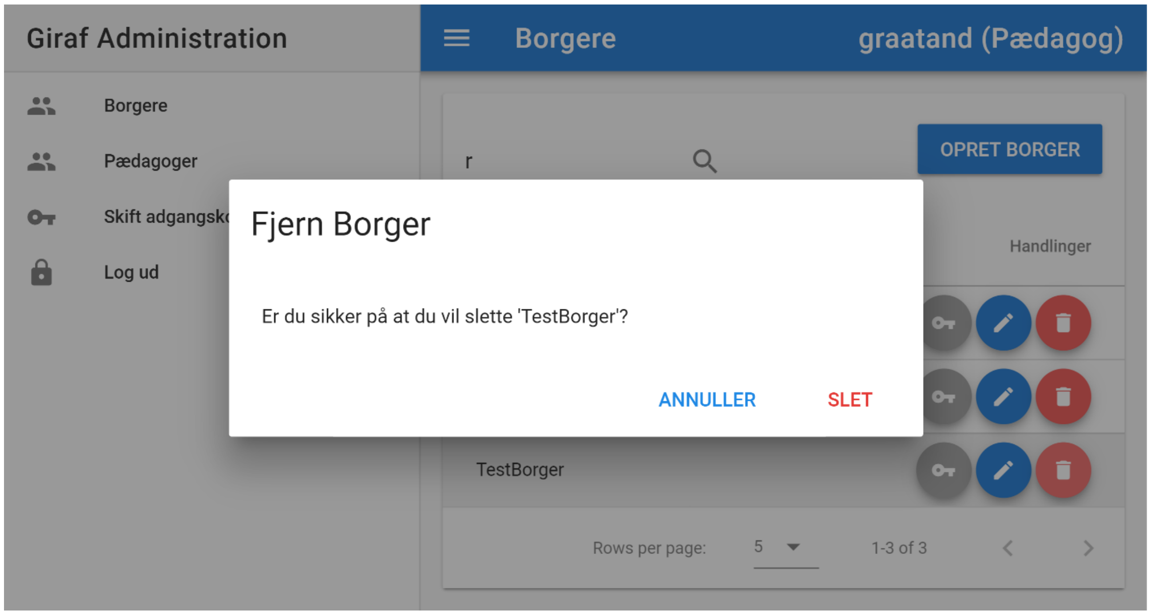
Figure 7.3: Dialogue to add new citizen

There are three actions which can be performed on an existing citizen.

The first action is symbolised with a key icon. This action generates a reset password link which will be described later.

The next action is showing a dialogue like a dialogue in figure 7.3, used to edit the information about the citizen i.e. the name of the citizen, in case it was typed incorrectly into the system.

The third action is symbolised with a red button with a trash bin icon. This action shows the delete dialogue as shown in figure 7.4 to remove the citizen from the system.



Er du sikker på at du vil slette 'TestBorger'?

Figure 7.4: *Delete* dialogue to confirm deletion of citizen in the system

## Managing Guardians

Managing guardians in the admin panel is similar to how citizens are managed, as shown in figure 7.5.

In the implementation of the UI for managing citizens and guardians, we decided to create a common component for both citizens, guardians, and departments. One of the benefits of this is that we reduce the amount of code to write and maintain. By using a common component, we also obtain a consistent user experience across the citizen, guardian, and department page which should make it easier for users to comprehend the system, compared to if we had separate ways to manage citizens, guardians, and departments.

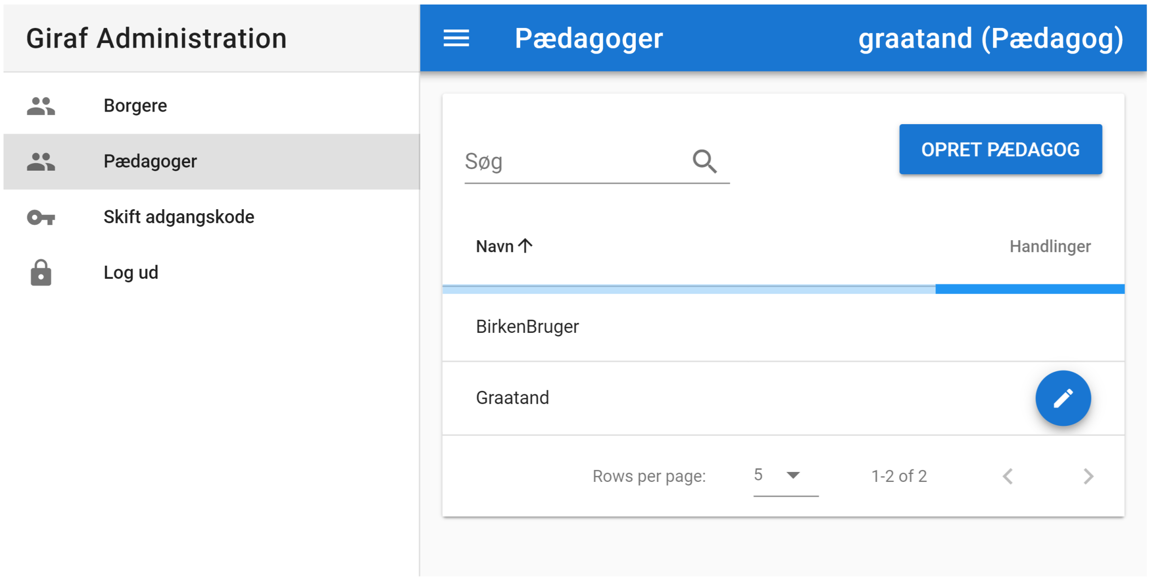


Figure 7.5: Guardian page showing the list of guardians in the same department as the authenticated user. Showing that guardians have access to add a new guardian or edit their own name.

The blue line above the list of guardians is a loading indicator. We added this to the common component to be visible whenever data is being loaded from the backend API. This is useful if the internet connection speed is slow, as the user could otherwise be led to believe that there is no data available. The effect of this is, of course, depending on how familiar the user is with this type of activity indicator.

### Managing Departments as a Super User

If a superuser (administrator) logs into the admin panel, the department page will be shown as demonstrated in figure 7.6. This page enables super users to add, edit, and delete departments.

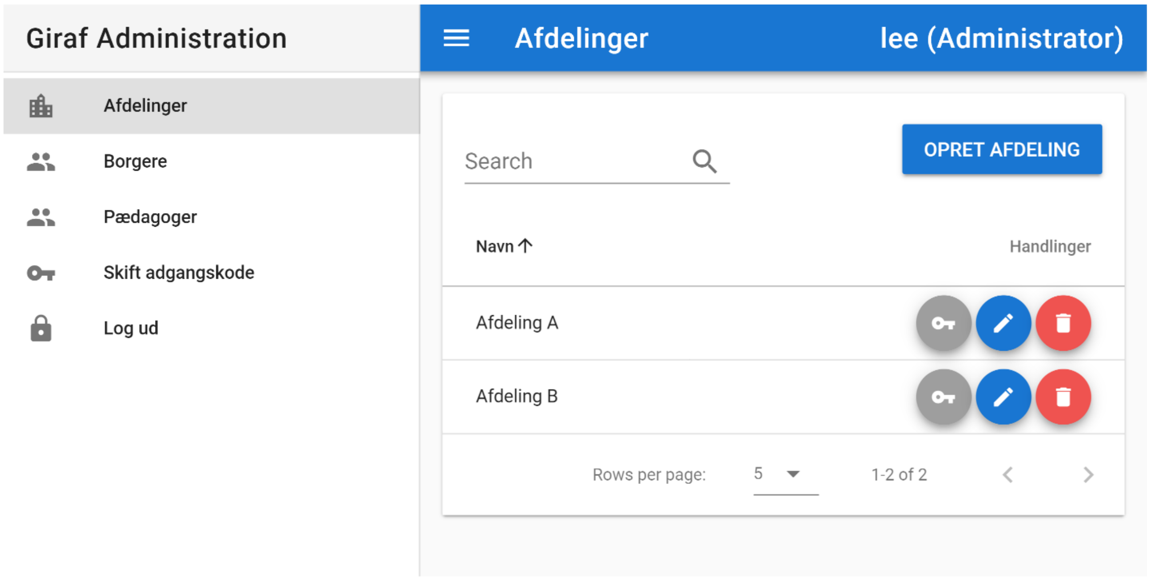


Figure 7.6: Department page showing all departments

The department page uses the same common component as the citizen and guardian page. The department page helps to support that new departments e.g. kindergartens can begin using Giraf.

## Managing Citizens and Guardians as a Super User

Because superusers are not related to a specific department, the citizen and guardian page contains a select menu, to let the superuser choose which department they are currently managing as shown in figure 7.7.

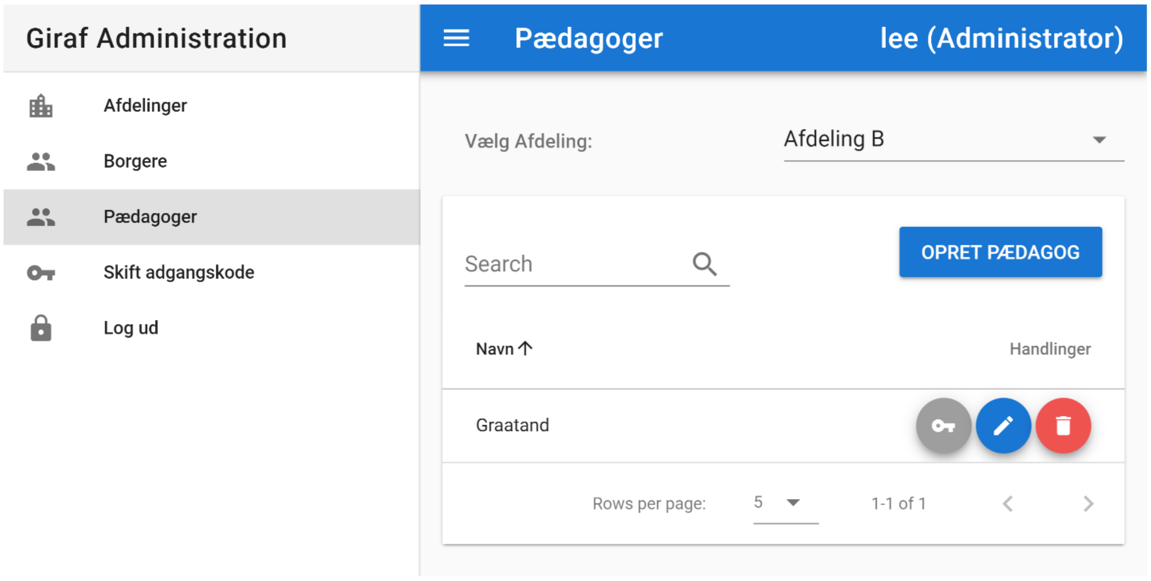
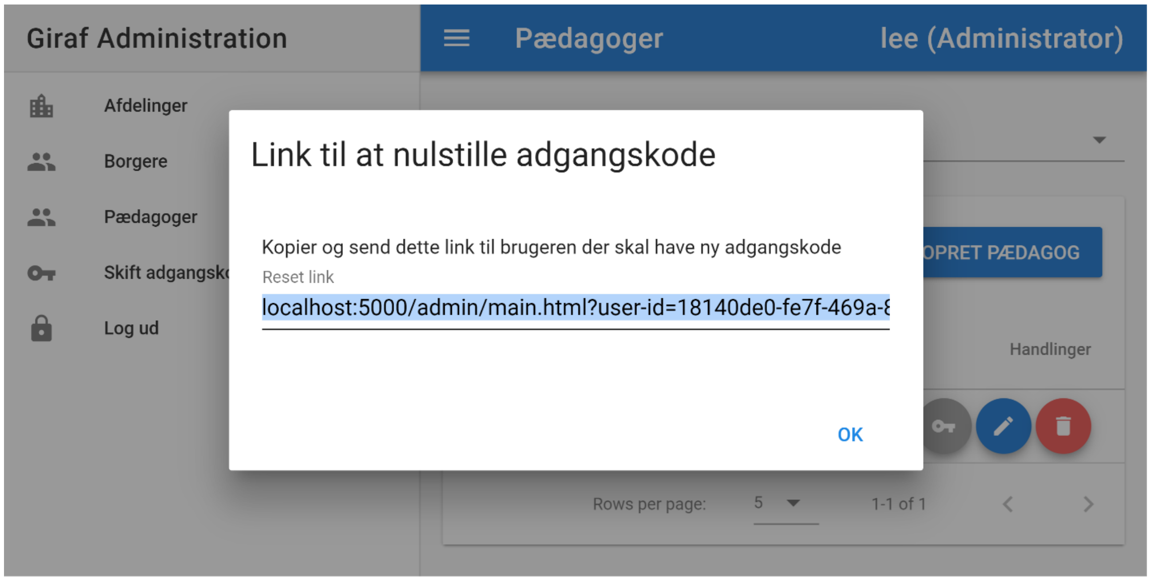


Figure 7.7: Guardian page including department select menu when logged in as super user

In figure 7.7, we see an example of a department with only one guardian. If that guardian forgets their password, the superuser can log in to the admin panel and generate a reset password link for the guardian, by tapping on the key icon to the right of the guardian.

### Generating a Reset Password Link

After the generate password action has been tapped, a dialogue appears as shown in figure 7.8. This dialogue contains a link which can be sent to the guardian who has forgotten their password.



Link til at nulstille adgangskode

Kopier og send dette link til brugeren der skal have ny adgangskode

Reset link

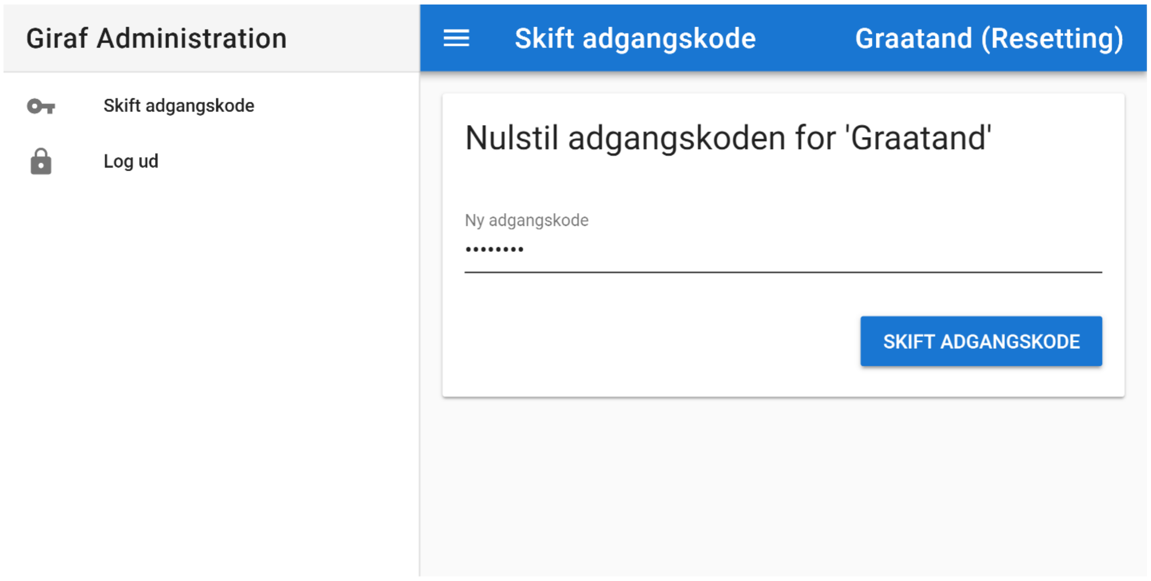
localhost:5000/admin/main.html?user-id=18140de0-fe7f-469a-8

Figure 7.8: Dialogue showing a generated reset password link

A reset password link, like the one shown in figure 7.8, navigates to the reset password page. Encoded in the link is a reset password token, which ensures that only users which are authorised can generate a valid reset password link. It also ensures that the link only can be used to reset the password once. If a user forgets their password again, a new link must be generated.

### Resetting the Password

If an unauthenticated user navigates to the admin panel through a reset password link, they will see the reset password page as shown in figure 7.9.

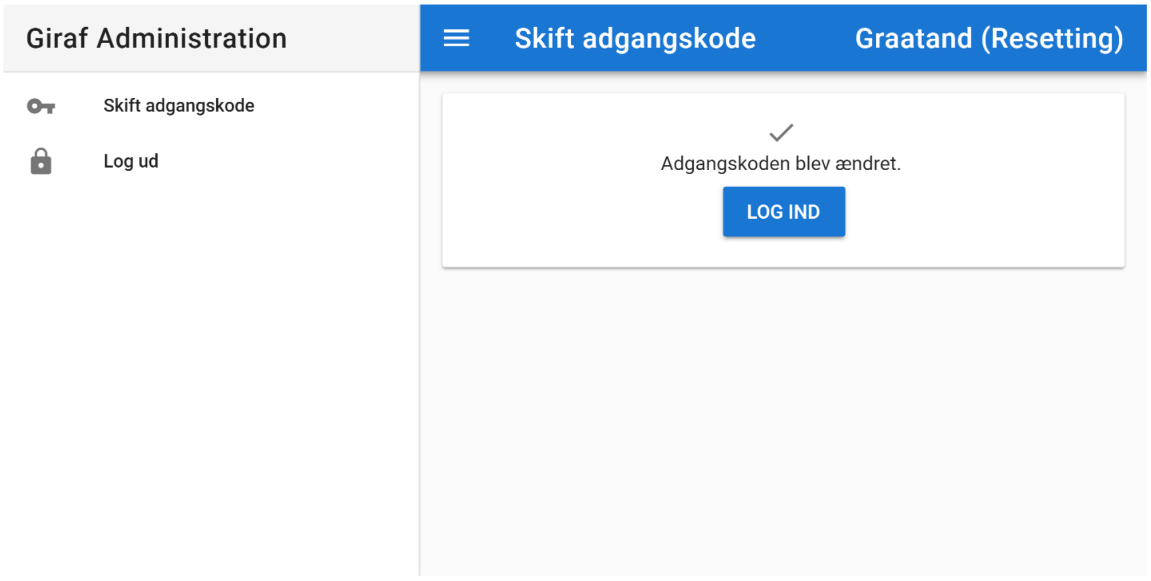


Nulstil adgangskoden for 'Graatand'

Ny adgangskode
••••••••

Figure 7.9: Reset password page enabling a user to type in a new password

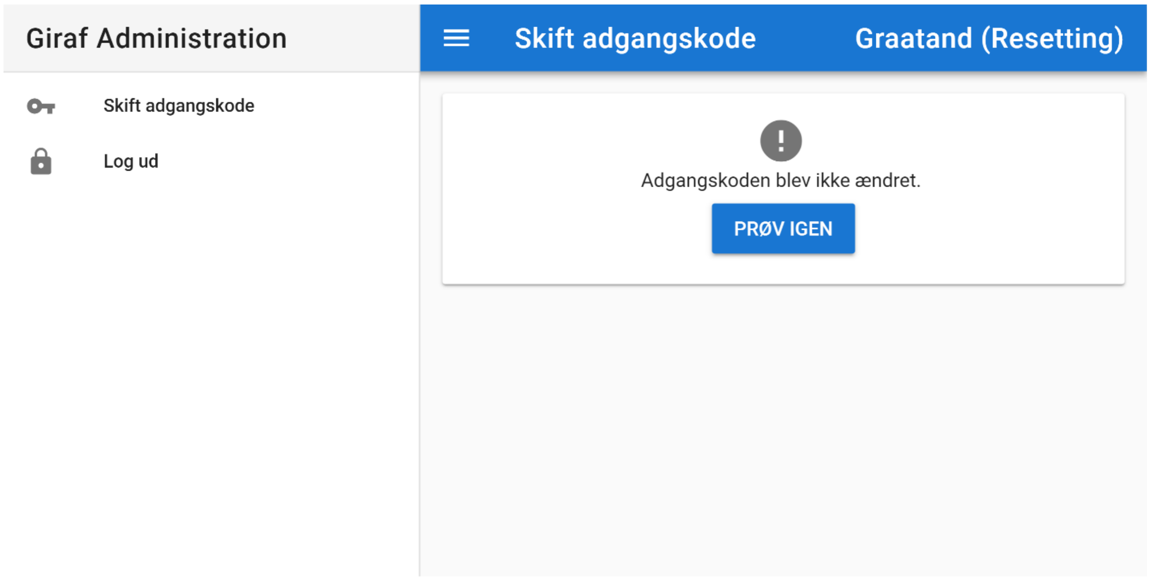
When navigating to the admin panel through a reset password link, the citizen, guardian, and department tabs are not accessible. This is because the user is not actually authenticated and should therefore not be able to change anything other than their password. After the user has typed in their new password and taps the change password button, a message will appear as shown in figure 7.10 or figure 7.11, informing the user that the password was changed successfully.



Adgangskoden blev ændret.

Figure 7.10: Message describing that the password has been changed successfully

If the password has been changed successfully, the user will be informed of this, as shown in figure 7.10. A login button is added below this message to make it easy for users to navigate to the login page, where the user can log in with their new password.



Adgangskoden blev ikke ændret.

Figure 7.11: Message describing that the password has not been changed

If the password was not changed, the message in figure 7.11 will appear.

There are two actions that trigger the message shown in figure 7.11. One action is if the user types in a password which is not valid according to the password requirement of the backend.

The second reason is if the user tries to reset the password by using the same reset password link multiple times. The message in figure 7.11 does not explicitly state the reason why the password was not changed. This could be added in a later release version of the admin panel to make it easier for the user to understand the cause of the error.

### Changing the Password

Besides resetting the password by a reset password link, it is also possible for authenticated users to change their own password if they know their current password. The possibility to change password is important when considering security in the Giraf system, if a users login becomes compromised.

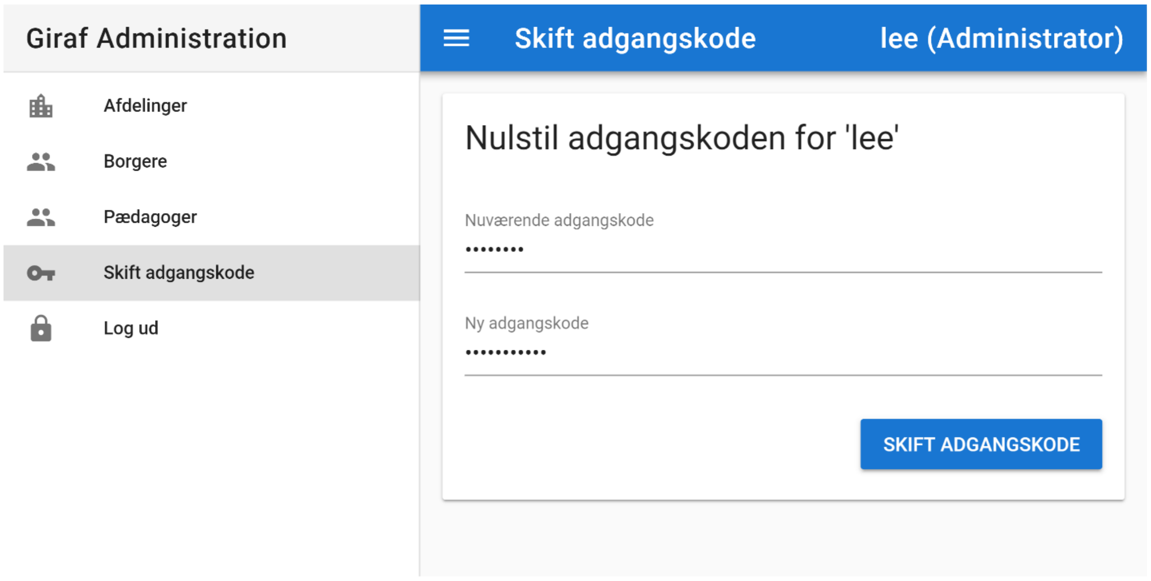


Figure 7.12: Change password page containing text input field for the user's current password and a text input field for the new password

After tapping the change password button, the message in figure 7.10 or 7.11 will be shown similarly to the behaviour of the reset password page.

#### Security Concerns

An issue in the latest version of the API is that even though a user change their password then old tokens can still be valid. This can be seen in the admin panel, where the user is still authenticated after they have changed their password. It is a security problem if a user's credentials are compromised because it is currently not possible to invalidate old tokens generated by the compromised credentials. Therefore, in the next release of the API, it should support that all tokens for a user are invalidated when a user changes or resets their password. Another security concern is that the superuser can gain access to all other users by generating reset password links. This means that the security of the entire system depends on the strength of the superuser's password. Another issue is that the administration panel and API is currently hosted by a server at a domain which does not enable https. This means that credentials are sent in clear text to the server when the user logs in to the system.

### 7.2.2 Changing Account Controller

In parallel with developing the admin panel we saw the need for expanding the API to support the different tasks in the admin panel. Therefore, we expanded the account controller with the following functionality:

- Deletion of users
- Registration of citizens, guardians, and departments
- Updating user and department names
- Changing password from old password

- Generating a reset token
- Resetting password from reset token

### 7.2.3 Administration Panel Usability Test

This section documents the usability test of the Giraf administration panel, as well as a description of possible future additions. It is a collaboration between our group and the PO group, sw609f18.

In the administration panel, it is possible to manage departments, guardians, and citizens, which includes features such as creating, modifying, and deleting users and departments.

#### Usability Test

The Giraf Administration web page was developed based on the functionality requested by the stakeholders: adding and removing citizens and guardians, and not based on how the user interface should be designed.

The first time the customers got an opportunity to see the Giraf Administration was at the meeting, which made it an ideal time to do a usability test of the web page.

As the customers had no prior knowledge of how to navigate the application, the results of the test would be similar to a real-life scenario of a user logging into the system for the first time and try to figure out how to navigate the system. To make the test as close as possible to such a real-life scenario, the test was constructed such that it describes a story. This story should not tell the interviewee in detail how to navigate the application, but rather what he or she wants to accomplish.

The usability test conducted during the meeting consisted of two parts, one with the web page and the other with the WeekPlanner app. The first part is listed below, showing a translation from Danish of the tasks the interviewees had to complete:

Log in as an administrator

1. You are a guardian at an institution and have been given permission to use the administrator credentials to create a guardian user for yourself in the WeekPlanner app. You open up the web page that manages guardians and citizens.
2. Use the following username and password to log in:
   (a) `Username: lee`
   (b) `Password: password`

Create a new guardian

1. Choose guardians from the menu. As you work at the Birken institution, choose this institution, and then create a new guardian.
2. Create your own username and password

Create a new citizen

1. After creating your login credentials, log out of the Giraf Administration session.
2. You now login with your newly made username and password.

3. One new child at your institution has no user in the WeekPlanner app. You therefore decide to create a user for him.
4. The citizen's name should be "Mads". Create this citizen and log out of the Giraf Administration web page.
5. Now find your tablet, to manage Mads' week plans.

### Test Result

The usability test was conducted on a single stakeholder from the *Center for Autism and ADHD* at Aalborg Municipality. The test indicates that:

- The log out button is hard to locate.
- To create a citizen after creating a guardian user, it is not clear that one has to choose citizen in the left menu.

The following points were not observed during testing, but the interviewee brought it up after the test:

- The word *pædagog* (eng. pedagogue) should be replaced by *medarbejder* (eng. employee).
- It should be possible to create more than one user with the same name.

### Test Discussion and Conclusion

As the results indicate, the user was able to use the administration panel with only minor usability issues, which should be addressed in the future (see future works below). A limitation of the conducted usability test is that it only tests a subset of the available features and was only performed on one user. Future usability tests might need to be performed, to test the entire functionality of the administrator panel, as well as ensuring that it is usable by more than one user.

### Future Works

Based on the test results, we establish the following tasks that the administration panel should support in the future:

- Setting the screen name when creating a citizen, guardian, or department.
- When changing the name of a department, the user name of the associated department user should also change.
- Rename *pædagog* to *medarbejder*.
- The log out button should be more visible/placed more intuitively.

## 7.2.4 Access Control

One of the problems with the current API, as illustrated in figure 4.1, is that in order to access the data of a citizen, the guardian has to log in as the citizen and in that way impersonate the citizen. This can be a problem for security reasons if, for instance, a citizen has private pictograms or other information that only they can see and edit.

Furthermore, it makes it more difficult to log requets to the API, as we have to save information in the token, if a user is impersonating another user (see section 6.2.5 for the data logging mechanism we implemented). It also makes it more difficult for the frontend-developers as they have to manage and save multiple tokens. In this sprint, we change this.

First, we establish a general hierarchy of which users should be able to edit and get information about other users. Currently, in the API there are four user roles; superuser, department, guardian, and citizen. The hierarchy for the roles is illustrated in figure 7.13, where the arrows indicate the hierarchy, meaning that a superuser is higher in the hierarchy than department and so fourth.

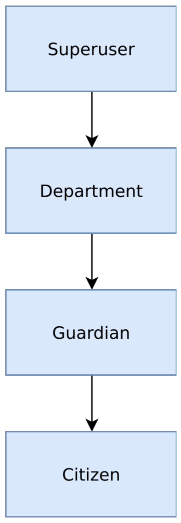


Figure 7.13: Role hierarchy

We changed the week controller and user controller endpoints to receive the user `resource` and user `identifier`, and implemented an authentication service that is injected into the constructors of the controllers. In the authentication service, we created a simple method for checking if a given user has read and edit rights to the information on another user. A pseudocode implementation of this method can be seen in algoritm 2. Note that the check `userRole < authRole` will return `true` if `userRole` is further down in the role hierarchy than `authRole`.

**Algorithm 2** Method for checking if a user can access information about another user

```
procedure CheckUserAccess(authUser, userToEdit)
    authRole ← GetUserRole(authUser)
    userRole ← GetUserRole(userToEdit)
    if authUser is userToEdit then return true
    if authRole is SuperAdmin then return true
    if authRole is Department OR authRole is Guardian then
        return IsSameDepartment(authUser, userToEdit) AND userRole < authRole
    return false
```

Due to this new way of accessing the information, we removed the possibility of logging in as another user and modified the logger (described in section 6.2.5) to not include information about impersonation.

### 7.2.5 Endpoint for Getting Version Information

Due to the new process model described in 3.2, the frequent releases meant that it was hard to keep track of which versions of the API were running on which server, especially since the continuous integration implemented by one of the server groups were not working properly. To help get an easier overview of which version of the API was running on which server and from which commit, we created a simple version-info endpoint that uses the information in the .git directory to extract the branch and commit hash that the current API runs on.

### 7.2.6 Create, Read, Update and Delete Endpoints for Week Templates

In sprint 2 and 3 we created the database structure for week templates and some basic endpoints for reading week templates. In this sprint we extend the API to also support updating, creating and deleting week templates.

### 7.2.7 Faster Loading of Basic User Information

In many of the endpoints throughout the API, it is necessary to retrieve information about users. The current methods for retrieving information about users involve eager loading all data associated with a user, which can potentially be a lot of data that have to be fetched from the database. To fix this issue, we took the methods that were responsible for eager loading all this data, called `LoadUserAsync` and `LoadUserById` and split them up into methods responsible for loading only specific entities from the database, such as basic user-information, and all associated pictograms.

To test whether the optimisation is significant, we seed the database with the sample-data from the `DBInitializer` class and add an additional 50 references from one of the sample users, Kurt, to `Pictograms`, such that Kurt now is associated with 50 `Pictograms`. We then start the API and run a simple script that makes a login request and tries to get the week-schedule of Kurt. Because the first request takes longer time to run, since data is written to the cache, the script is first runs a single time without recording the results and then an additional 50 times where the results are logged. This is done before and after the optimisation has been performed, where the data in the database is the same. The script can be seen in appendix D.

We then check out on commits [30] and [31] from before and after the optimisation, and run the script. Figure 7.14 gives a descriptive analysis of the results.

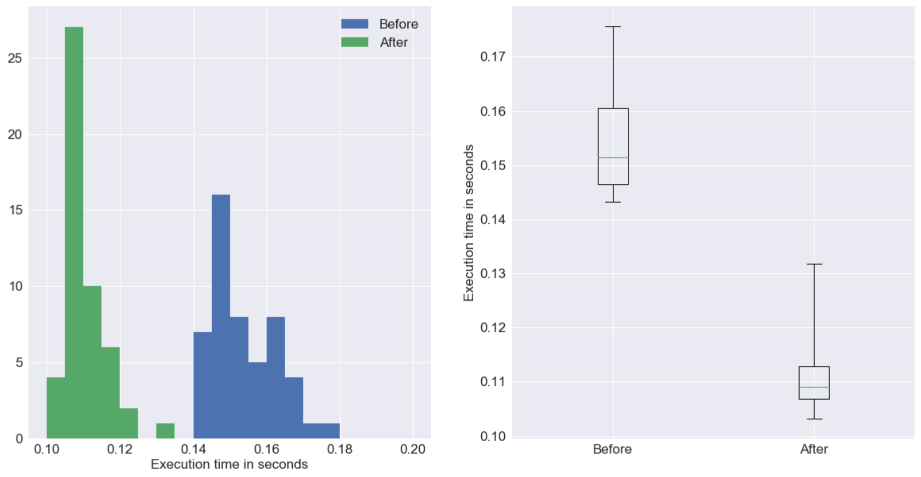


Figure 7.14: Histogram and box-plot before and after optimisation

The descriptive analysis conducted above indicates that the running time has been improved after the optimisation. When optimising code it is also essential that we do not break any functionality. Running the unit tests and integration tests we see no differences in the functionality in the results before and after the optimisation.

### 7.2.8 Migrations

In chapter 6, it was discussed how we migrated the production database and that we decided to delete all previous migrations. Since then, we have had two stable releases meaning that any future updates to the database should be handled automatically by EF migrations, without us manually updating the database. Most of the time, EF is able to automatically generate valid migration scripts, but sometimes one must manually update the scripts to ensure that they correctly migrate between the new and old database structure. Listing 7.1 gives an example of an EF migration which contains a handwritten `Up` and `Down` method.

The `Up` method describes the structural changes that should be applied to the database when running the migration script and the `Down` method describes the structural changes when the migration must be undone. Listing 7.1 shows a migration for changing the foreign key, `GirafUserId`, between the `Week` and `User` entity to be required, instead of optional. To do this, the `Up` file must describe what should happen to all data where the `GirafUserId` is null. In the example given below, we first run an SQL query deleting all rows from the `Weeks` table where the `GirafUserId` is null and then altering the `GirafUserId` column to not be nullable. In the `Down` script we reverse the daatabase to the old state where the `GirafUserId` is altered from non-nullable to nullable.

```
[...]
protected override void Up(MigrationBuilder migrationBuilder)
{
    migrationBuilder.Sql("DELETE FROM Weeks WHERE GirafUserId is
        ↪ NULL");
    migrationBuilder.AlterColumn<string>(
        name: "GirafUserId",
        table: "Weeks",
        nullable: false,
        maxLength: 127,
        type: "varchar(127)"
    );
}
protected override void Down(MigrationBuilder migrationBuilder)
{
    migrationBuilder.AlterColumn<string>(
        name: "GirafUserId",
        table: "Weeks",
        nullable: true,
        maxLength: 127,
        type: "varchar(127)"
    );
}
[...]
```

Listing 7.1: Up and down method for making the foreign key between `User` and `Week` required

## 7.3 State of the API

In this section, we describe the final structure of the REST-like API developed in this project. The purpose is to gain an overview over the API, how requests are evaluated into JSON responses, and finally to get an overview of the structure of the database.

### 7.3.1 API Structure

Figure 7.15 gives an general overview of the API structure. In the illustration, an arrow indicates that a component pointed from is dependent on the component pointed to. The dashed line from EF to the database indicates that EF is responsible for executing queries on the database.

As illustrated in the figure, there are eight `Controllers` in the API which are responsible for handling all requests to the API. The `Controllers` uses the `DTOs` to define the JSON structure for the objects in the HTTP message body. Furthermore, the `Models` have a dependency on the `DTOs` as they take the `DTOs` as parameters in the constructors. To extract information from the database the `Controllers` uses EF, which executes queries to the database. To extract and store data through EF, the `Controllers` use the `Model` classes. The `Controllers` also use services to make authentication checks and other common functionality like retrieving data from the database. As described in chapter 5, we use EF to create the database, which is defined by the structure of the `Model` classes and additional configurations through the EF fluent API.

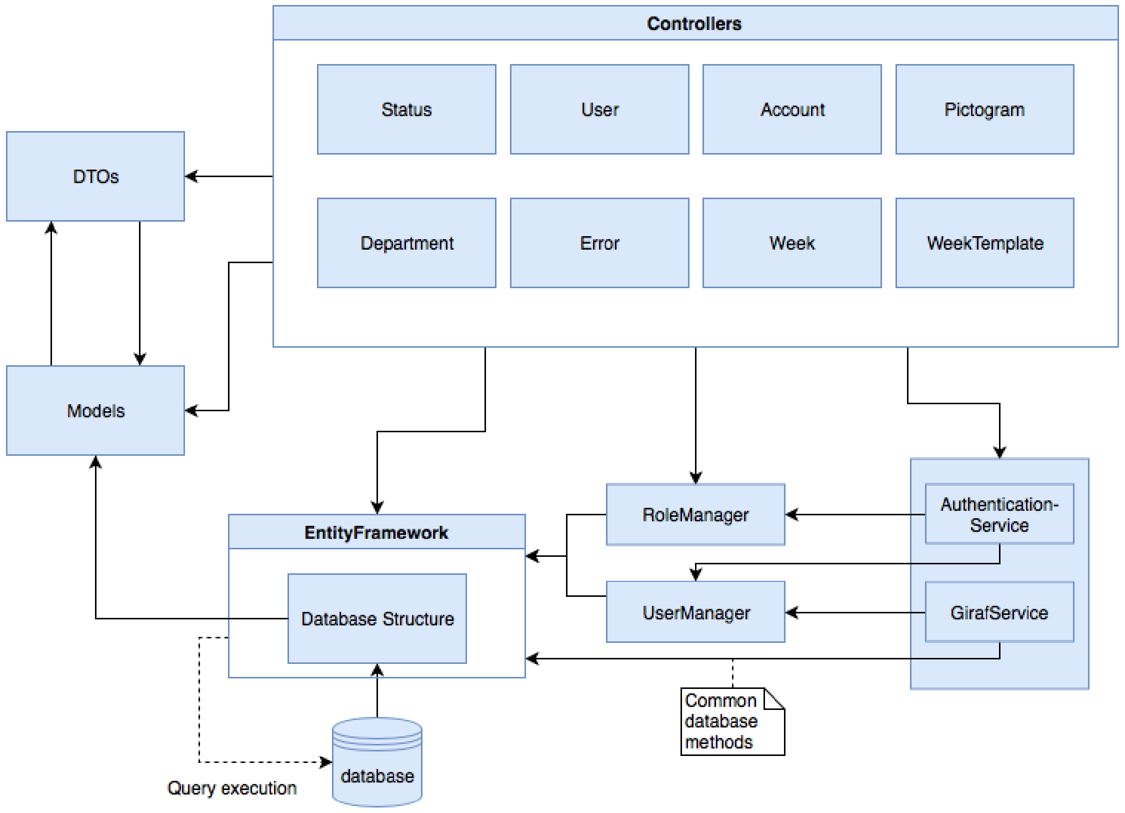


Figure 7.15: General structure of the API

### 7.3.2 API Requests

Figure 7.16 gives a simplified overview of what happens from the moment a `User` makes an interaction in the WeekPlanner app, until the `User` receives feedback from the application.

First, the WeekPlanner will use the generated client-API to make an HTTP request to the server. On the server, the HTTP request is parsed with the purpose of checking whether or not there is a matching URI and HTTP method; if no match is found, the request is redirected to the `ErrorController`, which sends an error response as JSON back to the WeekPlanner app.

If there is a match, the relevant method is called, which uses the information in the HTTP body and header to produce a JSON response accordingly.

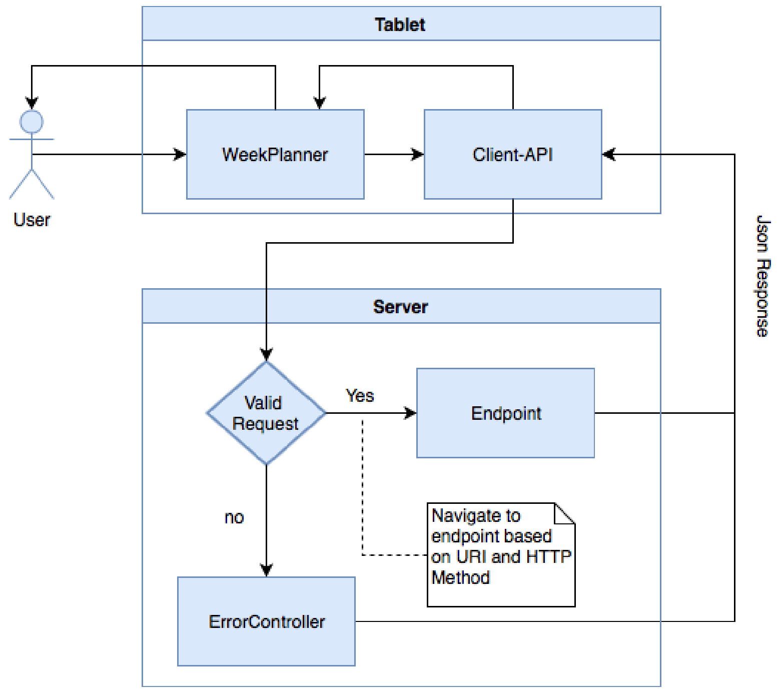


Figure 7.16: From Request to Response

### 7.3.3 Database Structure

We end this chapter by giving an overview of the database structure at the end of the fourth and final sprint in this year's Giraf multi-project.

The structure of the database is illustrated in a UML diagram. The database structure can be seen in figure 7.17, where arrows indicate that the object pointed from has a foreign key to the object pointed to. The numbers next to the arrows indicate the cardinality on both sides of the relation, and PK, FK, or CK indicates if an attribute is also a primary key, foreign key, and composite key, respectively.

As the figure illustrates, we use the ASP.NET Identity membership system [32] to manage users. Each user has a relation to a `Settings` entity, which defines configurations specific to a user, which can be set in the WeekPlanner app. Furthermore, a user has a reference to private `Pictograms` through `UserResources`, which is the pivot table for describing a many-to-many relationship between `AspNetUsers` and `Pictograms`.

A user is also part of a `Department` and has a list of week schedules, which is modelled with the `Weeks` table. A `Week` has a reference to up to seven `Weekdays`, where each day contains an `Activity`. An `Activity` is a `Pictogram` that is related to a specific `Weekday` in a `Week`, and has an order which indicates the index of the `Activity` in the `Weekday`. The `Activity` entity also contains a state attribute for storing the state that the `Activity` is in.

A `WeekTemplate` is a generic form of `Week`, but unlike `Week`, the entity does not have a year and week-number and belongs to a `Department` instead of a `User`.

`Departments` can also own `Pictograms`, which is modelled through the `DepartmentResources` entity.

In this project, we have modified and improved most of the database design. However, we have not created or changed some of the ASP.NET generated tables as well as `DepartmentResources` and `UserResources`. The `DepartmentResources` and `UserResources` entities are currently not being used by the API, but may be relevant for future Giraf groups.

As illustrated in figure 7.17, a `Setting` can belong to multiple `Users`. This comes from last year's structure for `LauncherOptions` (see figure B.1 in appendix B) which we renamed to `Settings`. Even though there is a one-to-many relation between `Setting` and `Users` in the database, the current API makes sure that a `Setting` can only belong to one `User`.

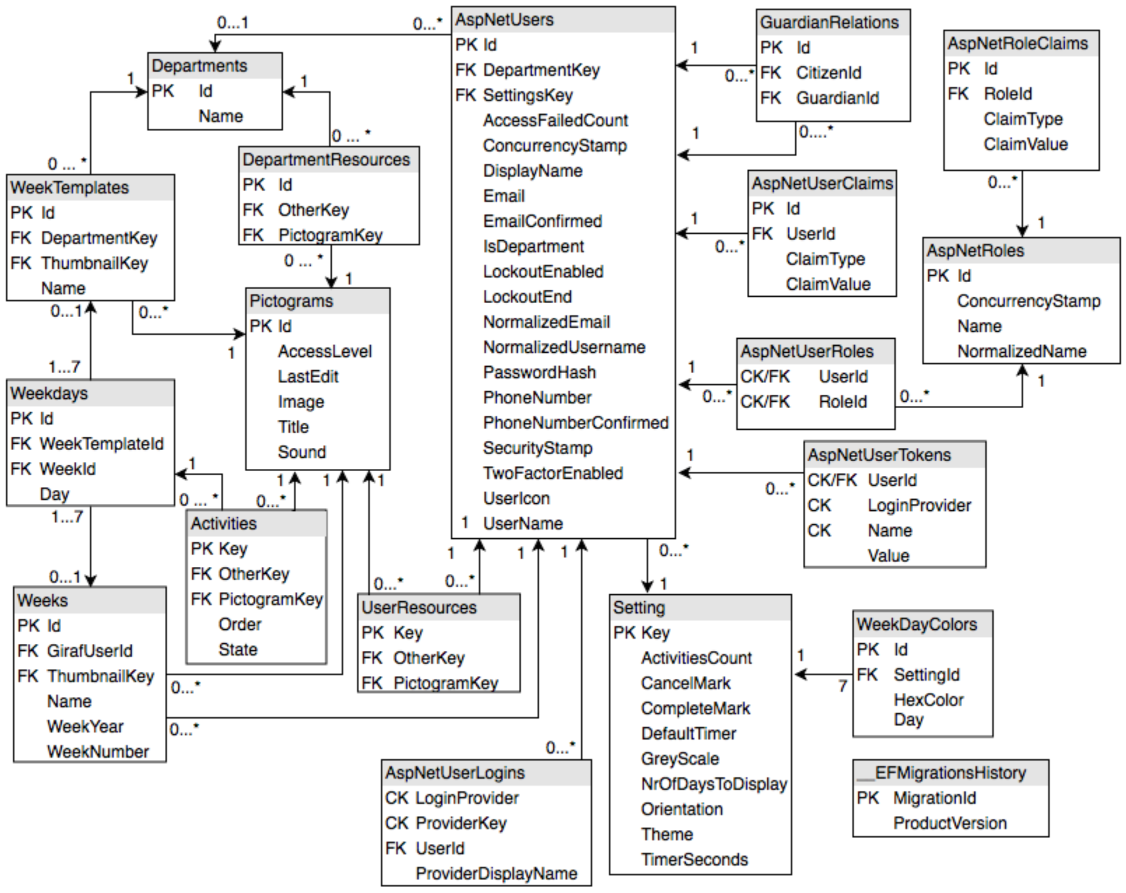
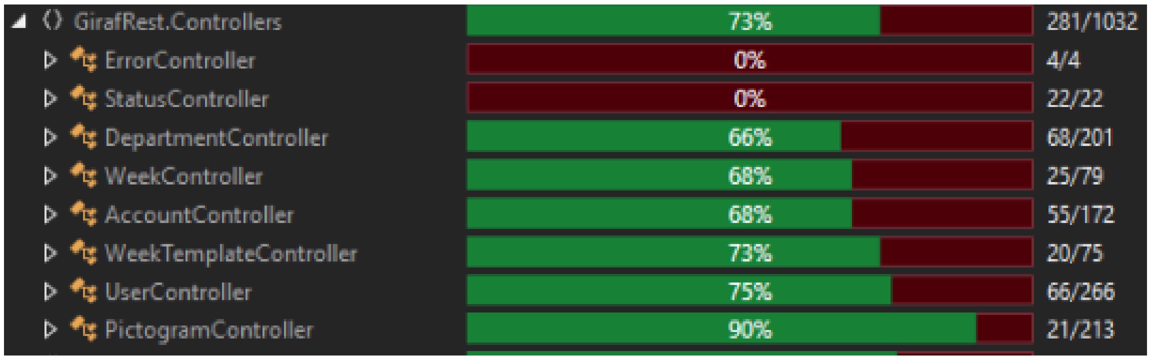


Figure 7.17: UML-diagram of the database structure at the end of sprint 4

## 7.4 Test

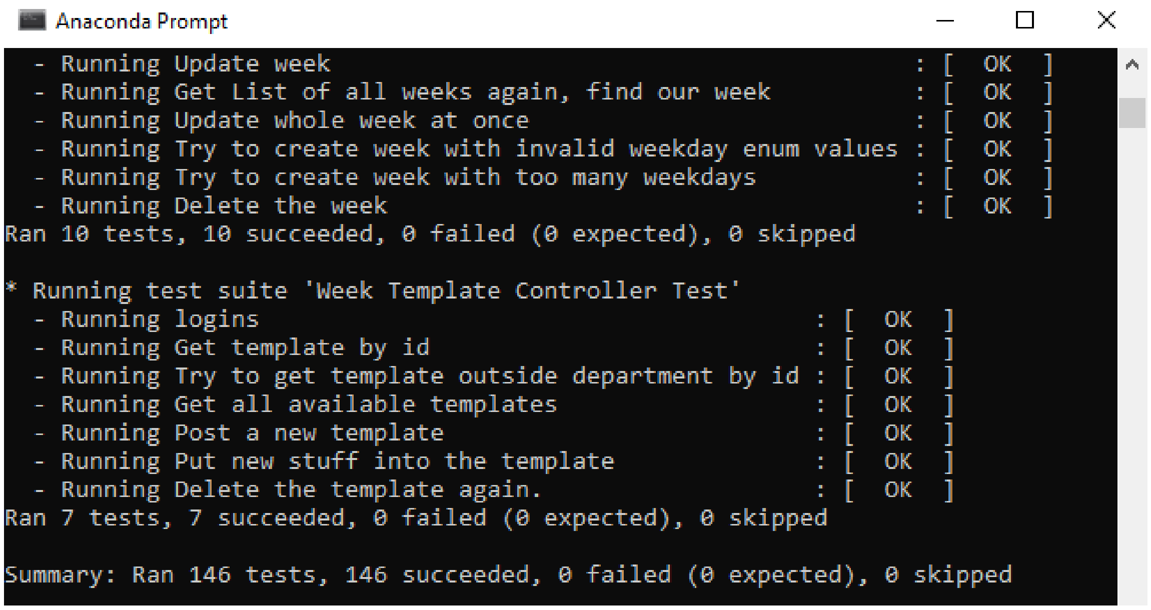
Although improving test coverage was not a focus point of this sprint, it naturally improves as new functionality is added and tested to ensure it works correctly. In addition to that, outdated functionality is deprecated, resulting in less code that is not updated in regards to unit tests. Running the same `dotCover` test as in section 5.3.2, we get the results shown in figure 5.3.3. The code coverage increased approximately 4 percentage points from 69% in sprint 2 to 73% in sprint 4. The exact commit on which the test coverage is performed can be found at [33].



| | | |
|---|---|---|
| GirafRest.Controllers | 73% | 281/1032 |
| ErrorController | 0% | 4/4 |
| StatusController | 0% | 22/22 |
| DepartmentController | 66% | 68/201 |
| WeekController | 68% | 25/79 |
| AccountController | 68% | 55/172 |
| WeekTemplateController | 73% | 20/75 |
| UserController | 75% | 66/266 |
| PictogramController | 90% | 21/213 |

Figure 7.18: A list of the percentage covered by unit tests

In addition to the 73% Controller code coverage, we also added more integration tests in connection with adding new functionality. At the end of sprint 4, the number of integration tests reached 146, which all pass as shown in figure 7.19, and the number of unit tests reached 342 against 227 when we started. The integration tests were not included in the coverage results, as we were unable to get `dotCover` to record code-coverage while making requests to the API.



```
Anaconda Prompt
  - Running Update week                                         : [  OK  ]
  - Running Get List of all weeks again, find our week          : [  OK  ]
  - Running Update whole week at once                           : [  OK  ]
  - Running Try to create week with invalid weekday enum values : [  OK  ]
  - Running Try to create week with too many weekdays           : [  OK  ]
  - Running Delete the week                                     : [  OK  ]
Ran 10 tests, 10 succeeded, 0 failed (0 expected), 0 skipped

* Running test suite 'Week Template Controller Test'
  - Running logins                                       : [  OK  ]
  - Running Get template by id                           : [  OK  ]
  - Running Try to get template outside department by id : [  OK  ]
  - Running Get all available templates                  : [  OK  ]
  - Running Post a new template                          : [  OK  ]
  - Running Put new stuff into the template              : [  OK  ]
  - Running Delete the template again.                   : [  OK  ]
Ran 7 tests, 7 succeeded, 0 failed (0 expected), 0 skipped

Summary: Ran 146 tests, 146 succeeded, 0 failed (0 expected), 0 skipped
```

Figure 7.19: Screenshot of console output from running the 146 passing integration tests after sprint 4

## 7.5 Collaborating on Writing the Documentation

Good documentation is essential in the Giraf multi-project, as developers are substituted once a year, after working on the project for half a year (the summer semester) while also attending courses. The communication relevant for the turnover of the project primarily happens through text in project reports or on the Giraf Wiki [34].

It had to be ensured that duplication of work between groups was minimised, and that all documentation written were collected in one place, to make it coherent and easily accessible. To do this, a member of our group was elected to gather representatives from other groups.

These representatives were expected to make an extra effort of keeping their groups updated on what needed to be done in terms of documenting the work to next years students, while also meeting before the last sprint finished, to brainstorm which topics should be written about. The results of this brainstorm were eventually collected on a Trello board [35] where tasks could be claimed by individual groups. After all groups had finished working on the code, another meeting was held among the representatives to talk about how the tasks were coming along, and which of them should be prioritised.

A wiki for the Giraf project already existed from the previous years, and this was the obvious choice of medium for the documentation.

Some information about the server infrastructure and the backend code base already existed, but changes and new features made during this semester still had to be recorded.

As the WeekPlanner app had been completely rebuilt from scratch, so should the documentation for it. Most notably, instructions for building the project were lacking.

Table 7.1 contains all articles on the wiki about the backend and what work we have done on each of them. These articles are hosted on [34].

| **Article Title** | **Description of Our Work** |
|---|---|
| Get Started | Almost everything was re-written |
| Database | Minor changes |
| Entity Framework | Minor changes |
| Models and DTOs | Updates to reflect changes in code base |
| Authorization Roles and Policies | Almost everything was re-written to reflect changes |
| Giraf Service | Updates to reflect changes in code base |
| Backend Architecture | Original work by this group |
| Endpoints and Controllers | Updates to reflect changes in code base |
| Swagger | Original work by this group |
| Unit Testing | Some parts re-written for clarification |
| Integration Testing | Original work by this group |
| Future work | Original work by this group |
| Formatting Guide | Original work by this group |
| Relevant Repositories | Major changes to reflect the new state of the Giraf project |

Table 7.1: Articles about the backend code-base and a description of what part of them is done by us

## 7.6 Conclusion

To ease the accessibility of administrative tasks in the Giraf app environment, an administrator web page was created. This system was created in the last sprint of the project, and therefore had to focus on implementing the necessary functionality that was not yet supported in the WeekPlanner app. It was presented to stakeholders of the Giraf project at the acceptance test of sprint 4. As concluded in 7.2.3, the stakeholders were able to navigate the system with only minor usability issues. These issues were noted and can be seen as future work to future Giraf projects.

Sprint 4 also included a redesign of the access control. This was needed, as a guardian accessing one of their citizens previously required the guardian to impersonate the citizen, which also meant that they had access to all information for that citizen. The redesign now does not require the guardian to impersonate, but rather just send the relevant information for the given citizen.

The unit tests of the backend, as shown in 7.18, ended up covering 73%, with an additional 146 integration tests.

# Chapter 8

# Discussion

The purpose of this chapter is to describe problems encountered and considerations during the four sprints of the 2018 Giraf multi-project.

## 8.1 Development Method

Initially, the development method decided upon for the multi-project was a Scrum of Scrums [36], where each individual group was given the choice of deciding whether or not they wanted to abide by the Scrum method internally, but Scrum was used externally, between the groups.

### External Development Method

The previous Giraf multi-project communicated through their report to later projects that they recommended using Scrum as the external development method. This is advantageous according to Boehm and Turner's model of home grounds, which is used to get an indication of whether an agile or plan-driven method is better suited for the project [37].

A project like Giraf is characterised by *chaos*, since there is a great deal of uncertainty, based on lack of experience of working in multi-projects. In addition, the project is characterised by *limited time* for the teams to understand the project at the time of take-over, develop the project, and document their work. Furthermore, the project is *dynamic* in the sense that we have different stakeholders with different requirements and ideas, for the system. The ideas often change when the stakeholders are introduced to new versions of the system. Using an agile development method helps us support changing requirements during the development of the project.

The development of the WeekPlanner is also defined by some degree of *criticality*, as we have a system containing sensitive personal data. Another factor contributing to the criticality of the project is the diagnose of the users, especially the children diagnosed with ASD: "If there is a sudden change in routine, or if a routine is disrupted, this can have a very negative effect on the child's behaviour." [38]. Therefore, if the system is unreliable, this may have a worse impact on the user, compared to another schedule application not aimed at people diagnosed with ASD or similar disorders.

### Change of Method

As the first two sprints went by with no releases, our group took the initiative to adopt the IWWP process described in section 3.2. We did so because we believed that continuing in the same manner, would never lead to a stable release, as new features were added before old features were stable, and working as they were meant to.

The focus of the new process model was to make smaller releases and ensuring better configuration management between the different Giraf components. By adopting this process model, we managed to get three stable releases onto the Google Play store.

It is difficult to say whether or not the new process model ended up giving us a better product than the old process model would have yielded. We believe, however, that adopting the process helped create value for the customers, as they could see a working system evolving, while also improving the motivation internally by shifting focus to concrete and frequent results.

### Configuration Plan

From our experience working in a multi-project setting, we believe that it is essential to have a proper configuration management plan, to avoid problems with incompatible software where sudden changes in one software component break the build of another component. Therefore, we believe that it is important early on to establish good conventions for managing configurations, as well as explicitly stating the feature-set that must be included in the upcoming releases.

In this project, we established a configuration plan early in sprint 1 (see appendix A), but it was not before we changed to the new development model, IWWP, that we saw the configuration management be used. Before changing to IWWP, the frontend group was not working on the designated release branches, but instead pulling directly from our develop branch, which frequently led to broken builds as we merged new features, changing the function and/or names of endpoints, to the develop branch.

## Internal Development Method

An internal development method was not a requirement for the six Giraf teams, but at the start of the first sprint, our group assigned a team member the role of Scrum Master. After looking at relevant Scrum practices, we decided upon a few practices which we found relevant, and opted to neglect the rest. We did so because of the time limit of the project, and the time required to enforce that the practices are being followed, and also because a majority of the team are used to working together.

We introduced practices such as keeping an updated backlog to get an overview of tasks within each sprint. One of the Scrum practices that we opted out of, was the daily Scrum, as our experience told us that the communication between the members in the group room worked well informally. As inter-team communication was critical when working horizontally (teams assigned to specific components, but developing in features, see IWWP), it was not a problem that we did not have an active Scrum Master to e.g. protect the team from outside interference. Furthermore, having separate groups that acted as PO and Scrum masters also removed the need for having a separate Scrum master within our group. Therefore, we decided to not have an active internal Scrum master. Looking back, we could probably have benefited from having a more active Scrum master at times, as the backlog was not always kept up-to-date, and tasks were not properly estimated.

## 8.2 Progress

As mentioned in section 2.2, the overall goal of this project was to make the pre-existing WeekPlanner app stable enough to be useful in Kindergarten Birken. As mentioned in 5.1, it was the frontend teams belief that it would be less time-consuming to re-implement the WeekPlanner app in C# and Xamarin, than to fix the current app. This was obviously a rather large setback, especially as the first sprint was also used on fixing the state of the original front end.

It is uncertain whether or not we would have been able to implement more features, and how the WeekPlanner would have looked, if we had opted to continue the development from last year. However, we have managed to get a working app with three stable releases, which a stakeholder present at the last acceptance test stated would be usable, if the database were to be seeded with better pictograms.

Furthermore, we have left the code that we worked on in working state on stable branches, which we believe is important as one of the reasons we did not choose to continue development of the old WeekPlanner app was because it was left in an unstable state due to last minute changes.

### Internal Backend-specific Goals

As the state of the backend required fewer changes than other components of the WeekPlanner app, we created technical stories to improve the backend in regards to software quality factors, like stability.

As described in 4.1, the backend already contained a lot of unit tests, although these contained compile errors when we took over the project. Therefore, a goal of the backend was also to get these working, and, in addition to this, ensure that a majority of the code is properly tested. At the time of sprint 4, 73% of the controller classes' source code is covered by the unit tests. This is in addition to all the integration tests added to the project, which were not included in the results of the test coverage.

## 8.3 General Considerations

The purpose of this section is to discuss the following points: Swagger, horizontal development, and working in a multi-project setting.

### Using Swagger

As described in section 4.3.6, Swagger-codegen was used to generate the client-side API. Using Swagger made it easy for the front-end groups to keep up with the newest changes from the backend. However, one of the problems we encountered was with SwashBuckle [39], which we used to generate the Swagger OpenAPI specification, which did not support polymorphism by default. Even though it was possible to manually configure Swashbuckle to support polymorphism, Swagger-codegen still had problems generating valid enums. We discovered that this was an issue with Swagger-codegen for C# and tried to fix it by using NSwag [40] to generate the client. Even though we were successful in using NSwag, the code generated this way was put into one file and was not ideal for larger applications. We therefore decided to continue using Swagger-codegen without polymorphism.

Using Swagger has it advantages and we recommend that future groups also use Swagger to ensure that changes in the backend can be incorporated quickly in the frontend. Currently Swagger is in active development and we believe that some of the problems encountered with Swagger-codegen will eventually be fixed by the Swagger development team.

### Working Horizontally

As explained in chapter 3, the project groups this year decided to develop horizontally instead of vertically. In retrospect, the advantage of having explicit areas of responsibility is that a single group has a limited part of the code base that they are responsible for and can thereby enforce that it adheres to certain standards. In addition to this, getting a Giraf group familiar with only a single code base is quicker, than if they had to learn the entire code-base, which decreases the time between the start of the project and the first feature.

By developing horizontally, there is a natural communication between groups, as one group will inevitably be dependent on the work of other groups. In our experience it ensures that no group is able to work completely independently of the other groups, which suits the learning goals of this semester.

A disadvantage is that it can be hard for some groups, especially server and backend groups, to see what value their work brings to the entire project. In our experience, this can be helped by having frequent releases to App stores, so that the different groups can see how their work integrates with the system as a whole.

### Collaborating with Other Groups

In regards to working in a multi-project setting, it is different from what we are used to, in terms of the communication required when implementing a new feature, since the decisions the group makes can affect the entire project. To facilitate good communication between groups and to control the different components, we learned the importance of following a proper process model and configuration plan, which has not been essential in other projects where the single group could quickly resolve issues informally.

# Chapter 9

# Conclusion

As stated in chapter 2, the goal for this project was to make the WeekPlanner app stable and usable. To facilitate the development of the app across the six Giraf groups, each grup was given different responsibility areas, ours being the backend-side of Giraf, which is the API and the database. Initially, Scrum was used for project-management and the project was divided into four sprints.

In sprint 1, we improved the stability of the backend by fixing the unit tests from last year and integrated Swagger as middle-ware, to document the endpoints of the API. Furthermore, we updated .NET core to the latest version and changed the response types of the API to a custom format. To ensure proper version control and management of components, we also created the Giraf Configuration Management Plan (see appendix A)

In sprint 2, we replaced cookie authentication with token authentication as a REST API should be stateless. Furthermore, in order to test the dependencies used in the API, such as authorisation and database access, we implemented several integration tests to ensure high stability of the API. By having a high test coverage for the unit tests of 73% in addition to the integration tests, we ensured that only few errors had to be discovered by the frontend groups during development.

In sprint 3, several problems with the process model in use were discovered, which resulted in it being substituted by our own process model, the Iterative Waterfall with Pipelining (IWWP) model. It emphasises smaller sprints and frequent releases. With the new process model, two stable releases of the WeekPlanner app were published on the Google Play Store. The backend was improved by fixing several problems with the current database structure of week schedules, as well as implementing user-specific settings and a custom log filter for logging all requests to the API.

In sprint 4, we developed the admin panel, through which users and departments can be managed. The admin panel is designed to be used by superusers, department-users, and guardians and supports different features depending on the access level of the user. Features include searching for departments and users, creating, modifying, and deleting users and departments, as well as generating password reset links for specific users, depending on the administrators access level. Furthermore, we added several authentication checks to the API and ensured that only users that were supposed to, could access other users' information.

The acceptance test conducted at the end of sprint 4 indicates that the stakeholder present at the acceptance test appreciated the new WeekPlanner in terms of functionality and stability. Furthermore, the usability test conducted on the administration panel (see section 7.2.3) indicates that the administration panel was usable with only minor usability issues.

The stakeholder also stated that it would be possible to use the application if the database was to be seeded with specific pictograms and if the icon for showing if an activity is a choice board was to be changed. However, as only one stakeholder was present at the last acceptance test, it is not possible to conclude if we managed our project goal of making the WeekPlanner application stable and useful as this requires more insight and testing.

Besides being responsible for the backend of Giraf, we also managed to adopt the process model used to ensure more stable releases, create a configuration management plan for managing the components of Giraf, as well as developing the core functionality of an admin panel for managing users and departments in the WeekPlanner application.

Through our work in the multi-project setting, we learned that it is especially important to have a proper process model, where frequent releases are deployed and shown to the stakeholders. This ensures that features are fully implemented before development of new ones starts, which gives value to the users in the form of stable products that they can use.

Likewise, we have learned that it is essential to have a proper configuration plan for managing the different components in terms of version-compatibility. It is also important that a fixed feature-set is chosen to be included for each release, as this ensures that each group know what the common goal for the release is, which makes it easier to work together when multiple teams are developing horizontally.

# Chapter 10

# Advice for Next Years Students

The purpose of this chapter is to share our experience and advice with next year's students.

## Drop Practices That Do Not Create Value

During the sprint reviews, all project groups perform time estimation of the different tasks using planning poker. However, the results of the time estimation process were never used. We recommend that you identify all practices used and ask yourself what value they bring to the project.

## Use a Development Method

Besides using an agile framework like Scrum, we recommend that you use a development method. This year, we used the IWWP development method (see section 3.2) which we recommend next year's groups to use. IWWP ensures frequent releases and reduces idle time for the different component layers.

## Horizontal Development

We recommend working horizontally, unless all developers have experience in Xamarin, Entity Framework, Vue etc. Working horizontally results in less overhead when acquiring new knowledge and as each group is responsible for a specific part of the code-base, they ensure that it adheres to a certain standard. That said, it does not mean one group cannot work on other components. This should be agreed upon beforehand and should be done through pull-requests. This year we identified three component layers: Server, API, and App. We chose to have two groups on the server component, one on backend, and three on developing the app where two of the app groups also had other responsibilities, one being Product Owners and one being the role of external Scrum Master. We recommend not having two groups on server, but instead putting four groups on frontend-related tasks like administrator panel and app.

## Roles in Giraf

We had two groups with extra responsibilities. One was the Scrum group, responsible for facilitating the Scrum practices. The second group was the Product Owner group, responsible for communication with the clients, getting their ideas realised by designing prototypes, and deciding which features needed to be implemented along with a prioritisation of these. But seeing as people tend to form groups based on relationships, and not based on what they want to work on, one will often end up with groups that do not agree on which component level and special responsibilities to work on. Furthermore, if everything has to be agreed on by committee, one tends to get less work done. Therefore, we recommend having extra roles assigned to two person teams, examples of could be:

**Design lead** : Design prototypes and decide on the UI design.

**Scrum Facilitator** : Responsible for preparing for the different Scrum rituals that you chose to follow.

**Product Owner** : Responsible for conducting meetings with the clients and representing their will towards the project groups. Product owner would also be responsible for defining which changes should be included in which release.

**Secretary** : Who would book rooms, keep a calendar of deadlines for the project, and take notes at important meetings.

**Development lead** : Helps the product owner decide what can be accomplished in a development cycle as well as how tasks can be broken down into smaller, more manageable tasks. The development lead should also judge if a release will be done in time, and if not, help figure out a solution which will usually either be to reduce the number of features in the release, or to find more people which can help complete the features.

The reason for being two person teams is that there is shared responsibility, as well as sharing the workload, which should also ensure that one of them is always available during working hours. We also recommend that the two are not from the same project group, to introduce other opinions and promote working together between groups.

### Configuration Management

This year, we experienced that having a configuration management plan is not the same as following one. During the first two sprints all app groups worked from our `develop` branch instead of our `release` branch, which meant that new features and changes in the backend code-base often led to issues in the front-end, since changes was pushed without warning.

Our advice is to have a meeting with everyone at which you decide on a useful set of conventions i.e. the GitFlow Workflow [8]. For inspiration, see our configuration management plan in appendix A.

### Recommendations for Project Focus

We recommend that the project focus for next year is to continue the development of the Week-Planner and getting it to a state where the clients find it a useful tool in their work. Stop work on WeekPlanner if you reach this state and let the clients use the system for a while to uncover problems and get a feel for what they really need and let next years students take over from there.

We also recommend spending some time figuring out what the Giraf project as a whole is working towards. Should we continue working towards the multi-app tool-set that has been? Is there a better way of structuring the features into a new set of apps?

Furthermore, we recommend that the server groups focus on continuous integration, automatic builds, and web-hooks. These, if implemented properly, will help all the development groups and dramatically reduce the workload on the server-groups. Continuous integration should ensure that any changes to a component will not break the build and that all tests (integration and unit) pass before automatically deploying the changes to the server and relevant app stores. We recommend first focusing on implementing automatic builds.

The backend groups should focus on security, better search and possibly better data-structure of users, especially if the stakeholders want support for parents in the system.

# Bibliography

[1] Swagger. *World's Most Popular API Framework, Swagger.* English. 2018. URL: https://swagger.io.

[2] *Målgruppe (Target group).* Danish. Fagcenter for Autisme og ADHD Nordjylland - Specialbørnehaven Birken. URL: http://bhbirken.dk/index.php/malgruppe.

[3] *Samarbejde mellem universitetet og Specialbørnehaven Birken.* Danish. Kindergarten Birken. URL: https://www.moodle.aau.dk/pluginfile.php/1119544/mod_folder/content/0/Griraf%20Sus%2CKristine.pptx?forcedownload=1.

[4] Multiple. *Giraf - Autism environment for Android devices.* English. URL: http://giraf.cs.aau.dk/.

[5] Roy T. Fielding. "Architectural styles and the design of network-based software architecture". PhD thesis. University of California, 2000.

[6] Ben Morris. *Pragmatic REST: APIs without hypermedia and HATEOAS.* 2015. URL: http://www.ben-morris.com/pragmatic-rest-apis-without-hypermedia-and-hateoas/.

[7] Richard Lawrence. *Vertical Slices and Scale.* English. 2016. URL: https://agileforall.com/vertical-slices-and-scale/.

[8] Vincent Driessen. *A successful Git branching model.* English. 2010. URL: http://nvie.com/posts/a-successful-git-branching-model/.

[9] *Postman | API Development Environment.* Mar. 2018. URL: https://www.getpostman.com/.

[10] *Migrating from ASP.NET Core 1.x to 2.0 | Microsoft Docs.* Microsoft. Mar. 2018. URL: https://docs.microsoft.com/en-us/aspnet/core/migration/1x-to-2x/.

[11] Google. *Volley.* English. URL: https://github.com/google/volley.

[12] Xamarin. *Xamarin - Everything you need to deliver great mobile apps.* English. URL: https://www.xamarin.com/.

[13] Swagger. *Swagger Codegen.* English. URL: https://swagger.io/swagger-codegen.

[14] RestSharp. *RestSharp - Simple REST and HTTP API Client for .NET.* English. URL: http://restsharp.org.

[15] Microsoft. *Introduction to Entity Framework.* English. URL: https://msdn.microsoft.com/en-us/library/aa937723(v=vs.113).aspx.

[16] Rowan Miller et al. *SQLite EF Core Database Provider Limitations.* English. 2017. URL: https://docs.microsoft.com/en-us/ef/core/providers/sqlite/limitations.

[17] JWT.io. *Introduction to JSON Web Tokens.* English. 2018. URL: https://jwt.io/introduction/.

[18] *The Levenshtein-Algorithm.* English. URL: http://www.levenshtein.net/.

[19] *Wagner-Fischer algorithm.* English. Wikipedia. May 2018. URL: https://en.wikipedia.org/wiki/Wagner%E2%80%93Fischer_algorithm.

[20] elastic. *Elastic front page.* English. URL: https://www.elastic.co.

[21] Roy Osherove. *The art of unit testing : with examples in C.* Shelter Island, NY: Manning Publications, 2014. ISBN: 9781617290893.

[22] *integrate - Testing framework for integration tests.* anfema. URL: https://github.com/anfema/integrate.

[23] *Installing Python Modules*. Python Software Foundation. URL: https://docs.python.org/3/installing/.

[24] *dotcover: A Code Coverage Tool for .NET by JetBrains*. JetBrains. URL: https://www.jetbrains.com/dotcover/.

[25] sw613f18. *The commit on which test-coverage was run after sprint 2*. English. URL: https://gitlab.giraf.cs.aau.dk/Server/web-api/commit/4c6162155f2335b5a83c183863b1ae0d8a16e2a8.

[26] Releases. *Release 2018S3R1*. English. 2018. URL: http://web.giraf.cs.aau.dk/w/releases/.

[27] *IActionFilter Interface*. English. Microsoft. URL: https://msdn.microsoft.com/en-us/library/system.web.mvc.iactionfilter(v=vs.118).aspx.

[28] *Material Design Component Framework*. English. Vuetify. URL: https://vuetifyjs.com/en/.

[29] vuetifyjs.com. *Centered layout*. English. URL: https://github.com/vuetifyjs/vuetifyjs.com/blob/master/examples/layouts/centered.vue.

[30] sw613f18. *The commit before doing optimisation*. English. URL: https://gitlab.giraf.cs.aau.dk/Server/web-api/commit/dec45b6888d30a784362ab3a32a4dcf21c24667d.

[31] sw613f18. *The commit after doing optimisation*. English. URL: https://gitlab.giraf.cs.aau.dk/Server/web-api/commit/a1bcc363885e6fa0e02895677b880b16bbd62ebf.

[32] Galloway et al. *Introduction to ASP.NET Identity*. English. URL: https://docs.microsoft.com/en-us/aspnet/identity/overview/getting-started/introduction-to-aspnet-identity.

[33] sw613f18. *The commit coverage was run on after sprint 4*. English. URL: https://gitlab.giraf.cs.aau.dk/Server/web-api/commit/8713d0473f81cc5a9419cf310336281bd1d9d05b.

[34] Several years of Giraf groups. *Wiki Frontpage*. English. 2018. URL: http://web.giraf.cs.aau.dk/w/.

[35] The Giraf F18 groups. *The Giraf Archives(Trello)*. Danish. 2018. URL: https://trello.com/b/MiqbvxaW/the-giraf-archives.

[36] Jeff Sutherland. "Agile Can Scale: Inventing and Reinventing SCRUM in Five Companies". English. In: (Vol. 14, No. 12). URL: http://www.controlchaos.com/storage/scrum-articles/Sutherland%20200111%20proof.pdf.

[37] Barry Boehm and Richard Turner. "Observations on Balancing Discipline and Agility". English. In: (). URL: https://www.moodle.aau.dk/pluginfile.php/937150/mod_resource/content/1/BoehmTurner.pdf.

[38] *Negative Response to a Disruption of Routine*. English. Special Learning, Inc. URL: https://www.special-learning.com/article/negative_response_to_a_disruption_of_routine.

[39] domaindrivendev. *Swashbuckle.AspNetCore*. English. URL: https://github.com/domaindrivendev/Swashbuckle.AspNetCore.

[40] RSuter. *NSwag: The Swagger/OpenAPI toolchain for .NET, Web API and TypeScript*. English. URL: https://github.com/RSuter/NSwag.

[41] Giraf. *GitLab Giraf front-page*. English. URL: https://gitlab.giraf.cs.aau.dk/Server/web-api.

[42] Github - Choose a License. *MIT License*. English. URL: https://choosealicense.com/licenses/mit/.

[43] BitBucket. *Gitflow Workflow*. English. URL: https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow.

[44] Giraf. *PO Requirements for release 2018S3R2 - User settings*. English. URL: http://web.giraf.cs.aau.dk/w/releases/2018s3r2/requirements/.

[45] *Giraf-Rest web-api Changelog*. Giraf multi-project 2018. URL: http://git.giraf.cs.aau.dk/Giraf-Rest/web-api/src/v1.002.01/CHANGELOG.md.

# Appendices

Appendix A

# The Giraf Configuration Management Plan

## A.1 Preamble

### Authors

sw613f18

### A.1.1 Purpose

The purpose of the Giraf Configuration Management plan is to establish the rules and guidelines for managing versions of the individual components and managing the versions of different components in a complete system release.

Configuration management is important in order to keep track of what versions of which code-files exist where and how they relate to a component, as well as keeping track of baselines, their component versions, change-logs and requirements to future baselines.

## A.2 Version Control

The Giraf project consists of different components e.g. the different Android apps, the java libraries, the backend API etc. Each component is managed through the decentralised version control system: Git. The components source code is hosted on [41] through the *GitLab* hosting platform. It is highly recommended that each repository contains a README.MD file with the following contents:

- Description of the component - A high level description of the purpose of the component
- Dependencies and installation guide - A guide to users, how do they get the component running
- Test execution guide - How are tests run on the component.
- Development guide - A guide on setting up a development environment so that developers can run, develop and debug the component.
- Component specific sections - e.g. API reference for the API component
- Contributors

- License - this year we used MIT [42]

### A.2.1 Git Workflow

Every component should follow the GitFlow Workflow described in [43] and in [8]. All project members should have read either of the two documents.

The short summary is: We work on a set of branches (see table A.1), each with a different responsibility i.e. `master` should only contain working releases, `develop` should be used for working, and as such can contain incomplete versions of the systems and forms a base for any further development as well, etc.

The only changes to standard Git workflow, is that `feature` branches are instead created from, and merged into, a `release` branch instead of from and to `develop`. This is because we want different components to work together when working on a specific release. Say release `2018S3R1` and `2018S3R2` are being developed simultaneously. Then we do not want the apps to use the version of the API which is on `develop` where there may be changes that are only meant for `2018S3R2`. So when a new release starts, create a `release-*` branch from `develop` and then create `feature-*` branches from `release-*`. Any `release-*` branch can always be merged into newer releases.

| **Branch** | **Description** | **Created from...** | **Merged into...** |
|---|---|---|---|
| `master` | Stable versions suitable for use by the end user | N/A | N/A |
| `develop` | Contains latest working versions and is the base of further development | `master` | N/A |
| `feature-*` | Any features under development | `release-*` | `release-*` |
| `hotfix-*` | Quick fixes to fatal bugs on master that need to be fixed quickly | `master` | `develop and master` |
| `release-*` | Configuration items for each release | `develop` | `develop and master` |

Table A.1: The different types of branches in the GitFlow Worklow. At the end of a branch's life cycle, it is merged into the mainline branches described in this table

The `master` branch contains the newest stable version suitable for users of the Giraf applications. Before code is merged into `master` it must be approved by the product owner. Code from `master` is merged from `release` and some `feature-*` branches.

The `release` branches contain the code for the release of the different versions of the component and code is merged from `develop`. These branches should be named according to the following format:
`"release-<year>S<sprint-number>R<sprint-release-number> "`.

The `develop` branch is the baseline for any new development. Any branch can always be merged into `develop`.

A `feature` branch should be created for each new functionality that is to be added to a component. Once a feature is implemented and tested, it is merged into the release branch it was created from and the corresponding feature branch should be deleted. It is recommended to merge via pull-requests and have someone else approve your changes.

A `hotfix` branch is created from the `master` branch when a critical bug is discovered which needs to be fixed before a new release is published. Once the bugs are fixed the branch is merged and deleted.

A `bugfix` branch is created for fixing a bug. Usually has an Issue-number if on Github or some other way of identifying the bug. When merged, the branch should be deleted, as with `feature` and `hotfix`.

**year:** The current year, e.g. 2018.

**sprint-number:** The sprint number you are in. Historically, a Giraf semester has been split into four sprints. So a sprint number could be 3.

**sprint-release-number:** The number of the release. Resets each sprint. So you might have release 2018S2R4 followed by release 2018S3R1.

## A.3 Version Management

Each software component should have their own baseline, and changes to the individual components should be controlled using Git as described previously.

### Managing Component Dependencies

To keep track of releases, we use the Phabricator wiki. On the wiki a separate article will be created for each complete system release.

A system release consists of a release schedule with a series of deadlines, a list of requirements that defines the development task for the release and links to source code for each component related to the release.

- *Requirements* First deadline for each release is when the set of requirements that defines the release are finished. This document should contain enough information about each feature in the release, such that it can be developed throughout the stack.
- *First version* Next deadline is when the first version of the release can be put together and run. This version might be buggy, but the features described in the release-requirements should all be there.
- *Final* The next deadline is the final release-deadline and at this point, the release-branch should be ready to be deployed and now the PO-group can test if all requirements are met, to approve the release for publishing.

An example of the requirement document that we, together with the PO group, created for `Release2018S3R2` can be found by visiting [44] or in the electronic appendix.

# Appendix B

# Endpoints and Database Structure at the Time of Takeover

Figure B.1 shows the final database structure from the MySQL development database (giraf-dev) from 2017.

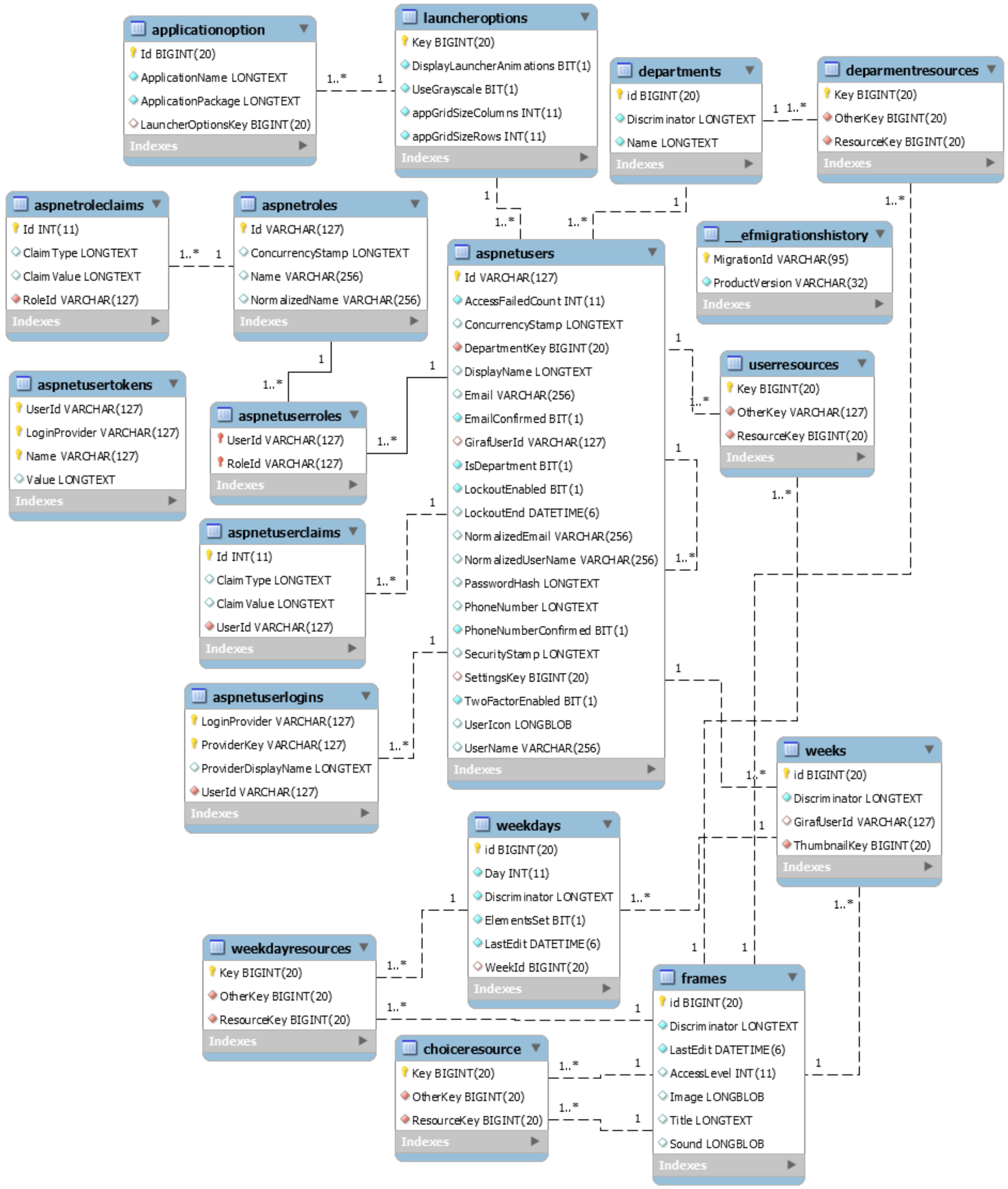


Figure B.1: UML of the database structure left behind by Giraf 2017. Generated using MySQL Workbench on the development database left by Giraf 2017.

# Endpoints

Updated 286 Days Ago All Users

Below you will find a complete list of all currently (as of 29th of May 2017) available endpoints on the REST API.
All endpoints must be preceded by the baseurl of the server the API is running on.

## Actions

### GET:

| | |
|---|---|
| **/user** | Gets the information about the currently authenticated user. If the currently authenticated user is in the Guardian role, it will return a list of Users in his/hers department and if the user is in the Department role then it returns the Guardian users of the department. |
| **/week** | Get a list of all the weeks available to the current user. |
| **/week/id** | Gets the week, of the current user, with the corresponding Id. |
| **/pictogram** | Gets all pictograms available to the user. |
| **/pictogram/id** | Get the pictogram with the given Id (if authorized) along with its image. |
| **/pictogram?title=string** | Searches for a pictogram with the given string as a title. |
| **/department** | Presents a list of all the departments in the system. |
| **/department/id** | Get the department with the given id. |
| **/department?title=string** | Presents a list of all the departments in the system with the given string as substring in the name. |
| **/choice/id** | Get the Choice with the given Id (if the current user is authorized to see it). |

### PUT:

| | |
|---|---|
| **/user** | Updates the entire current user. The body of the request must contain a GirafUserDTO. Any field not set in the GirafUserDTO will simply be updated to false or null. |
| **/user/icon** | Updates the current user's icon with a new one. The body of the request must contain a png image. |
| **/user/display-name** | Updates the current user's display name. Body must contain the new display name. |

| | |
|---|---|
| **/pictogram** | Updates an existing Pictogram with new information. Body must contain a pictogramDTO. |
| **/pictogram/id** | Updates the image for the pictogram with the given Id, i.e. deletes the old one and creates a new. |
| **/choice** | Update an existing Choice with new information. Body must contain a choiceDTO. |

## POST:

| | |
|---|---|
| **/user/icon** | Creates a UserIcon for the user, can only be used if the user does not have an icon already. The body of the request must contain a png or jpeg image. |
| **/user/resource/userId** | Adds one of the current user's resources to the user with the specified Id. The body of the request must contain a long field called 'resourceId'. |
| **/user/grayscale/bool** | Sets whether the user's applications should run in grayscale mode, based on *bool*. |
| **/user/launcher-animations/bool** | Sets whether the user's launcher should display animations, based on *bool*. |
| **/week** | Creates a week. Body must contain a weekDTO |
| **/pictogram** | Upload a new pictogram - the body of the request must contain a PictogramDTO. All new pictograms are automatically added to both the user's and his department's list of resources. |
| **/pictogram/id** | Upload a new image for the Pictogram with the given Id. |
| **/department** | Creates a new department. The body of the request must contain a departmentDTO. |
| **/department/user/id** | Add a user to the department specified by Id. The body of the request must contain a user. |
| **/department/resource/id** | Add a resource to the department. The current user must be owner of the resource. A resource-id must be specified either in the body of the request or as a query parameter. |
| **/department/resource/id?resourceid=*id*** | Add a resource with id *id* to the department. The current user must be owner of the resource. |
| **/account/register** | Register a new user. The body of the requests must contain four fields; Username, Password, ConfirmPassword and DepartmentId. |
| **/account/login** | Login to the system with a user. The body of the request must contain two fields; Username and Password. Unless already logged in as a Department or Guardian as they can log into their corresponding Guardians and citizens only with their Username as a field. |
| **/account/logout** | Log the current user out of the system. |

## DELETE:

| | |
|---|---|
| **/user/icon** | Deletes the current user's profile icon. |
| **/user/resource** | Remove the resource with the given id from the current user. The body of the request must contain a long field called 'resourceId'. |
| **/user/application/userId** | Removes an application from the given user's list of applicatoins. The body of the request must contain the id of user-application relationship (you may find this id by issuing a get on the UserController). |
| **/pictogram/id** | Deletes the pictogram with the given Id along with its image. |
| **/department/user/id** | Remove a user from the department specified by Id. The body of the request must contain a user. |
| **/department/resource/id** | Remove a resource from the department specified by Id. |
| **/department/resource/id?resourceid=*id*** | Removes a resource with id *id* from the department. The current user must be owner of the resource. |
| **/choice/id** | Deletes the Choice with the given Id. |

Appendix C

# Complete Lists of User Stories for Each Sprint

## C.1 Sprint 1

### C.1.1 User Stories

As a guardian, I would like to

- . . . be able to save my schedule.
- . . . be able to log in to the system.
- . . . choose a citizen to manage.
- . . . create a schedule for the week of a citizen.
- . . . update a week schedule.
- . . . be able to read a week schedule.

Technical tasks in the first sprint include setting up a system to document endpoints (Swagger), updating .NET, including API version number, and improving error responses. It was also necessary to update most pre-existing unit tests.

## C.2 Sprint 2

The user stories for the second sprint of the Giraf project are identical to the user stories from the first sprint.

### C.2.1 Technical Tasks

- Simplify endpoint to add user icon.
- Change development from using SQLite to using MySQL and fix migration files.
- Add `ReadRawPictogramImage` method to `PictogramController` to handle images that are not base-64 encoded.

- Write scripts for automated integration tests.
- Add endpoint for pictogram search. Add pagination for this endpoint.
- Change citizen to guardian relation to a many-to-many relation. Add endpoint for getting a list of all guardians of a given citizen. Add endpoint for getting a list of all citizens of a given guardian aswell.
- Implement the functionality required for letting a citizen change the access level of the pictograms they own.
- Simplify endpoint to create a pictogram.
- Implement token authentication using JSON Web Token (JWT).

### C.2.2 Miscellaneous Tasks

In addition to the technical stories listed above, a lot of minor, miscellaneous tasks were created throughout the sprint with different purposes. These could e.g. be created upon discovery of a bug or by request of another project group. Below is a list of some the miscellaneous tasks that were created during this sprint even though we also did many other minor tasks like bug fixing and additional features that was not recorded on Phabricator most of which can be found by looking at the change-log for release-v1.002.01 [45].

- Create indexes for tables for faster look-up.
- Refactor DTOs to only return necessary information.
- Include image data in GET .../Week. Request by frontend group.
- Add endpoint for getting all citizens in a given department.
- Any citizen is able to update the information of other citizens, this should not be possible.
- Clean up Phriction for old and irrelevant documents.
- Enable Cross-Origin Resource Sharing (CORS).
- Change base URL to include version number.
- Add pictograms to database for test purposes.

## C.3 Sprint 3

### C.3.1 User Stories

- As a guardian, I would like to be able to log in to the system.
- As a guardian, I would like to be able to search for a variety of pictograms.
- As a guardian, I would like to be able to save my schedule.
- As a guardian, I would like to be able to view a week schedule.
- As a guardian, I would like a home-screen where all the citizens' week schedules are shown.

- As a guardian, I would like to access the application from a computer.
- As a guardian, I would like to be able to delete an activity from the schedule.
- As a guardian, I would like to be able to replace the pictogram on an activity.
- As a user, I would like to be able to change the state of an activity.
- As a citizen, I would like my current task to be highlighted.
- As a guardian, I would like to be prompted with a save option.
- As a guardian, I would like to be able to choose a citizen to manage.
- MasterDetailPage for Navigation
- As a user, I would like to be able to lock the screen.
- As a guardian, I would like to have settings for the application and the specific user I am managing.
- Setting - Change the number of activities shown to the user at one time.
- Setting - Change the number of days shown in the week-schedule.
- Setting - Change the number of days in a week which the guardian needs to create.
- Setting - Lock screen orientation to portrait or landscape mode.
- Setting - Change the way an activity is marked as completed.
- As a guardian, when creating a schedule, I would like to be able to choose a template from a selection.
- As a guardian, I would like to be able to copy or move activities.
- As a guardian, I would like to be able to mark activities.

### C.3.2 Subset of User Stories Relevant for the Backend

- As a guardian, I would like to be able to save my schedule.
- As a guardian, when creating a schedule, I would like to be able to choose a template from a selection.
- As a citizen, I would like my current task to be highlighted.
- As a guardian, I would like a home-screen where all the citizen's week schedules are shown.
- As a guardian, I would like to have settings for the application and the specific user I am managing.
- As a guardian, I would like to be able to search for a variety of pictograms.

### C.3.3 Technical Stories

- Add state attribute for weekday resource.
- Remove requirement for weekschedule to always have 7 days.
- Remove ElementIDs from WeekdayDTOs.
- Remove ApplicationOptions, as they are unused.
- Implement simple logger.

## C.4 Sprint 4

### C.4.1 User Stories

- As a guardian, when creating a schedule, I would like to be able choose a template from a selection.
- As a guardian, I would like to be prompted with a save option.
- As a guardian, I would like to be able to copy/move activities.
- Setting - Change the number of days shown in the week-schedule.
- Change the number of days in a week which the guardian needs to create.
- Setting - Change the number of activities shown to the user at one time.
- Force the application in portrait mode when 1-day setting is chosen.
- As a guardian, I would like to be able to add choice-boards to the weekplan.

### C.4.2 Subset of User Stories Relevant for the Backend

- As a guardian, when creating a schedule, I would like to be able choose a template from a selection.
- Setting - Change the number of days shown in the week-schedule.
- Change the number of days in a week which the guardian needs to create.
- As a guardian, I would like to be able to add choice-boards to the weekplan.

### C.4.3 Technical Tasks

Technical tasks in the last sprint:

- Add functionality to support the administrator web page
- Get Git version information i.e. commit hash of the running API.
- Query optimisation.

- Remove support for impersonation and add support for getting information on users according to the role hierarchy.
- Create the admin panel as defined in the requirement specifications for this sprint (see `http://web.giraf.cs.aau.dk/w/releases/2018s4r1/requirements/`).

Appendix D

# Script for doing Optimisation Test

Listing D.1 shows the script used in section 7.2.7 for writing the results before optimisation to a text file. To modify the script for after optimisation, line 14 in listing D.1 should be changed to write to a another file for storing the results for after the optimisation. Furthermore, the `Id` of Kurt should be changed to reflect the `Id` of Kurt in the database.

```
 1 import requests
 2 import time
 3 def auth(token):
 4     return {"Authorization": "Bearer {0}".format(token)}
 5 
 6 start = time.time()
 7 time.clock()
 8 TestUrl = "http://localhost:5000/v1/"
 9 token = requests.post(TestUrl + 'account/login', json = {"username":
       ↪ "kurt", "password": "password"}).json()['data']
10 
11 
12 response = requests.get(TestUrl +
       ↪ 'User/1f4f2da5-85e3-411c-8ad8-40c61ff6873b/week/0/0',
       ↪ headers=auth(token)).json()
13 
14 file = open("beforeOpt.txt","a")
15 
16 file.write(str(time.time() - start) + "\n")
17 
18 file.close()
19 
20 print(time.time() - start)
```

Listing D.1: Optimisation script that makes a login request for Kurt, get the week schedule for Kurt and writes it to a file