UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Faculty of Electrical Engineering,
Computer Science and Mathematics
Department of Computer Science
Database and Information Systems

Master Thesis

by

Jörg Amelunxen
Paulinenstrasse 9
33098 Paderborn
Student Registration Number: 650 44 65

Paderborn, February 28, 2015

HIP– DEVELOPING A WEB-BACKEND OF THE AR-APPLICATION 'HISTORY IN PADERBORN' USING AN AGILE SOFTWARE DEVELOPMENT APPROACH

Supervisor_____
Dr. Simon Oberthür

Examiners_____
Dr. Simon Oberthür
Prof. Dr. Gregor Engels

ABSTRACT

Lorem

ZUSAMMENFASSUNG

Ipsum

ACKNOWLEDGMENT

Libenter homines id, quod volunt, credunt.

Gaius Iulius Cäsar (100 - 44 v. Chr.)

At this point I would like to thank all people who supported me and made this thesis possible in the first place.

Furthermore, I would like to thank all people who proofread my thesis.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ABBREVIATIONS

AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
AR	Augmented Reality
CMMI	Capability Maturity Model Integration
CMS	Content Management System
CSS	Cascading Style Sheets
DOM	Document Object Model
HiP	History in Paderborn
HTML	Hypertext Markup Language
HTML5	Hypertext Markup Language V5

HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
JS	Javascript
JSON	JavaScript Object Notation
MVC	Model-View-Controller
REST	Representational State Transfer
SOAP	Simple Object Access Protocol
SPA	Single Page Application
TDD	Test-Driven Development
UML	Unified Modelling Language
URI	Uniform Resource Identifier
WebGL	Web Graphics Library
WSDL	Web Services Description Language

1

INTRODUCTION IN THE THESIS

This first chapter will introduce the benefits of the system that will be developed within this thesis and will explain the current situation without the system.

1.1 WHAT IS THE CURRENT SITUATION WITHOUT THE TOOL/APP

At the present point in time, guests of the city Paderborn has to look up information about the city in a tedious way, for example using Wikipedia or other existing platforms. On the other hand, people, who want to provide information about the city (e.g., university employees or students), have to provide these information in a general accessible way, again like Wikipedia. Thus, they are limited to the features that are provided by these platforms. Since Wikipedia has been founded in 2004 by the Wikimedia foundation (Wikimedia-Foundation (2014)), most of the used technologies of the web application are nowadays outdated and very general. So, there is a rising need for a new technological updated approach, which is more focused on the specific topic of the city Paderborn and its history.

Especially the use of mobile devices has been risen in this time, which is easily recognizable at the number of sales of the Apple iPhone. The iPhone has been sold 0.27 million times in Q3 2007 and 51.03 million times in Q1 2014 (Statista (2014). Of course, this shift from the device side (i.e., hardware) includes a major shift in (software-) technology as well. Technologies like the nowadays well known Augmented Reality (AR) would not be possible in 2004. Of course, this new technology includes a lot of new opportunities to transfer knowledge between people and cultures. AR is a great example to show how 'the real world' is blended more and more with artificial information; for example in the form

of call-outs and layers. AR is a technology that displays virtual (i.e., digital) information on top of a real object or location using the camera of a mobile device as input for the real objects. So, it ends up to be a combination of both worlds; the real and the digital one. Azuma et. al. has shown a lot of possible fields where the usage of AR would be a great improvement, which includes the field of annotation and visualization (Azuma (1997)). Furthermore, path finding and navigation are fields that could be revolutionized by using AR on mobile devices.

With a simple information website or app like Wikipedia, we include the tedious situation that the person that wants to get to the place he just read about, needs to input the address into another app to navigate him to the position. After he have arrived, he need to switch back to, for example, Wikipedia to manual compare the written information with the object or place he sees in front of him. If the person wants to change the shown information on his mobile device, he does this in general by using the touchscreen of the device. Nevertheless, he is comparing and looking at something that is placed in front of him. This leads to a break within the action and perception space (Hampel (2001)) and is a bad example with respect to the locality of the information (Bondo et al. (2010)). As we will see, AR is a tool that we can use to remove this problem and join the action and perception space while keeping the locality of the information in mind. Furthermore, at the moment we have a lot of unnecessary overhead due to the needed app switching between the information app and the navigation app.

Now, even if somebody wants to publish information about Paderborn on Wikipedia to enable guests of the city to get knowledge about the environment, it is only possible to publish the information as static content (that includes text, graphics, audio and video). On top of that, it is not possible to review the information privately and in enough detail to create university courses that do not include a written paper as the final exam but an entry within such an information system. So, if we would have private annotations within a system that is owned by the university, it would enable the university employees to offer courses that fill the database about Paderborn with high quality content by students.

This leads us to the application that should be prototypical developed within this master thesis, which will be described in the next section.

1.2 WHAT WOULD THE SYSTEM LOOK LIKE (BRIEFLY)

As we will see in chapter **??**, the system will be divided in two big parts. One part is the web-backend, which is connected to an *MongoDB* to store and retrieve the needed information. This backend will provide a Representational State Transfer (REST) interface, which enables it to be connected two different kind of frontends.

These frontends create the second big part of the system. The first prototype of the system will include a web-frontend to access the backend for administrative purposes (e.g., including new data by students, creating groups, review data, etc.) and will be driven by the play framework in combination with AngularJS.

The second kind of frontend, which will be needed in the first prototype of the system, will be the smartphone frontend. With this frontend, the end-users (i.e., everybody who downloads the app from the app-store) are able access the information, which are included in the backend. Furthermore, the smartphone frontend will make use of AR features to show the information that is stored in the backend.

After we have now seen, how the system will look like, we will now take a short look at the question, who will actually benefit of such a system.

1.3 WHAT WOULD BE BETTER IF THE APP WOULD EXIT? WHO WOULD BENEFIT?

On the one hand, users would benefit from the app by having a neat tool to discover the history of the city paderborn. It will be a great experience to be guided trough the city and learn a lot of important and interesting facts about the environment. On the other hand, the system will be a nice variation for the students, which may be bored from the typical *send in a homework to pass the exam* cycle and can include the information directly into the backend and are able to see *their* work some time later via the app in the frontend. So, they are actually able to **do** something, which is used in the future.

1.4 OUTLINE

The second chapter will explain all needed fundamentals of this Master thesis in detail and will show the used frameworks and tools.

The third chapter will outline the application design and describe some general design decisions.

The fourth chapter will show important parts of the actual implementation, used tools and the final Unified Modelling Language (UML) diagrams of the application.

The fifth chapter will show unit tests and the results of a survey that was used to evaluate usability of the system.

The sixth and last chapter will deal with arisen problems and will discuss the development for future work.

2

TECHNICAL AND METHODOLOGICAL BACKGROUND

> *All the revision in the world will not save a bad first draft: for the architecture of the thing comes, or fails to come, in the first conception, and revision only affects the detail and ornament, alas!*
>
> *(T. E. Lawrence (1931))*

Because an idea about the system and the background of the used technologies is very important (just like the quotation above shows), the following two chapters will contain details about the planned system.

In general, it should be able to handle the complete workflow for a university course situation. So, the backend should offer features that offer groups of students the work on specific topics. The domain model showing this situation a general university course is shown in Figure 1.

model is wrong

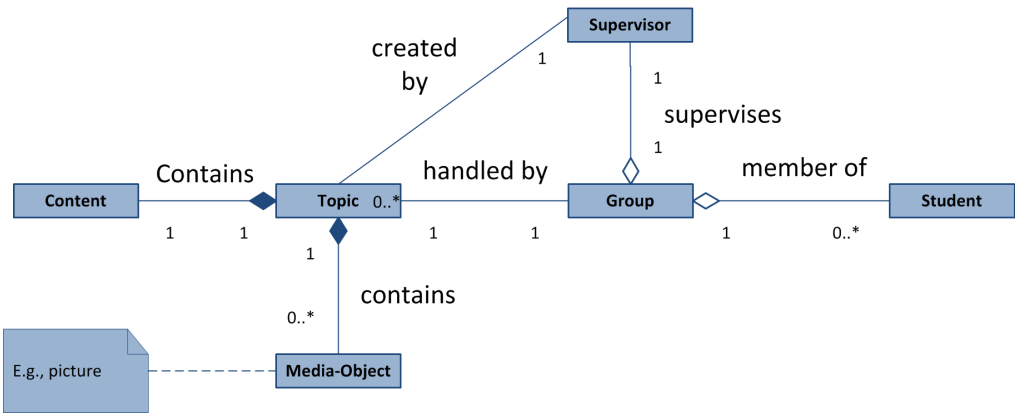

Figure 1: The domain model of the university courses

But before we come into more detail about the software itself, we will now get an insight into the used technical frameworks respectively tools and methodological concepts, which are used during the development process. Note that

we will focus on the frameworks and plugins that are needed to understand the details of the implementation in chapter 4. Although, the whole system uses a lot more on its whole.

The first thing, which is important to notice, is that the system will be developed with an agile software development method, which will be based on SCRUM. Because of that, we will start by explaining the SCRUM development method.

2.1 AGILE DEVELOPMENT – SCRUM

The main idea of agile development is described within the agile manifesto Beck et al. (2001)[1]. Within agile development, the focus is set to frequent software delivery and close customer relationship. Because Scrum is such an agile development method, we find similar ideas in the Scrum development process.

A big problem within the software industry is that projects take longer than expected and the expectation of the customer differs more and more from the view of the development team because doing a 6 month part of requirements engineering, followed by an 18-month development period lets one finish a product that is already obsolete at the day it gets published.

Similarly, the HiP-application will be developed in a Scrum-like fashion to achieve a high efficiency in the small timeframe. However, a full Scrum approach is not possible because the development team is very small. Nonetheless, we will include the ideas and concepts of the Scrum process but will, for example, combine different roles in one person. But to get a first intuition about the Scrum process itself, we will now describe the main ideas of Scrum and agile development in general.

Using Scrum means, the application will be developed within autonomous short *sprints* with a length between 1 up to 4 weeks. However Berczuk (2007) points out that a four week sprint is in many cases problematic because a lot of customer requests have to be included into currently running sprints and so the backlog becomes more and more useless; he suggests having about 2 weeks long sprints. In any case, after every sprint the product should be more refined (Schwaber and Beedle (2002)). However, it should be possible to execute the application at the end of any given sprint, which will result in a fast and stable development process. The general development process is also shown in Figure 2. The Product-Backlog is the foundation of every sprint-backlog because it

1 The whole manifesto can be found here: http://agilemanifesto.org

contains every feature that will be needed in the product at some time. So, one derives the sprint-backlog, which includes every feature that should be added in a specific sprint, from the product-backlog for every new sprint. In addition to that, daily Scrum meetings should ensure that the whole team is up-to-date and as efficient as possible. In more detail, Scrum is known to reduce every category of work (i.e., defects, rework, total work required, and process overhead) within a Capability Maturity Model Integration (CMMI) compliant development process by almost 50% (Sutherland (2009)), which is also great for development in this short timeframe. The close customer relationship can for example be found in the fact the the customer is often invited in meetings after the sprints to see the progress on the product and, more important, to be able to influence the development process. However, Paulk (2002) claims that customers may also create a threat to finish agile development successful if they are not able or willing to maintain such a close relationship with the development team.

The development process also influences the order of the development because the chose of Scrum indicates that we will develop the front-end and back-end in parallel. We will use a Test-Driven Development (TDD)-like approach within every sprint (where possible) because we will need to adapt and refactor existing code often. So, we will at first create needed test cases and afterwards implement against these test cases. This approach will prevent that testing of the application will be shifted into the last week(s) of the development process and done in a superficial way. In addition to that, a comprehensive test suite is a great basis for further development (Maximilien (2003)).

Now, after we have seen the general concept of Scrum, we will now take a look at methods for cost estimation.

2.2 METHODS FOR COST ESTIMATION

To get a feasible estimation of the workload of a given backlog, as it has been described in section 2.1, we need some methods to create a good work- respectively cost-estimation. This is important to be able to choose a fitting amount of work per sprint.

As Keaveney (2011) points out, one of the main principles of agile methods is to have meetings with the customer within the development phase to adapt the requirements if needed (**?**). However, changing requirements within a currently running software development are a common cause of problems with respect

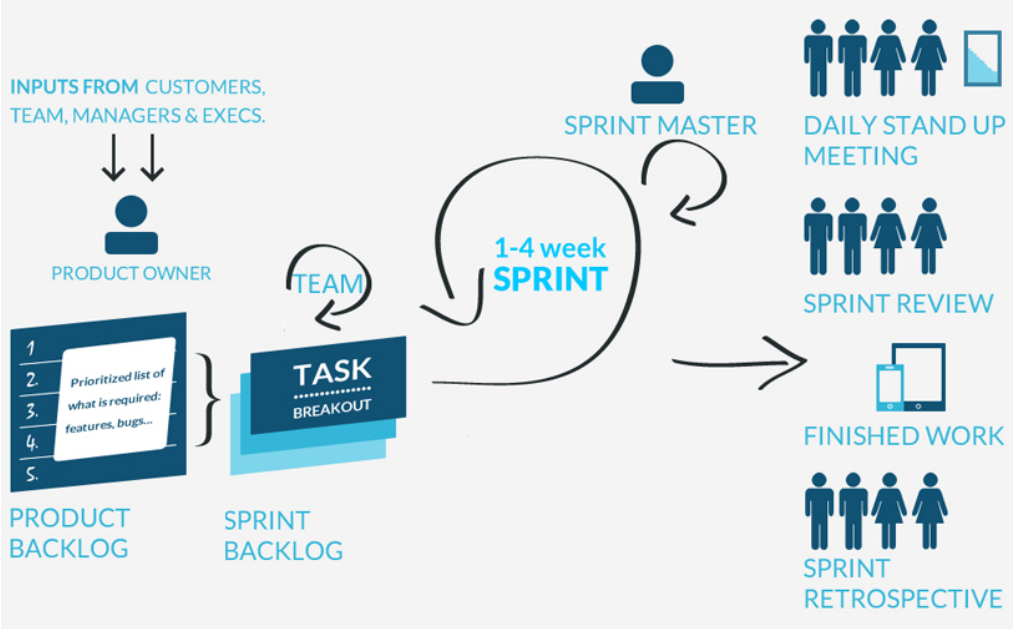

Figure 2: The diagram shows the Scrum development process. (Simplified, original work from Maxxor (2015))

to software cost estimating (Jones (2003)). So, our methods for cost-estimation has to be able to handle changing requirements in short time-frames. Similarly, surveys have shown that in real life scenarios, the techniques used to estimate costs in agile development projects are in general based on the expertise of the team-members. So, the developers look to past iterations (or even past projects) to produce estimates about the costs of the current project respectively sprint (Ceschi et al. (2005)).

To be able to formalize knowledge about past iterations and to be able to compare the data with the current iteration, one can use diagrams like burn-down charts.

2.2.1 Burn-down charts

A burn-down chart shows the amount of remaining story points (which correspond to a specific estimated time-frame, like one story point per 30 minutes - *Story points will be explained in the next section. For now, see them as a unit of needed time* -) on the y-axis and the days of the sprint on the x-axis. With such a chart, one can estimate the remaining time, and can derive if the sprint can be finished in the given time-frame, by looking at the *slope* of the graph. Figure

3 shows an example burn down chart over 8 days. The OPT line shows the optimal slope that ends up on 0 remaining story points on the last day. If the sprint curve is above the OPT line, you are to slow in the current sprint and in the case the sprint curve is below the OPT line, you are ahead of the time. Obviously, burn-down charts do not have problems with changing requirements outside of the current sprint. But, in addition to that, we can also include new tasks to a running sprint and can simply adapt the OPT line with its slope to track the new added tasks within the active sprint.

add user stories - basis : see wikipedia

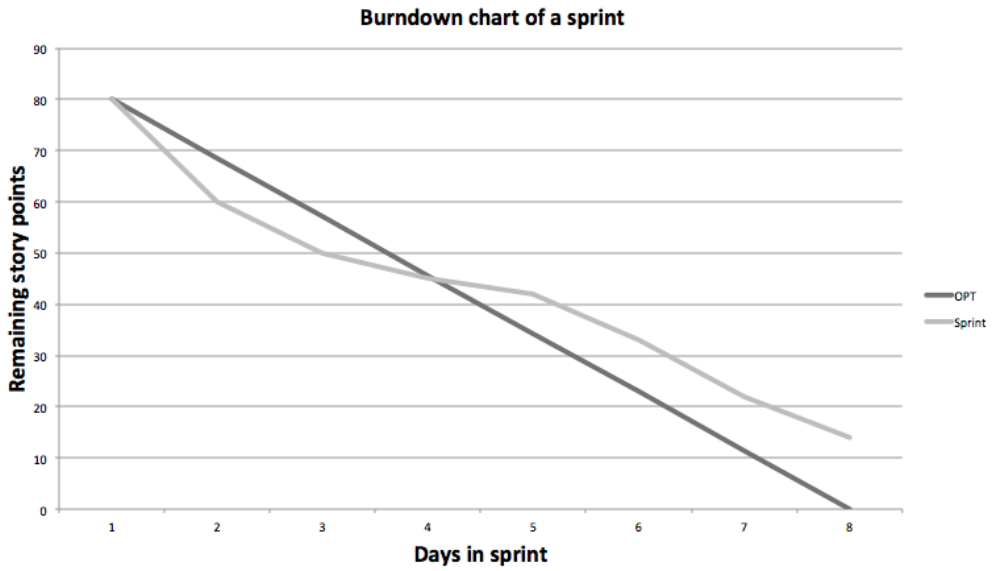

Figure 3: The burndown diagram shows an example sprint.

2.2.2 Story points

In the previous section, we have implicitly used *story points*. Story points are tightly coupled with the idea of agile development because they are a unit of needed time which takes the team members effort and the subjective degree of difficulty into account (Danube (2014)). This cannot be done by a manager, who is estimating the time and assigning tasks based on conjectures about the team performance. The main idea about story points is that the actual amount of time which is expressed by a story point does not matter. The important thing is that the collaborating teams share a common understanding of its scale. So, every member of the teams should be comfortable with the represented time.

Another idea is that you can easily imagine the analogy 'this story is like that story, so it well take about the same time', which tries to improve the quality of time estimations Cohn (2004).

2.3 DEVOPS

Besides the cost estimation, the deployment process is an important detail within the software development process. It determines the speed in which new features are shipped to the customer (respectively released to an online service, which is in general the same).

Now, DevOps can be seen as the practice of *operations* and *development* engineers participating together in the entire service lifecycle, from design through the development process to production support (Mueller et al. (2011)). This approach is, for example, used by Netflix, Amazon and Etsy to deliver their new features in a fast way to the enduser (Duvall (2012)).

Using a DevOps approach results in a couple of benefits like the fact that you learn about your induced problems much faster because you get much faster feedback from your enduser. Obviously, it is much easier to fix a bug in such a situation, as in the case that your customer detects a bug in a 'new' release, which is four month old on the development machines. This results also in the benefit that you problems are much smaller and need less time to be fixed (Duvall (2012)). Furthermore, the focus on DevOps leads to the usage of tools that offer the possibility for new ways of structuring the architecture of the system (Cukier (2013)). As soon as you start thinking in much smaller autonomous chunks, it becomes much easier to migrate your system into a cloud service in a **PaaS!** (PaaS!) like fashion (Cukier (2013)).

One of the tools you need to create a DevOps process (respectively a DevOps culture), is **CD!** (CD!).

2.3.1 Continuous Delivery and Continuous deployment

CD! is the automated implementation of the build, deploy, test and release process. A CD! process runs the software tests on every version that is committed to the version-control system and provides a quick automated feedback on the test result. By using a CD! process, the release of a new software version becomes easy and fast. Thus, CD! is the needed brick for creating a continu-

ous deployment process. Within such a process every commit that is done by the developer and send to the code-versioning software becomes automatically shipped to the enduser (or at least parts of them - like only in North America for a specified time) by using the CD! system. However, a continuous deployment process cannot be used in every situation but can be a very important thing for fast paced companies like Netflix or Flickr.

Now, after we have gained some insights about the process itself, we will take a look at the frameworks that will be used in the development process.

2.4 USED FRAMEWORKS

Because the time frame for the project is quite small, it is not possible to create the whole application from scratch and, thus, we need a couple of frameworks to accelerate the process. We will use the Play-Framework in the backend, which offers a REST-interface and handles the routing from Hypertext Transfer Protocol (HTTP)-requests to application code. On the frontend-side, AngularJS will be used to create a fast and responsive web-interface and Junaio will be used to include the AR-functionality on the smartphone application.

2.4.1 Play Framework

We will use the Play framework for the backend of the application because Play is an open source web application framework, which is written in Scala and Java, follows the Model-View-Controller (MVC) architectural pattern and handles the routing from HTTP-requests to application code.

A simple example for the routing configuration file is shown in Listing 2.1. In this file, each documented route consists of an HTTP method and Uniform Resource Identifier (URI) pattern that is linked to a call of a, so called, *action method* within the Java respectively Scala code. As one can easily see in line 9, it is quite easy to pass parameters to the application code. Furthermore, one can see that the parameters are type-safe; thus, for example, a String passed in as an Integer would result in a compilation error.

Listing 2.1: Simple routing configuration file within the Play Framework

```
1 # Routes
2 # This file defines all application routes (Higher priority
    routes first)
```

```
3  # ~~~~
4
5  # Home page
6  GET /                controllers.Application.index()
7
8  # Usage of parameter
9  GET /thesis/:grade   controllers.Application.exp(grade: Integer)
10
11 # Map static resources from /public to the /assets URL path
12 GET /assets/*file    controllers.Assets.at(path="/public", file)
```

Very briefly, the Play framework includes three different parts:

1) Java Code that implements the controllers. The controllers are used to handle requests that get routed to them via HTTP. A simple controller is shown in Listing 2.2. As one can see within line 10 of Listing 2.2 the String "Your new application is ready" gets passed to the render function of the class index and returned as a parameter within the *ok* function, which creates a simple HTTP header with return-code 200. The index class is a Scala class that gets automatically creates from the Scala/Hypertext Markup Language (HTML) template called *index.scala.html*. We will see this in more detail within point 3 of this list.

Listing 2.2: Simple Java-controller within the Play Framework

```
1  package controllers;
2
3  import play.*;
4  import play.mvc.*;
5  import views.html.*;
6
7  public class Application extends Controller {
8
9      public static Result index() {
10         return ok(index.render("Your new application is ready."
               ));
11     }
12 }
```

2) Java Code that implements the model entities. The model is used to do the actual calculation and data handling. In essence, we should include as less application-logic as possible within the controllers and use these model classes instead.

3) Scala templates that are used as *views*. As a return value of the controllers, they pass data to a fitting template and return a corresponding HTML-view. However, we may also skip the template engine sometimes to directly return

JavaScript Object Notation (JSON)-documents, which can be used to provide a Application Programming Interface (API). Listing 2.3 shows the used *index.scala.html* template used in point 1 of this list. In line 1, we declare the used parameters, in our case one String variable, which we have used to pass the String "Your new application is ready". The *@main* command in line 3 calls another template, which includes everything beside the HTML body. The body of the file is now included in line 4 by calling another framework specific method, which includes a welcome and documentation message and renders our passed String variable.

Listing 2.3: Simple Scala template within the Play Framework

```
1 @(message: String)
2
3 @main("Welcome to Play") {
4     @play20.welcome(message, style = "Java")
5 }
```

Besides the templating feature, Play can be augmented with Plugins that handle specific behavior. For example, the Plugin *SecureSocial 2* is able to handle the whole user registration and login process and offers and interface to get the data from users, who are currently logged into the system. In more detail, it offers out of the box support for web-services like Twitter, Facebook, Google, LinkedIn and GitHub. Furthermore, *SecureSocial 2* provides a Username/Password mechanism (will will be used by the HiP backend) with signup, login and reset password functionality.

As another important fact, Play emphasizes the usage of the REST principle, as it can be seen within the routing configuration file. We can easily and directly make use of the different HTTP commands and use them to structure our API accordingly. In general, Representational State Transfer (REST) is a style of software architecture that is used to build distributed systems and has been introduced in the dissertation of Fielding (2000). As Rodriguez et. al. point out, REST based web services are easier to use than Simple Object Access Protocol (SOAP) and Web Services Description Language (WSDL)-based ones and getting more and more importance since mainstream web 2.0 service providers are taking up on REST (Rodriguez (2008)). Furthermore, the Play-framework comes with integrated unit testing and full support of asynchronous I/O. So, all in all, Play will noticeably enhance the development speed of the backend.

2.4.2 MongoDB

As Trelle (2014) describes, data entries within a MongoDB are called documents and are in essence ordered sets of key-value pairs. However, values can also be complete documents and arrays, so one can store complex hierarchical structures within a MongoDB. Similarly, the data gets stored in a **BSON!** (BSON!)-format, which is a binary JSON format with more datatypes and better traversability.

These documents are stored in so called collections, which are in general comparable to the tables in a relational database, like **MySQL!** (MySQL!).

For every document within the database we need to include a field called *_id*, which contains the primary key of the document. Of course, this primary key has to be unique within the collection that contains the document. If an document is stored within the database without a *_id*field the field gets automatically generated.

An insert into a MongoDB is easily done and shown in Listing 2.5.

Listing 2.4: Inserting into a MongoDB

```
1 var db = ... // contains the connection to the database
2
3 var object = {
4   firstname : 'John',
5   lastname : 'Doe'
6 };
7
8 db.hipUsers.insert(object);
```

Similarly one can read documents from the database by creating a document that is matched against the collection. For example, to retrieve the user *John Doe* that has been included within Listing 2.5 by matching against his lastname, one would need to do the steps shown in Listing **??**.

Listing 2.5: Reading documents from a MongoDB

```
1 var db = ... // contains the connection to the database
2
3 var object = {
4   lastname : 'Doe'
5 };
6
7 db.hipUsers.find(object);
```

Now, within the following section, we will take a look at the frontend technologies.

2.4.3 AngularJS

After we have now seen the basics of the Play framework and the MongoDB, which will handle the backend functionality, we will now take a look at *AngularJS*, which will provide needed features to create a fast and responsive frontend. The frontend will be designed as a Single Page Application (SPA). A SPA is in general an orthogonal approach to the common way of creating websites as a set of linked pages. A SPA is a composition of individual components which can be updated respectively replaced independently of the complete site and, thus, without any reload after the actions of the user. This results in a couple of benefits, like improved interactivity, responsiveness and user satisfaction (Mesbah and van Deursen (2007)). Another important fact with respect to the system performance is that AngularJS offers functions (as we will see: *directives*) to circumvent the need for changing the Document Object Model (DOM)-Tree directly. AngularJS relies on a **MVVM!** (MVVM!) architecture and tries to create the same behavior by doing changes to the ViewModel. The ViewModel sits behind the concrete **UI!** (UI!) layer and exposes data needed by a View from a Model. Because of that the ViewModel can be viewed as the source our Views go to for both data and actions.

Listing 2.6: Simple example that shows the use of an AJAX request that shows the reponse text within a specific div container

```
1 xmlhttp.onreadystatechange=function(){
2   if (xmlhttp.readyState==4 && xmlhttp.status==200){
3     document.getElementById("myDiv").innerHTML=xmlhttp.
          responseText;
4     }
5   }
6 xmlhttp.open("GET","ajax_info.txt",true);
7 xmlhttp.send();
```

Obviously, creating a SPA implicitly forces the usage of some kind of request mechanism to get the data that the user needs, without reloading the site. This could for example be done with an Asynchronous JavaScript and XML (AJAX) request like the one that is show in Listing 2.6. The listing shows how the request is being made in line 6 and 7. Line 3 shows the exchange of the content of the div container with the id *myDiv*. However, using AJAX is cumbersome

and can nowadays easily be hidden in very sophisticated frameworks, like AngularJS.

The *AngularJS* framework will be explained briefly in the following. AngularJS makes heavy use of expressions and directives. While directives in AngularJS are functions that get run when the DOM is compiled by the compiler and are shown as simple tags or attributes, an expression is a term that is encapsulated by *{ { ... } }* and gets evaluated while the page gets loaded.

Listing 2.7: Simple example that shows the use of expressions

```
1 <div class="panel-heading">
2   {{lc.getTerm('system_group_navigation')}}
3 </div>
```

Listing 2.7 shows a simple example, where an expression is used to call a method of a controller object. As soon as the page gets rendered, the DOM-tree will be loaded with the result value of the given javascript function called *getTerm(String)*. Another major feature of AngularJS is the so called two-way data binding, which is closely coupled to expressions. The two-way data binding ensures that the rendered value of the function getTerm(String) gets automatically updated, as soon as the function returns a different value.This creates a source-code that includes less unnecessary lines of code for updating the values in the view.

Furthermore, AngularJS offers directives like *ng-class* which adds dynamically a specific class to a DOM-element if a given expression evaluates to true.

Listing 2.8: Simple example that shows the ng-class directive to change the style respectivly color of an alert depending on its type

```
1 <div ng-class="{'alert-warning' : alert.type == 'warning',
2                 'alert-danger' : alert.type == 'danger
                      ',
3                 'alert-info' : alert.type == 'info',
4                 'alert-success' : alert.type == '
                      success'}"
5      role="alert">
6
7   {{alert.msg}}
8 </div>
```

Listing 2.8 shows how the *ng-class* directive exchanges the used style of the alert depending on the boolean expression that is places behind the ':'. So the syntax of the attribute value is { class : expression }.

Of course one can also create own directives to get a much cleaner code. For example, it is possible to create a directive called *chat-box* which can directly included within the DOM-tree. Thus, the usage of the created chat element folds down to the code that is shown in Listing 2.9. The same can be achieved by using web components or Google polymer, which is in essence an extension of the web-components technology. However both technologies are, at the present point in time, only fully compatible to Google Chrome. Because of that we will use custom AngularJS directives to create clean and maintainable code.

Listing 2.9: Simple example that shows the usage of a custom directive

```
1 <!-- some code up here -->
2
3 <chat-box> </chat-box>
4
5 <!-- some code down here -->
```

Besides AngularJS, Twitter Bootstrap will also play an major role in the frontend development process.

2.4.4 Twitter Bootstrap

Twitter Bootstrap[2] is an open and freely available collection of tools on the basis of the HTML, Cascading Style Sheets (CSS) and Javascript (JS) and can be used to support and accelerate the building of web applications. We will use Twitter Bootstrap inside the user front-end of our web application because it works nicely together with AngularJS and can for example be used to create tabs and alerts. Furthermore, Twitter Bootstrap is nowadays used a lot by common web-applications and, thus, we enhance the external consistency of the HiP-application in respect of other web-applications, which may be well known to the user.

ref external consistency

Twitter Bootstrap is licensed under the terms of the Apache License v2.0[3].

Furthermore, Bootstrap can be used with Bootstrap UI, which are Bootstrap components that have been written in AngularJS and can easily be reused. Examples for these components are tooltips, datepickers, timepickers, etc. So,

2 Twitter Bootstrap is hosted on GitHub and can be downloaded here: https://github.com/twitter/bootstrap

3 The terms of the license can be found here: http://www.apache.org/licenses/LICENSE-2.0

this is also a great repository for components to accelerate the development process. [4]

2.4.5 Junaio - Metaio

The AR-functionality will be offered by the framework Junaio. The company Metaio, which runs Junaio, offers a developer program to develop own applications on the basis of the Junaio (eco-)system. Moreover, it is completely free of charge for the developers (Junaio (2014)). However, deployed apps will be shipped with a Metaio watermark inside as long as you do not buy a specific license.

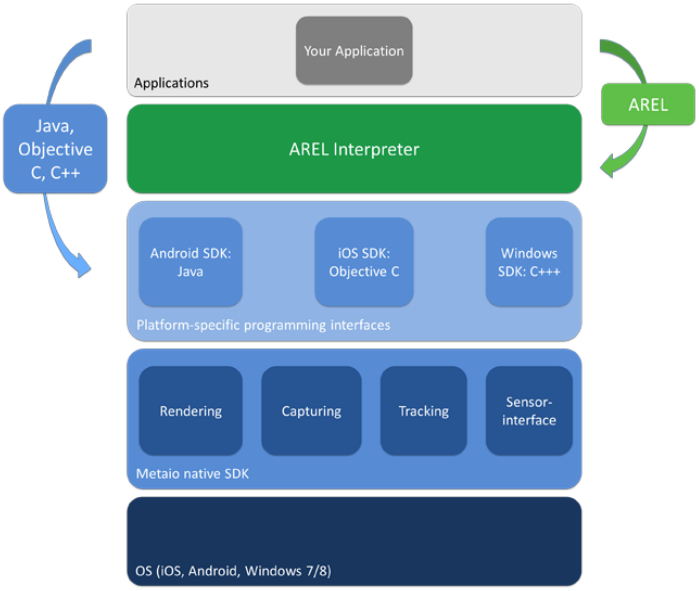

Figure 4: The figure shows the placement of the AREL interpreter within the platform stack. (Taken from Dev.metaio.com (2015))

Furthermore, Metaio has developed a JavaScript binding of the SDK used for AR-applications called **AREL!** (**AREL!**) which can be used as a platform to write your AR apps without writing platform specific code of the mobile operating system Dev.metaio.com (2015). Figure 4 shows the placement of the **AREL!** interpreter within the platform stack.

4 Bootstrap UI can be downloaded here: http://angular-ui.github.io/bootstrap/ and is distributed under the MIT license https://github.com/angular-ui/bootstrap/blob/master/LICENSE

To use **AREL!** we need three different parts that that get combined to an **AREL!** application.

Listing 2.10: Example for the HTML5 layer

```
1  <html>
2     <head>
3        <!-- Integrates the arel javascript bridge -->
4        <script type="text/javascript" src="http://dev.junaio.com
            /arel/js/arel.js"></script>
5
6        <!-- Includes application logic -->
7        <script  type="text/javascript" src="logic.js"></script>
8     </head>
9
10    <body>
11    </body>
12 </html>
```

First of all, we need a static content definition, which is a **XML!** (XML!) document that references the models and graphics that will be loaded when we start the **AREL!** application. The second part is the Hypertext Markup Language V5 (HTML5) layer, which is also addressed in the static content definition. The HTML5 binds the static content definition to the application logic and may also add additional **GUI!** (GUI!) functionality. An example for such an HTML5 layer is shown in 2.10. The last part, the application logic is written in JS and loads objects that are afterwords tracked by patterns.

So, **AREL!** allows scripting of AR-applications on mobile operating systems like iOS or Android based on common web technologies such as HTML5, **XML!** and JavaScript.

2.4.6 WebGL

Last but not least, we will need Web Graphics Library (WebGL) to create a possibility to render and manipulate the 3D-point clouds, of the scanned objects, right within the browser.

2.5 TESTING TECHNIQUES AND TOOLS

Because we use an agile development approach, testing becomes an important aspect even in the development process itself. This founds on the core aspect of agile development that even in early stages of the development process the requirements are going to slightly change and, thus, we need to adapt the existing code. This leads us to TDD, which is a developing technique which relies on the heavy use of tests. TDD will be explained in the following section.

2.5.1 TDD

The main idea of TDD is that one develops the test cases upfront and implements the needed functions afterwards. This is a major shift in the way software gets developed as, traditionally, unit testing has been done on exiting code, after it has been implemented. According to **nerur2005!** (nerur2005!), this TDD approach leads to code that is more understandable and maintainable. However, TDD is not only a different testing technique. As the definition of the Agile Alliance (Alliance (2015)) states *"Test-driven development" refers to a style of programming in which three activities are tightly interwoven: coding, testing (in the form of writing unit tests) and design (in the form of refactoring)*. So, TDD is not only a testing technique, it is a programming technique which follows a couple of rules to achieve a tight coupling of coding, testing and design. While the two parts coding and testing are easy to grasp, design seems to be a bit different to grasp it with a technique that relies on testing. However, as Janzen and Saiedian (2005) points out, while writing a test one is deciding what the program should do, which is an analysis step. This is how analysis gets coupled with testing.

Furthermore, Janzen and Saiedian (2005) states that the positive aspects of the usage of TDD have also been shown in studies of the **NCSU!** (NCSU!), which has performed a couple of empirical studies (George and Williams (2004), Maximilien (2003), Williams et al. (2003)) on TDD in industry settings. These studies showed that programmers who used TDD to produce code created 18 up to 50 percent more external test cases than code that has been produced by corresponding control groups. The studies also reported that the TDD developers spent less time while debugging their code. Nevertheless, they reported also that the TDD project took up to 16 percent longer. But, in the case that took 16 percent more time, researchers noted that the control group without TDD wrote far fewer tests than the TDD group.

According to **GAA2015!** (GAA2015!) the TDD process can be expressed with the following set of steps:

1. write a "single" unit test describing an aspect of the program
2. run the test, which should fail because the program lacks that feature
3. write "just enough" code, the simplest possible, to make the test pass
4. "refactor" the code until it conforms to the *simplicity criteria*
5. repeat, "accumulating" unit tests over time

Note that the *simplicity criteria* within step 4 of the procedure has been defined by Beck (1999) as: *At every moment, the design runs all the tests, communicates everything the programmers want to communicate, contains no duplicate code, and has the fewest possible classes and methods. This rule can be summarized as, S̈ay everything once and only once.¨.*

So, all in all, the TDD process relies on writing unit test before writing the application code itself and use them as tests in the developing phase to check if the currently written code is able to fulfill the requirement. Step 5 shows, that the sum of all test cases is then also used in a *traditional* way to find bugs in the existing application-code.

2.5.2 Jasmine and Karma

Testing will be a major part of the development process. Thus, the chose of a good test environment is important. Karma is a test runner, which can be extended with a couple of plugins (e.g., code-coverage, more available browsers, etc.) that allows you to execute JavaScript code in multiple real browsers. The tool has been created by the team that has created AngularJS and is, thus, suggested as the main test runner within an AngularJS environment (AngularJS (2015b)).

Furthermore, Karma can be used with an extension called *Karma-jasmine* to run Jasmine test cases. *Jasmine* is a behavior-driven testing framework that can be used to test JavaScript code. As a brief explanation, **BDD!** (BDD!) is a development method that has been evolved from TDD to get the idea to a bigger audience and to shorten the gap between behavior (which can easily be explained to the customer) and code North (2012). Thus, we will use Karma with the Jasmine extension to run our Jasmine test case for the application.

2.6 TOOLING

A couple of frameworks and techniques is a good start for creating such a sophisticated system, however, we will also need fitting tooling to support the development. These tools will be described in the following section.

2.6.1 Git

We will use Git, which is a commonly used distributed revision control and source code management system, for the versioning of our source-code. Git is free software distributed under the terms of the GNU General Public License version 2.

The service GitHub offers his users the possibility to maintain public and private Git repositories. The usage of GitHub is free, if the user uses public repositories only. We will use GitHub to host our source-code.

2.6.2 Jira

Jira is a proprietary software for project tracking purposes, which has been developed by the company Atlassian. It has support for agile development methods like Scrum or Kanban and offers a couple of features for bug tracking and time respectively cost estimation, like burn-down charts, which has been explained in section 2.2.

We will use Jira to track the status of the HiP application.

2.6.3 IntelliJ IDEA

IntelliJ IDEA is a Java Integrated Development Environment (IDE) by the company JetBrains.

The current version offers support for Java 8, UI designer for Android development, Play 2.0 and Scala and is, thus, a good choice to work with because all of these features will be used in our development process.

The IDE is available as an Apache 2 Licensed community edition and a commercial edition. The commercial edition can also be downloaded for free for educational purposes.

After we have now seen all needed fundamentals to grasp the main idea of the application, we will take a look at the draft of the application.

3

DRAFT OF THE APPLICATION

Within this thesis, we will develop an application (with focus on the backend system) to handle all these problems that have been described above within the section about the current situation. This master thesis will handle the planning respectively cost estimation of the different parts/features of the general system and will, after the needed technologies/frameworks are elaborated and evaluated, include a prototypical implementation of the needed components of the backend system. But at first, we will start with the first step in a development process; the requirements engineering.

3.1 REQUIREMENTS ENGINEERING

Because the HiP-Application will be developed closely together with our *costumer*, other working-groups at the university of paderborn, the whole process starts with the requirements engineering phase.

First of all, a requirement is defined as "[...]A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents[...]" (IEEE (1990)).

We started the development of the application with a requirements engineering meeting together with our *customer* and ended up with a couple of cards with written user stories. Afterwards, these stories got refined to concrete high level requirements, which are measurable and prioritized. A complete list of all requirements, which were derived from these user stories, can be found within the appendix in tables 5, 6, 7, 8 and 9. These requirements can directly be used to derive test cases from them, which is good because we will use a TDD driven development approach in every sprint.

Because the system will be quite complex and big, we will need a couple of frameworks and libraries, which have mostly been explained in section 2. However, we will now take a look at the users of the planned system.

3.2 USE CASES – USER STORIES

The system on its whole will have four different kinds of users, which correspond to four different roles within the system. The roles are briefly described in the following:

1. Supervisors: Supervisors work at the university and create groups, topics and are able to review the information of their groups. The main goal of supervisor is the supervision of groups.
2. Students: Students are placed in groups by their supervisor and work on a specific topic. They are able to hand in their work for review by the supervisor.
3. Admin: The admins are able to assign users to specific roles and edit the system itself (e.g., edit translations, etc.)
4. Master: People, who have the role *Master*, are able to accept respectively reject topics for the front-end application

These four different roles are also shown in Figure 15 an **UML2!** use case diagram, which is placed in the appendix. The figure shows the roles together with the functionality that gets invoked by these roles.

After we have now seen the different roles that are involved in this system, we will include two use case tables as an example for the usage of the system here. However, a complete list of use case scenarios would be to large for this thesis.

Table 1: Use Case Scenario: student changes content on a topic

Step:	Involved:	Description of the activity:
0	Student	logs into the system
1	Student	navigates to the correct group
2	Student	navigates to the topic he wants to work on
3	Student	changes content on the topic
4	Student	saves the changes
5	Student	logs out

Table 1 shows that when a student wants to change the content of a topic he is working on, he needs to log into the system, navigate to the group and topic and is then able to change the content.

Table 2: Use Case Scenario: Supervisor creates a new group

Step:	Involved:	Description of the activity:
0	Supervisor	logs into the system
1	Supervisor	navigates to create group view
2	Supervisor	inputs name, member, topics, etc.
3	Supervisor	saves the group
4	Supervisor	logs out

The second use case scenario in Table 2 shows that it will be quite easy for a supervisor to create a new group. He only needs to log into the system, open the correct view and input all needed information, like members of the group.

include simple sequence diagrams

Now, after we have seen what the application is about, we will now take a closer look at the planned architecture.

3.3 ARCHITECTURE OF THE APPLICATION

The architecture design in agile projects is slightly different from the design in common respectively classical development projects. Within a classical development process, one would design the architecture of the system in its whole, before the actual programming phase is started. This is, obviously, not applicable within an agile development approach. As Mast (2013) points out, agile architecture design has the the following important attributes:

ref

1. At the beginning, one has only an idea about the architecture, which describes the most important constraints. However, there has to be enough space to be able to adapt the architecture to new or changing requirements within the development process.
2. This enables the developer to be able to use an iterative development style and to postpone important development choices to the *Least Responsible Moment*. The *Least Responsible Moment* is the latest possible point in time, where you can implement an architecture decision.
3. This way, detailed structures and technical concepts are created on-the-fly, while the application gets developed.

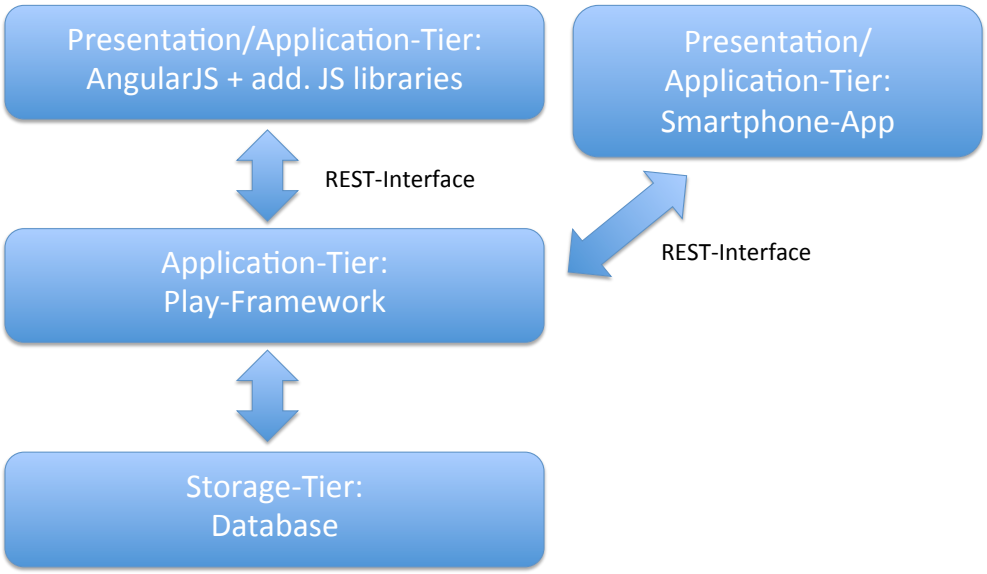

Figure 5: The general 3-tier architecture of the HiP-application with both presentation tiers

That said, we will choose a 3-tier architecture for the development for the application, which is, according to Eckerson (1995), a quite common for Client/Server respectively web applications. Within our application, the 3-tier architecture is nice because it enables us to exchange the presentation tier easily, which is a feature that we will need to support the web-backend and a smartphone front-end. The main idea about the architecture is shown in Figure 5. The figure shows the storage tier, which will be driven by a MongoDB. The data that will be stored in the MongoDB gets prepared by the Play-Framework, which will create the foundation for the application tier. Nevertheless, we want to build a fast and application that is able to give instant feedback to the user within the **UI!**. Because of that, parts of the application tier need to be included on the client side within the presentation tier. Because of that, the presentation tier will be much more complex as it is shown in this brief draft since AngularJS, which is used in this tier, relies on a **MVVM!** architecture on its own but this will be explained later. However, more detailed decisions about the architecture design are postponed to the *least responsible moment*, as it has been suggested by Mast (2013).

Now, we have an idea about the general architecture. However, we need to take a look at the usage of the different components (e.g., frameworks, libraries and tools) and how the will work together.

3.4 USAGE OF THE COMPONENTS

As we have seen in section 3.3 a lot of application logic will be included in the first tier (i.e., presentation tier). To offer all needed features of the backend in this short time frame, we will need to include a couple of frameworks and libraries. A brief overview about the components and how they work together is shown in Figure 6.

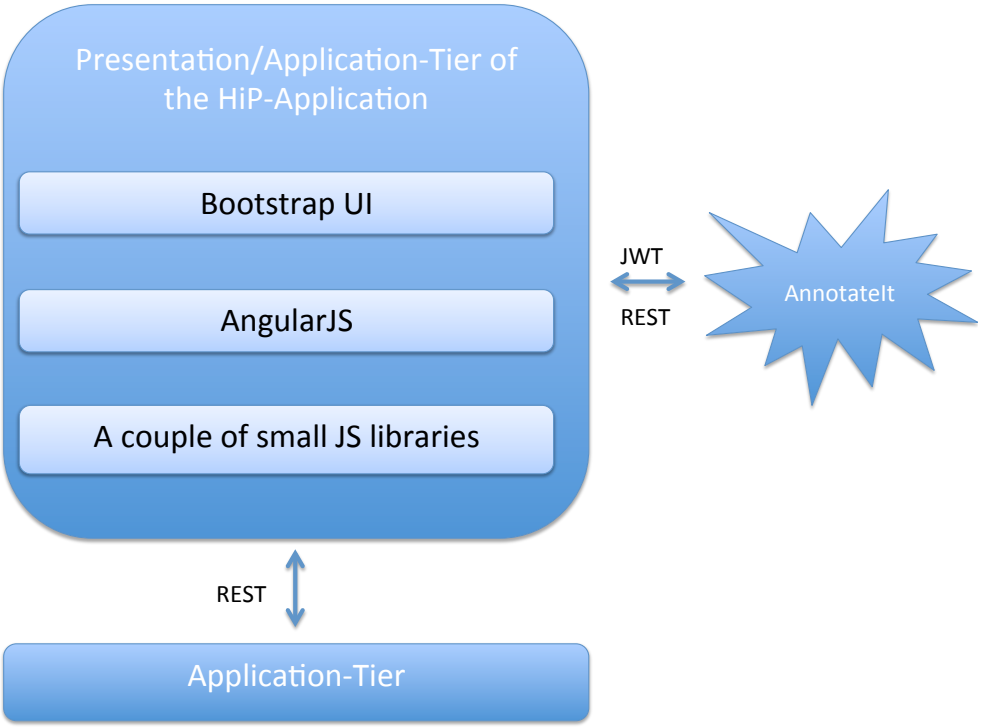

Figure 6: The usage of the different components and how they work together

The figure shows that the backend system will make use of AngularJS and Bootstrap UI to create the UI itself. Furthermore, These two components will be supported by a couple of smaller JS libraries for highlighting (i.e., *AnnotatorJS*) or word processing environments (i.e., *textAngular*). To create highlighting with AnnotatorJS that is persistent, we need to send the information via

another REST interface to storage service. However, this creates the need for the user of the HiP application to register on this backend service, which is not acceptable. This is why the authentication of the user can be pushed from the HiP backend to the storage service (i.e., AnnotateIt) with a **JWT!** (JWT!). This JWT! is a digital claim encoded as a JSON object that is digitally signed using **JWS!** (JWS!). It offers a possibility to represent claims that can be transferred between two parties in a safe way. More details about this authentication process will be shown in chapter 4.

After we have now seen the general architecture and the used frameworks, we will take a look at the different part of the system in more detail.

3.5 BACKEND (WEB-SERVER)

The most important part of the system for this thesis is contained within the backend-web-server. The backend should contain the whole data handling and assessment. The students should be able to add data to the system (e.g., a textual article, graphics, AR-data, etc.) and to modify existing data via a Content Management System (CMS). These entries get reviewed, for example by the course supervisor, and unlocked for the frontend application by an user, who represents the master role. To do this, the backend needs features like annotations and highlighting, which should be private for a specific user. By using this, the supervisor can evaluate the given texts right within the CMS and give his final judgement. If the supervisor is not satisfied with the quality of the given text, he should be able to send the document back to the student, to get a revised and updated version of the document. If the supervisor is satisfied, he can unlock the information for showing in the frontend application. The whole process is also shown in Figure 14, which can be found in the appendix. The figure shows the complete workflow within the backend system as flowchart.

The data (i.e., topics, graphics, etc.) should be stored in a way that it can be shown within an AR-environment in the smartphone application. Of course, we will need some mechanism to structure the data, for example tags or stored categories. This kind of information (especially tags) are also very important for the described filtering techniques on the client side.

Furthermore, the backend should include a way to modify the point-clouds of the objects that has been scanned with the smartphone application. It will need features to add annotations directly to these point-clouds to show them

Figure 7: A mockup showing the augmentation editor that will be included in the web-application. The editor will be used to edit the point-clouds, which have been added with the help of the smartphone-application

afterwards within the AR-environment. This editor will be created on the basis of HTML5 and WebGL. A mockup of this site is shown in Figure **??**. These annotations should also be assessable and (un-)lockable for the supervisor.

3.5.1 Input data/content via CMS in the system

The storage of the data respectively content of the topics and groups will be mainly done with using the MongoDB. For example, a topic will be an object with information like version number, content, etc. So, when an user changes something at a topic, the data is changed on the model in the view and send back to the model in the backend. This can be done quite easily with AngularJS because it provides a HTTP module that can send HTTP requests with a simple call like `http.post(destination, data)` to update the model with values from the view-model. Furthermore, the status of the topics should be reflected by

constraints (e.g., maximal or minimal characters in the topic) about the topic that get automatically evaluated while the students are working on the topic.

However, to handle the organization of these topics, we will need to implement groups and their relation to users in the system. So, we will have group documents in the MongoDB, which are linked to the user objects that get returned by the *SecureSocial* 2 plugin, which has been explained in section 2.4.1.

technical details

3.5.2 Manage content as a reviewer

The reviewer should be able to review the content that gets send in by the students. In this process he should be able to attach comments to the topic itself and send feedback to the students. After that, the supervisor can mark the topic as 'ready for publish' if he is satisfied with the given quality.

technical details

3.5.3 Including a 3D-Tooling system for point-clouds (WebGL)

The augmentation editor will be needed to edit the objects (i.e., point clouds) that have been scanned with a smartphone by a student. The editor should be able to show the stored point-clouds and attach objets (for example images) to it. These images can, for example, contain annotations and/or further information. Figure 7 shows a mockup of such a system.

mehr

technical details

3.5.4 Cost estimation of the backend

Now, as we have seen the major ideas about the backend, we will try to create a rough cost-estimation about the backend. The story points have been explained in section 2.2.2 and should represent 6 hours per point. This way we get a rough estimation, which should, however be sufficient to get an idea about the needed time. However, note that the parts of the system could be subject for change because we are using an agile approach and some new requirements may come up within the development process.

Table 3 shows that the Backend system needs a lot of typical CMS functions, which sum up to 35 story points. So, the Backend system should cost about 210 hours to create a prototypical but running system.

Table 3: A brief cost-estimation about the backend

Name	Description	Story points
Create login system	Login needed for right/role management	2
Create language system	The system should be able to handle multiple languages	2
Create chat system	Chats are needed for (group-) communication between users of the system	1
Create message system	Messages are needed for direct communication between users of the system	1
Create group system	Groups will be needed for organization of students	2
Create topic system	Topics will be the main objects, which can be changed by students	10
Create constraint system	Constrains should be created by supervisors and are automatically evaluated	4
Create annotation system	Content of topics should be able to be annotated by students (e.g, highlighted)	3
Create AR - editing system	The 3D objects should be editable by the students	4
Add tooltips	The main features should be explained with tooltips	1
Media upload	The user should be able to upload multiple media formats like pictures, 3D objects, etc.	1
Create version system	One should be able to restore/compare different versions of a topic	4
		$\sum$ 35

3.6 FRONTEND (APP)

The smartphone application is the part of the system that gets shipped to the end-user (respectively downloaded via an App-Store like Google-Play). The user can use the app to find interesting places respectively objects in Paderborn and is able to start a navigation to the place/object easily. Furthermore, the user can get an overview about all places in Paderborn by activating a map that shows all entries within the system. A mockup of this view is shown in Figure 8. Of course, the user will be able to set up specific filters like 'show only art', 'show only historic buildings' or 'use simplified language' to adapt the system to his own experiences and educational qualifications. Moreover, if the university courses would add information over years, the system will need filtering features like this to handle the complexity of the data.

After an user has reached an interesting place, he can use the details tab to switch into the AR-mode. With this view, the user can use the smartphone-camera to embed information, which has been added via the backend, right into the picture of the object. An mockup of this view is shown in Figure 9.

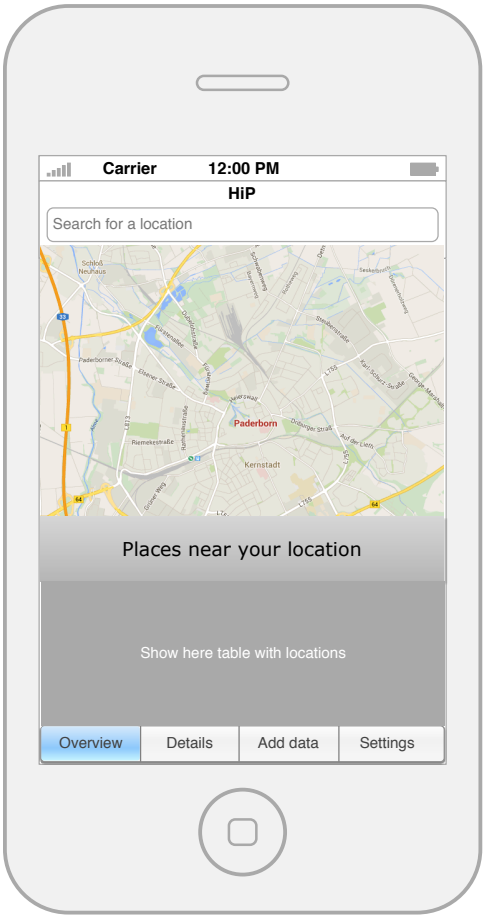

Figure 8: A mockup showing the main page of the frontend application showing a map of paderborn and a general overview about the UI-elements

To create a feasible input for the AR system, the user should be able to scan objects in 3D directly with his smartphone and send the data (i.e., a point-cloud of the scanned object), back to the web-server. Afterwards, the user can add annotations to the point-cloud via the web-backend of the system.

In the following we will focus on the planning and cost estimations off the frontend applications because our actual prototypical implementation contains only the backend system. However, code examples (for Android) and ideas will be included.

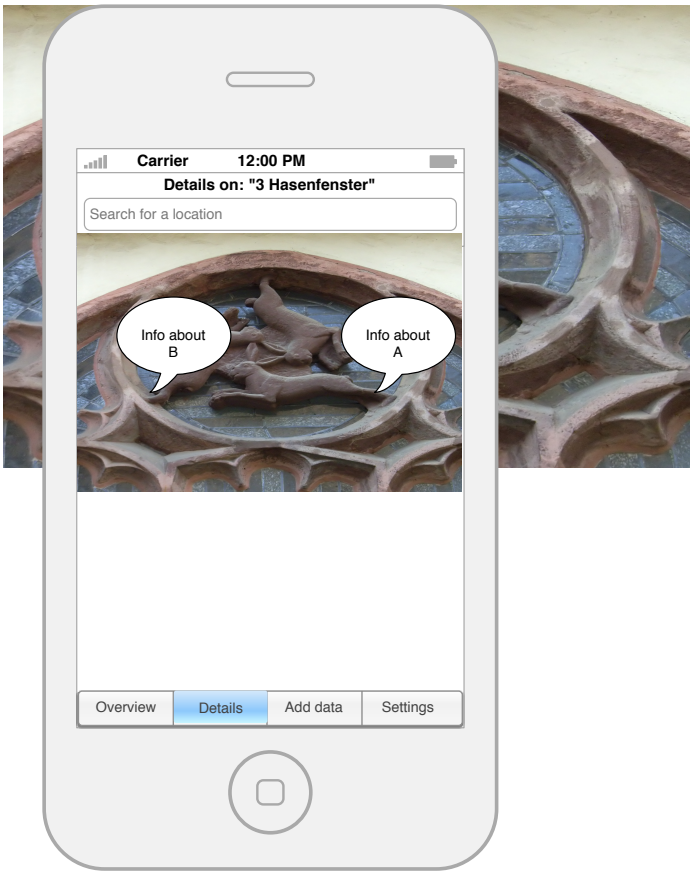

Figure 9: A mockup showing the details page of the "Dreihasenfenster" while the camera of the smartphone is pointing to the window itself

3.6.1 Input data into the system (scan objects and annotate them)

With the Junaio eco-system of the company Metaio it is quite easy to scan objects and store them as point clouds that can be used for further use. more

Data format for AR files

3.6.2 Show close "interesting places" within a map

Because the students should add **GPS!** (GPS!) coordinates for exhibits and locations that are included in the topics, we can make use of these informations to show them on a map on the smartphone.

Listing 3.1: Example for using the GPS coordinates within an Android application

```
1  import com.google.android.gms.maps.*;
2  import com.google.android.gms.maps.model.*;
3  import android.app.Activity;
4  import android.os.Bundle;
5
6  public class MapPane extends Activity implements
       OnMapReadyCallback {
7
8      @Override
9      protected void onCreate(Bundle savedInstanceState) {
10         /* some initialization data */
11     }
12
13     @Override
14     public void onMapReady(GoogleMap map) {
15         LatLng paderbornDom = new LatLng(51.718889, 8.755278);
16
17         map.setMyLocationEnabled(true);
18         map.moveCamera(CameraUpdateFactory.newLatLngZoom(
               paderbornDom, 13));
19
20         map.addMarker(new MarkerOptions()
21                 .title("Paderborner Dom")
22                 .snippet("The cathedral of the Catholic
                       Archdiocese of Paderborn")
23                 .position(paderbornDom));
24     }
25 }
```

The usage of the **GPS!** information is quite easy within Android because we can make use of the *Google Maps API v2*. As Listing 3.1 shows we can directly manipulate the Google Maps view by using the LatLng object. Line 18 shows the movement of the camera (i.e., the excerpt of the map) and line 20 the creation of a marker on the map.

3.6.3 Navigation to "interesting places"

An easy way to create a navigation feature would be the usage of the google directions service Google (2014). However, this runs not natively on Android, so we would need a wrapper to access the HTML and JS based source code in our Android application.

Listing 3.2: Example for construction of the DirectionsRequest JSON object that is needed to use the directions service

```
{
  origin: LatLng | String,
  destination: LatLng | String,
  travelMode: TravelMode,
  transitOptions: TransitOptions,
  unitSystem: UnitSystem,
  waypoints[]: DirectionsWaypoint,
  optimizeWaypoints: Boolean,
  provideRouteAlternatives: Boolean,
  avoidHighways: Boolean,
  avoidTolls: Boolean
  region: String
}
```

After doing that, we can access the direction service by posting an JSON object to the google service and using the returned data to render the navigation path in our application. The construction of such a JSON object is shown in Listing 3.2. As we can see, these object include information like the position of the navigation origin, the position of the destination and specific travel options (e.g., using train, car, etc.). After we have send this object, we need to render the returned data. This is, for example, possible by drawing lines on the map using the *google.maps.Polyline* class. This is shown in Listing 3.3.

Listing 3.3: Example for writing a poly line on the google map

```
function initialize() {
  var mapOptions = {
        /* positioning of the map */
  };

  var map = new google.maps.Map(document.getElementById('map-
      canvas'),
      mapOptions);

   /* array with the coordinates */
  var flightPlanCoordinates = [
    new google.maps.LatLng(37.772323, -122.214897),
    [...],
  ];
  var flightPath = new google.maps.Polyline({
    path: flightPlanCoordinates,
    [...] // options for the rendering of the polyline
    strokeColor: '#FF0000'
```

```
18    });
19
20    flightPath.setMap(map);
21  }
22
23  google.maps.event.addDomListener(window, 'load', initialize);
```

As we can see in Listing 3.3, we can directly draw on the map, which is really fast. The listing shows the code in Javascript. The Polyline class can, however, be accessed natively with the Android SDK (Google (2015)). Furthermore, the simplicity of the usage of the Google Maps API is a good foundation for more complex functionality, like the treasure hunt respectively geocaching functionality which has been expressed within the first requirements meeting. One one just need to create an array of LatLng objects from the locations that should be used as waypoints, the remaining work is done from the Google directions service.

Another, more complex but more powerful, approach would be to fork the **OsmAnd!** (**OsmAnd!**) project and use the code as a basis for the HiP-Navigation feature. **OsmAnd!** is a map and navigation application that uses the **OSM!** (**OSM!**) data. The application offers routing, with optical and voice guidance, and (just like google directions service) offers this navigation for car, bike, and pedestrian (OpenSource (2012)). Furthermore, all main functionalities can be used online and offline, which is nice, because we can directly include the needed maps for the area of paderborn. This would reduce the load on the internet connection of the mobile device (i.e., smartphone). Another great feature of **OsmAnd!** is that it is capable of using different map overlays like for touring features. These functions could be modified within the HiP-Application to render different city plans (e.g., from different epochs). Last but not least, **OsmAnd!** supports intermediate points on your route, which can be modified for the treasure hunt functionality.

IBeacon support for indoor navigation

But, as we have stated before, forking this project to create the HiP-navigation application results in much more work but we end up with a complete navigation application with every common function.

3.6.4 Fetching topics and PDF export

As soon as an user arrives at a specific location, he is able to gather data about the object and collect the information he wants. On a technical level, this is a simple HTTP GET request to our backend to download the information.

Listing 3.4: Creation and usage of an Android HTTP client

```
1  /* init variables */
2  [...]
3
4  /* create HTTP client */
5  HttpClient httpclient = new DefaultHttpClient();
6
7  HttpGet request = new HttpGet();
8  URI url = new URI("http://yourHipServer/topics/uIDOfATopic");
9  request.setURI(url);
10
11 /* send request and get response*/
12 HttpResponse response = httpclient.execute(request);
13
14 /* read response */
15 in = new BufferedReader(new InputStreamReader(response.
       getEntity().getContent()));
16 String line = in.readLine();
17 textv.append(" First line: " + line);
```

As Listing 3.4 shows, it is only a matter of a couple of lines to create a HTTP client for Android. Line 5 shows the creation of a HttpClient object, which is used in line 12 to send the GET request.

Another important thing is the **PDF!** (PDF!) export functionality of the data, that we have just downloaded from our backend server. Because there is no out-of-the-box way on Android to create PDF! documents, we need to make use of an external library. One choice for such a library is iText. This is a PDF! library for Android that offers features to create, adapt and maintain PDF! document (iText Software Corp (2015)). With this library, it is quite easy to export the downloaded data to a PDF! file because we can directly create chapters and section with content within our PDF! file. For doing this, we can make use of classes like `chapter`, `Paragraph` and similar environments (Vogel (2010)). The creation of a new paragraph could for example look like:

```
myCategory.add(new Paragraph("This is a very important message"));
```

All in all, **PDF!** export can be included easy and fast within the smartphone application.

3.6.5 Rendering AR-data with Junaio

Another very important part of the HiP frontend is the rendering of the AR data. As it has been explained in section 2.4.5, we need to create a static content definition, a HTML5 layer and the application logic. While the application logic can be general for the whole application, we need to generate the static content definition and the HTML5 layer for every topic because the shown objects are changed, as soon as we change a topic.

more

3.6.6 Cost estimation of the frontend

Now, after we have seen the technical details about the different parts of the frontend, we can create our cost estimation. One problem is that the cost estimation of the frontend is quite complex and depends heavily on the choice of the basis for the navigation feature. Thus, we assume ,in the following, that we use the **OsmAnd!** project as the basis for the HiP-navigation features. Like in section 3.5.4, one story point should represent 6 hours.

Table 4: A brief cost-estimation about the frontend

Name	Description	Story points
Create running prototype	We need to set up a running Android application that can be used as a starting point for the HiP frontend	5
Showing locations on map	The exhibits and locations should be shown on the map	3
Download data and PDF export	More information should be downloaded as soon as the user arrives at the position. After that, he is able to export the data to a **PDF!** document	4
Include treasure hunts, etc.	The user should be able to join treasure hunts within Paderborn	2
Include the Junaio framework	The user should be able to render AR-information, as soon as he arrives at a specific position	6
Fork and understand **OsmAnd!**	We need to find the important parts of **OsmAnd!** for the HiP application. To find out, what we need to include and how this is possible	6
Include needed parts of **OsmAnd!**	Include the needed parts of **OsmAnd!** in the HiP application	8
		$\sum$ 34

Table 4 shows the minimal features of the smartphone frontend, which sum up to 34 story points. So, the frontend system should cost about 204 hours to create a prototypical but running system.

4

IMPLEMENTATION DETAILS

This chapter will show details about the implementation. However, note that the application is too complex to be explained in full detail in this chapter. Because of that, we will pick specific parts, which are important within the system and show these parts in this chapter. Furthermore, we will show things that has been changed from the draft because the requirements came up within the implementation phase in the agile process. These changes are mostly features, which are not included in the general list of requirements (tables 5, 6, 7, 8 and 9), because they were added afterwards as a result of discussions within the biweekly meetings with our *customers*.

4.1 GENERAL OVERVIEW OF THE SYSTEM

As it has been explained in the previous chapter, the third tier (i.e., the storage/database tier) of the application is driven by a MongoDB. The connection to the MongoDB is done by the play framework with a single line within the configuration file of the play framework (i.e., application.conf):

```
mongodb.uri = "mongodb://localhost:27017/hip"
```

Note that one can easily connect to a remove database with a similar configuration line:

```
mongodb.uri = "mongodb://user:pass@yourDomain:63859/hip"
```

After Play 2.0 has created the connection to the MongoDB we can use it within the framework by extending a controller with the `MongoController` trait (a trait is similar to an interface in Java). By doing this, we have created a couple of Scala Controllers which form the second tier of the application. The main function of these controllers is the handling of data on the way to the database.

However, as we will see, the second tier includes also functions for creating thumbnails and the creation of **JWT!**.

However, the most application logic is included on the client side within the AngularJS framework. Here, we have four main points were we include the application-logic and **GUI!** parts:

partials: Partials are small HTML snippets that get loaded as seen as the AngularJS router gets a HTTP request to the corresponding HTML page. Afterwards, the fitting partial gets loaded at a predefined position within the DOM tree. Thus, you can see a partial as a part of a view of the system that contains **GUI!** elements and their connections to the controllers. The complete HiP backend consists of 22 different partials.

controllers: The 16 controllers, which have been created for the HiP backend, are getting commands and events from the user as he is browsing on the partials. These controllers are able to fetch data from the MongoDB and respond to the actions of the user. To offer the needed functionality, they make use of the next building block: services.

services: The services encapsulate application logic. They can be created by controllers and are working completely autonomous. For the HiP backend, we are using 4 different services.

directives: Last but not least, we are using directives to encapsulate application logic and **GUI!** data that are used in one component. For example, the MediaGallery directive can be used like a usual HTML tag and contains the complete logic for a media gallery including meta data for pictures, etc. However, we will take a closer look at the used directives in section 4.4.

4.2 SCALA: HIGHLIGHTING AND ANNOTATION WITH ANNOTATORJS

The highlighting of the content of topics itself is handled by the JS library *AnnotatorJS*. Technically, the plugins creates `span` HTML tags with the class `annotator-hl` and adds them to the DOM-tree at the same position were the original text was placed. So, the creation of the highlighting itself is quite simple. However, we wanted to attach the information to a given topic and store it in some way.

AnnotatorJS supports the storing of data on an external storage system that needs to be addressable by a specific REST API. We decided to use the external

storage system **AnnotateIt!** (**AnnotateIt!**), which accepts the requests from the *AnnotatorJS* library and stores the data on their server. However, this created a major problem: The user needed to login into the *AnnotateIt* website to be able to store highlights within the HiP backend. To prevent this problem, *AnnotateIt* accepts remote accounts to use the storage system if they are certified with a **JWT!**. To get a better understanding of this process, look at the following process:

1. Alice registers the HiP backend by AnnotateIt, and receives a *consumer key/secret key* pair
2. Bob (a user the HiP backend) logs into the HiP backend and receives an authentication token in the **JWT!** format, which is a cryptographic combination of some details about this user and the consumer secret of the HiP backend
3. Bob's browser sends requests to *AnnotateIt* to save annotations, including the authentication token as part of the payload
4. *AnnotateIt* can verify the Bob is a real user from HiP, and thus stores his annotation

An example for such an **JWT!** authentication token is shown in Figure 10.

Figure 10: An example for a JWT authentication token.
On the left side: The JWT in Base64 encoding. On the right side: As the decoded token.

As one can see, the tokens contains a header, which specifies the used algorithm, a body, which contains the payload and an authenticator, which is a

HMAC! (HMAC!) based on the **SHA256!** (SHA256!) algorithm. The payload contains mainly the userID and consumer key of the application. So, this token proves that the accept comes from an authenticated user of the HiP backend.

The creation of these tokens is done by the part of the backend that is included into the Play 2.0 framework. The action that generates such a token is shown in Listing 4.1.

Listing 4.1: Generation of JWT tokens within the backend

```
def getToken = UserAwareAction { implicit  request =>
    request.user match {
      case Some(user) => {
        /* prepare secret */
        val sharedSecret = "9043aa09-e8c1-46b3-b570-2
            da27a018ac3" getBytes

        // Create HMAC signer
        val signer = new MACSigner(sharedSecret)

        // get current time
        [...]

        // Prepare JWT with claims set
        val claimsSet = new JWTClaimsSet()
        claimsSet setCustomClaim("consumerKey",   "
            ed83da60ae5e4d159729eef16a207525")
        claimsSet setCustomClaim("userId",  [...]);
        claimsSet setCustomClaim("issuedAt", nowAsISO)
        claimsSet setCustomClaim("ttl", "86400")

        var signedJWT = new SignedJWT(new JWSHeader(
            JWSAlgorithm.HS256), claimsSet)

        // Apply the HMAC
        signedJWT.sign(signer)

        val s = signedJWT.serialize

        Ok(s)
      }
      case _ => {
        Ok(views.html.loginplease())
      }
    }
  }
```

Line 14-18 in Listing 4.1 shows the preparation of the JWT's body and in line 20, we create the concrete JWT with the header and the already created body. The signing, which is used in the authenticator, is done in line 23.

Note, that we do not use a general Action but a *UserAwareAction* (line 1). The UserAwareAction is provided by SecureSocial2 and checks the session of the current user. The user object is afterwards checked in line 4. If the user is not logged in but tried to gain a JWT access token, he is send to the login-please.html page that contains the needed information for logging into the system. If the user is logged into the system, the JWT creation process is started.

After we have now seen the creation of JWT tokens, we will now take a look at something else, which is also mainly handled by an Action within the Play 2.0 framework; the upload process of media files.

4.3 SCALA: PICTURE UPLOAD AND THE CREATION OF THUMBNAILS

The upload of a picture is handled on the client side by dropzone.js, which is a small JS library that offers functionality for the easy upload of files to remote servers. On its core, dropzone.js uses the HTML5 upload mechanism to stay browser independent. After the file has been send, a controller called FileController accepts the connection and parses a `multiFormData` request. Listing 4.2 shows the Scala code of that part of the Action.

Listing 4.2: Snippet of the upload Action of the FileController for uploading pictures

```
1  def upload(topicID: String) = Action(parse.multipartFormData) {
       request =>
2      request.body.file("file") match {
3        case Some(photo) =>
4          [...]
5          val newFile = new File("/tmp/picture/uploaded")
6
7          photo.ref.moveTo(newFile)
8
9          val gridFS = new GridFS(db, "media")
10         val fileToSave = DefaultFileToSave(filename,
             contentType)
11
12         [...]
13
```

```
14          /* write file */
15          gridFS.writeFromInputStream(fileToSave, new
                FileInputStream(newFile))
16          [...]
17
18          /* store meta data of that picture */
19          metaCollection.insert(Json.obj(
20            "uID"    ->  cleanedID,
21            "topic" ->  topicID,
22            "thumbnailID" -> cleanedIDThumb,
23            "kvStore" -> "-1"
24          ))
25
26          Ok("File uploaded")
27        case None => BadRequest("no media file")
28      }
29    }
```

As one can see in line 9 from Listing 4.2 we are using GridFS to store the binary data within the MongoDB. GridFS exceed the **BSON!** document size limit of 16MB because it divides a file into parts and stores each of those parts as a separate document within the MongoDB (MongoDB-Inc. (2015)). The picture gets inserted into the MongoDB in line 15 and the meta data is written in line 19. However, by only using this code, we would need to download the complete full-size picture, even when this would not be needed. Because of that the upload Action creates thumbnails of the uploaded pictures, while the picture gets inserted into the database. The code for creating the thumbnails is shown in Listing 4.3.

Listing 4.3: Snippet of the upload Action of the FileController for creating thumbnails

```
1  [...]
2  val TARGET_W = 64; // width of the thumbnail
3  val TARGET_H = 64; // height of the thumbnail
4
5  val filename = photo.filename
6  val contentType = photo.contentType
7  [...]
8
9  /* create thumbnail */
10 val fileToSaveThumb = DefaultFileToSave("thumb_"+filename,
       contentType)
11 [...]
12
13 /* load image for scaling (needed to derive thumbnail) */
```

```
14 var before = ImageIO.read(newFile)
15
16 /* create scale operation */
17 val wScale = TARGET_W / before.getWidth().asInstanceOf[Double]
18 val hScale = TARGET_H / before.getHeight().asInstanceOf[Double]
19
20 var at = new AffineTransform()
21 at.scale(wScale, hScale)
22 var scaleOp = new AffineTransformOp(at, AffineTransformOp.
       TYPE_BILINEAR)
23
24 /* create object that will contain the scaled image */
25 [...]
26
27 /* use scale operation */
28 scaleOp.filter(before, after)
29
30 /* write image to output stream and to DB afterwards*/
31 ImageIO.write(after,"png", os)
32 val fis = new ByteArrayInputStream(os.toByteArray())
33 gridFS.writeFromInputStream(fileToSaveThumb, fis)
34 [...]
```

Now, we have seen both parts of the upload Action independently. To get a full view of the Action and a better understanding about how both parts work together the complete Action is shown in Figure A.1; however, this figure is placed in the appendix because it would disturb within the continuous text.

After we have now seen a couple of important Actions within the Scala controllers, we will start to look at the code on the client side by taking a look at important directives that are used on the client side.

4.4 IMPORTANT DIRECTIVES USED IN THE SYSTEM

In the following we want to outline two important directives that have been created for the client side. Although the system contains a lot more directives, showing all of them would be to large for this thesis. Both directives consists of three different parts that need to be combined to create the actual directive:

template: The template is a piece of HTML code that is inserted at the position where the directive is used

constructor: The constructor registers the directive at the AngularJS framework and is able to modify the scope of the template

controller: The controller is used within the template to create the needed functionality

For the following two example directives, we will show screenshots to represent the template, show JS code for the constructor and give some brief ideas for the controller because the controllers are too big to be shown in detail within this thesis.

4.4.1 A media gallery with the media-gallery directive

The media gallery directive can easily be used within the HTML code by using the tag shown in Listing 4.4. As the listing shows, we need to insert a couple of parameters for the directive to work correctly. However, the most important attribute is the files attribute. This attribute expects a list of media file objects that are shown within the gallery.

Listing 4.4: The listing shows the usage of the media gallery directive

```
1 <media-gallery files="tc.media"
2                picturetooltip="lc.getTerm('tooltip_img_use')"
3                deletetext="lc.getTerm('tooltip_img_delete')"
4                opentext="lc.getTerm('open_image_meta')"
5                sendmetadata="lc.getTerm('send_metadata')"
6                copyto="tc"
7                currenttype="lc.getTerm('current_type')"
8                updatetype="lc.getTerm('update_type')"
9                languagecontroller="lc"></media-gallery>
```

To get a better idea about how the media gallery looks like, we can see an example in Figure 11. This example shows a media gallery that contains two images. So the list of media file objects that we have send to the files attribute contains these two objects.

Figure 11: An example media gallery with two pictures

Directive definition

Now, we can take a closer look at how the directive is working. The configuration is done within the constructor that is shown in Listing 4.5.

Listing 4.5: The listing shows the initialisation of the media gallery directive

```
1  controllersModule.directive('mediaGallery', function() {
2      return {
3          restrict: 'E',
4          scope: {
5              files: '=files',
6              picturetooltip: '=picturetooltip',
7              deletetext: '=deletetext',
8              copyto: '=copyto',
9              opentext: '=opentext',
10             sendmetadata: '=sendmetadata',
11             currentType: '=currenttype',
12             updatetype: '=updatetype',
13             lc: '=languagecontroller'
14         },
15         templateUrl: '/assets/directives/mediaGallery.html'
16     };
17 });
```

There are three major parts within the return object of this constructor (i.e., the Directive Definition Object (AngularJS (2015a))), the restrict attribute, the scope attribute and the templateURL attribute.

The restrict attribute contains a String as a subset of EACM and restricts the directive to a specific directive declaration style. So it modifies the way the directive is used within the HTML code. These styles are:

E: Element name (default): <my-directive></my-directive>

A: Attribute (default):

C: Class:

M: Comment: <!– directive: my-directive exp –>

The scope attribute creates a new scope. If the attribute contains a new JSON object, then a new scope is created with these values. With other words, the new *isolate* scope does not inherit from the parent scope but uses the given data in the scope attribute. This is useful for the creation of reusable components because such components should not read or modify data in the parent scope, as they are used in different contexts.

The last attribute, the templateUrl specified the location of the the template that is asynchronously loaded, when the directive is used.

Controller of the media gallery directive

The controller is used for the whole data handling that is needed to offer the functionality of the media gallery. For example, the controller contains a function called openMetaData, which is shown in Listing 4.6, to download and prepare the meta data of the given picture. As one can see in that listing, the controller makes heavy use of the keyValueService. This service will be explained in more detail in section 4.5. However, for the moment take it as a provider for typical key/value containers. So, a container is a JS-object that contains a list of keys and values; however the needed keys depends on the *type* of the currently loaded container.

Listing 4.6: The listing shows the openMetaData function

```
1  this.openMetaData = function(uIDOfThePicture,
       uIDOfTheKeyValueStore){
2      if(uIDOfTheKeyValueStore != "-1"){
3          /* load it */
4          keyValueService.getKVStore(uIDOfTheKeyValueStore,
               function(store){
5              /* use store */
6              that.store = store;
7          });
8      }else{
9          /* create it */
10         var store = keyValueService.
               createEmptyStoreAccordingToType('img');
11
12         /* modify picture kvStore */
13         $http.put('/admin/picturekv/'+uIDOfThePicture+'/'+
               store.uID);
14
15         /* use store */
16         that.store = store;
17     }
18
19     /* trigger view */
20     $scope.collapse[uIDOfThePicture] = !$scope.collapse[
           uIDOfThePicture];
21 };
```

The function expects the uID of the picture and the uID of the key / value store and loads resp. creates the store for the type `img`.

Within this section, we have seen, the main parts of a directive in AngularJS and, to underline the theoretical background, an example for such an directive. The files used within the second directive, the template box, are quite similar. But, we will now take a closer look.

4.4.2 Template handling with the templates-box directive

The template box is used for creating new templates, sharing templates and using templates within topics. The UI! elements that are shown are dependent on the user that is currently logged in. For example, a student who takes a look at the template box has no buttons for sharing a specific template.

A typical example of the template box is shown in Figure 12. The figure shows a template box from the view of a supervisor.

Figure 12: An example template box with two templates. The menu of the lower template has been opened by using the triangle on the right side

The *Directive Definition Object* for this directive looks quite similar to the one from the media gallery directive and is shown in Listing 4.7. Because we have seen the theoretical background in the last section, we will explain this directive only briefly. Like in the last case, we restrict the usage to the element name with the `restrict` attribute, create a new isolated scope with the `scope` attribute and set the templateURL of the HTML code that is send back as soon as the directive is used.

It is noticeable within the *Directive Definition Object* shown in Listing 4.7 that it needs a lot more controllers as the *Directive Definition Object* in section 4.4. This results from the fact that the template box has a lot more functionality that needs to communicate with different aspects of the system, like the sharing from templates with groups and users.

Listing 4.7: The listing shows the initialisation of the template box directive

```
1  controllersModule.directive('templatesBox', function() {
2      return {
3          restrict: 'E',
4          scope: {
5              lc: '=languagecontroller',
6              uc: '=usercontroller',
7              gc: '=groupcontroller',
8              showcondition: '=showcondition',
9              tc: '=append',
10             directconnect: '@directconnect'
11         },
12         templateUrl: '/assets/directives/templatesBox.html'
13     };
14 });
```

This increased complexity is also recognizable within the *TemplateController*, which is the controller that drives the template box directive. The controller offers functions for fetching templates, transfer templates from one key/value store to another, transfer keys to groups, etc. These functions remain manageable because we use, again, the key value service to handle most of the data management.

After we have now seen these two directives that make use of the key value service, we will now take a look at the service itself.

4.5 ANGULARJS: KEY/VALUE STORES

The key value service is a small but mighty service that is used at multiple places within the HiP backend. A brief overview about the functionality is shown in Figure 13.

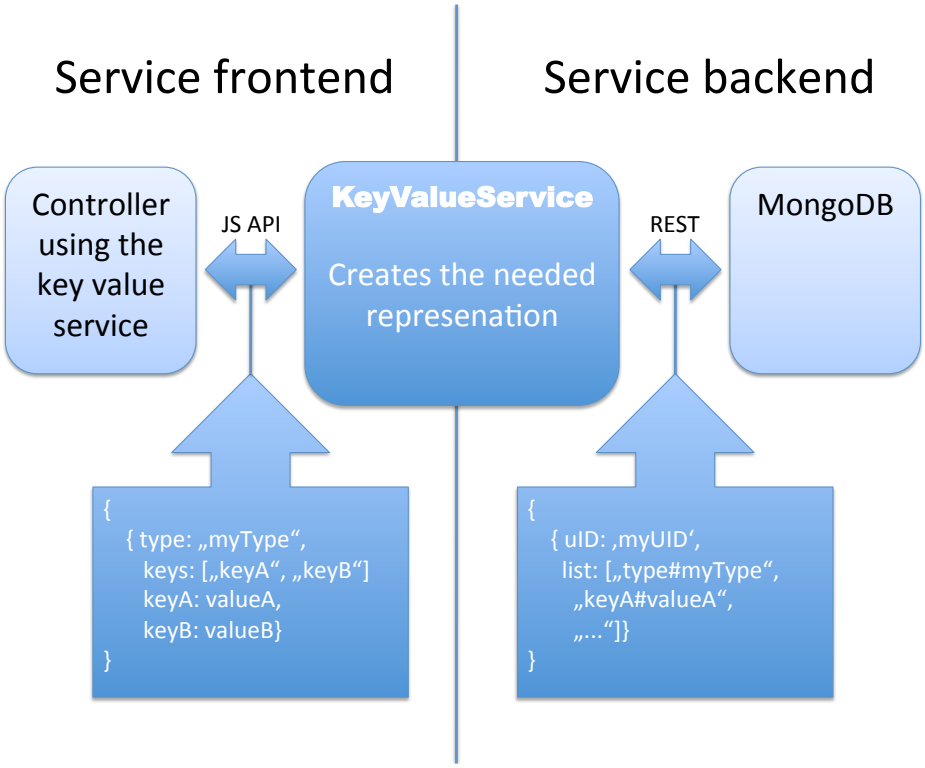

Figure 13: The frontend and backend of the key value service with the two different data formats

As one can see in the figure, the key value service can be accessed via a JS API within AngularJS to load and modify key value stores. These stores can afterwards been pushed to the MongoDB in a small and compact serialized data format. The main idea behind the serialization process is that we do not need any information about the used keys and we can create and use as many keys as we want but the database schema within the MongoDB stays the same. Every added key will end up as one new entry in the list array within the JS object that will be pushed to the MongoDB. This makes the key value service extremely adaptable to every situation. This can also be seen in the fact that the service is used within the HiP backend to store textual templates, meta-data for pictures and tag-clouds. So, it is used in very different situations for very different things but it can be adapted to every of these use cases by using

different keys. But we will get more into details about the keys in section 4.5.1 as we will need the typing to completely understand the topic.

All in all, the key value service offers functions for downloading key value stores from the backend database, delete stores, create new stores (empty and according to specific types), modify stores, transfer keys from one store to another, etc. However, a crucial part of the logic is encapsulated within the function that translates the data format because we need to change the data format after we downloaded a store from the backend. The translation function is shown in Listing 4.8.

Listing 4.8: The translation from the backend data format to the frontend data format within the key value service

```
1  function serializedFormToJSON(uID, listOfKeysAndValues) {
2      var JSON = {};
3      var keys = [];
4      listOfKeysAndValues.forEach(function(item) {
5          var token = item.split("#");
6
7          // add key and value to JSON object
8          if(token[0] != "type"){
9              JSON[token[0]] = token[1];
10
11             // add to key list
12             keys.push(token[0]);
13         }else{
14             JSON.type = token[1];
15         }
16     });
17
18     JSON.uID = uID;
19     JSON.keys = keys;
20     JSON.length = keys.length;
21     return JSON;
22 }
```

The code in that function is straightforward. We iterate over every entry of the list array in line 4, split the key and value in line 5 and store the resulting array in the variable token. After that, we add the key with its value to the variable called JSON, which ends up to be the frontend format representation (line 11-14). If the key has the name type, we copy the value directly to the frontend representation (shown in line 9). At last, we can return the variable JSON, which will be used within the service and by the controller that uses the service.

4.5.1 Typing of stores

One of the features that make the **KVS!** (KVS!) so reusable is the idea of typed key-value stores. A type specified the keys that need to be included within a specific key-value store. Of course, every store that is used within the system has a specific type and every type is a child of another one. If we create a new child that does not need to be derived from a specific type then we say that its parent type is *root*. On the other hand, if we have a type A with the keys key1 and key2 and create a new type B that is a child of type A than B will derive the keys key1 and key2 automatically because it is a subtype of A.

So, for every new situation were we want to use the **KVS!**, we just need to create a new storage type and the remaining work is done by the service. The editing of types (e.g., adding keys or even complete types) can be done from every controller within AngularJS by using the *TypeService*. This service encapsulates all needed functionality to modify types and is used within the HiP backend to create a type modify interface for the admins of the HiP backend. However, we will not get into more details about this service or the type modify interface within this thesis.

After we have now seen some of the details of the implementation, we will now come to the testing chapter.

5

TESTING THE APPLICATION

Although the section called *testing* is the second last of this thesis, testing was a major force within the whole development process. We had to do changes on existing code parts often, which was the main reason for the TDD development approach. So, the following section will contain the results of the final test suites for the current version of the HiP application. The tests have been developed with the help of the Jasmine framework.

5.1 TEST ENVIRONMENT

The Jasmine test suites were run within Karma on Mac OSX operating system with Google Chrome. The hardware configuration was a dual core **CPU!** (**CPU!**) and 8096MB **RAM!** (**RAM!**).

5.2 TESTING RESULTS

5.3 ACCEPTANCE TEST OF THE PROTOTYPE

Besides the technical testing of the application, we created an acceptance test for the smartphone-frontend-application and the web-backend-application.

5.3.1 Small usablity study of the app

5.3.2 Small usability study of the backend / CMS

6

DISCUSSION AND FUTURE WORK

6.1 HANDING OF THE PROJECT

– PG -> DevOps , Scrum ....

6.2 ARISEN PROBLEMS WITHIN THIS THESIS

Since this master thesis included two different aspects (i.e., the agile process and the actual development of the backend) of the actual development process, we want to express the problems separately.

6.2.1 The agile process

To develop an application with an agile process was a great experience. It was really nice to get a biweekly feedback about the functions and more ideas about the further development process. This is especially important because if one creates an application for a foreign field (which was partially the case in this development process - e.g., the needed keys for meta-data of pictures -) one just do not know what important points are. But this information can easily gathered within the meetings to create an application that can actually be used by the customer. However, we found one drawback of these meetings, which was the problem that a lot of new requirements and feature request was created within these meetings, which makes time-planning really hard. For example, in two meetings created that much new requirements that we needed the whole two weeks to implement the newly requested features. Thus, we did not reduce the amount of story points within the actual backlog. This

could result in problems for real industrial projects because some features are needed two specific deadlines.

After we now have taken a look at the agile process, we will now take a look at the development of the application.

6.2.2 The development of the application

On the other side, the development of the HiP-application was fine and AngularJS was definitely a great choice as a basis framework. However, sometimes the usage of AngularJS enforces the need of little work-arounds. For example, the simple call to the Google Maps API v3 for rendering a map resulted in a map that contained grey little boxes and the whole map needed to be resized again (Although, we decided to use HereMaps instead of Google Maps anyway). Nevertheless, such bugs were rare and, thus, AngularJS worked great.

Another problem was the complexity of the application. Especially in the last few weeks of the development process the complexity resulted in a lot more development effort as soon as code 'at the core of the system' needed to be changed because of new requirements that came up in the biweekly meetings. Thus, we cannot directly confirm the observation of Kent Beck that the curve that represents the cost of changes within an agile process is more or less flat Beck (2003). However, even changes at code parts with a lot of dependencies were quite good possible because the whole development was done with the idea of changing requirements in mind.

6.3 DISCUSSION AND FUTURE WORK

This last section of this master thesis will draw a conclusion and take a loot at possible future work.

6.3.1 Results / Conclusion

We think that we have created a great system within this master thesis, which is a good foundation for following project groups and/or bachelor and master thesis's. We started with the idea in mind that we would need a system that works on the one hand as a working environment for students and, on the

other hand, as a CMS backend for a smartphone application that presents the data that has been created by the students. more

6.3.2 Future work

Although we have implemented a lot of functions within the backend, there is a lot more functionality within our Scrum backlog. In general, future work can be done on two major area, the smartphone application (which gets at the moment developed by a bachelor Student) and has been sketched within this thesis. And the remaining functions of the backend. The smartphone application will need sophisticated 3D-Rendering functionality as well as navigation functions. The backend, on the other side, needs more functions for editing the point clouds that have been imported by the students. Furthermore, the backend should be configured in a way to support continuos delivery and the described patterns for DevOps architectures (Cukier (2013)) could be used within the system.

A

APPENDIX

The appendix contains some diagrams and tables that were to big to put them into the continuous text. Furthermore, it contains a small installation manual for the HiP backend.

A.1 INSTALLATION MANUAL

The installation process of the HiP backend is quite simple. In general you need to install 3 different things: The Play 2.0 framework, the MongoDB and **NPM!** (NPM!). In more detail the process looks like this:

1. Get your Play 2.0 framework instance from https://www.playframework.com/download
2. Install Play 2.0 (do NOT run it at this point in time)
3. Go to your HiP root folder
4. Switch to %hipRoot/public
5. Run npm link on your command line
6. Download MongoDB from http://www.mongodb.org/downloads
7. Run the MongoDB with the provided default database (this can be done on a Windows PC with the following command:
   `mongod --dbpath $<<$PATH TO THE PROVIDED DB$>>$)`
8. Again: Go to your HiP root folder
9. Start the HiP backend by using the command: activator run

Your HiP server is now running. After you have installed all the needed components you can easily run the test cases for the controllers with the following commands:

1. Switch to %hipRoot/public
2. Start the HiP test cases by using the command: karma start karma.conf

After we have now seen how we can start the application and the provided test cases, the remaining part of the appendix will contain bigger figures and tables.

A.2 FIGURES AND TABLES

Figure 14 shows the work-flow within the backend system as a flow diagram.

Furthermore, Figure 15 shows an **UML2!** use case diagram showing the different users (i.e., user roles) of the application.

The IDs of the list, which are written in bold letters, will be done within the master-thesis itself. The remaining parts can be done via, for example, project-groups, etc.

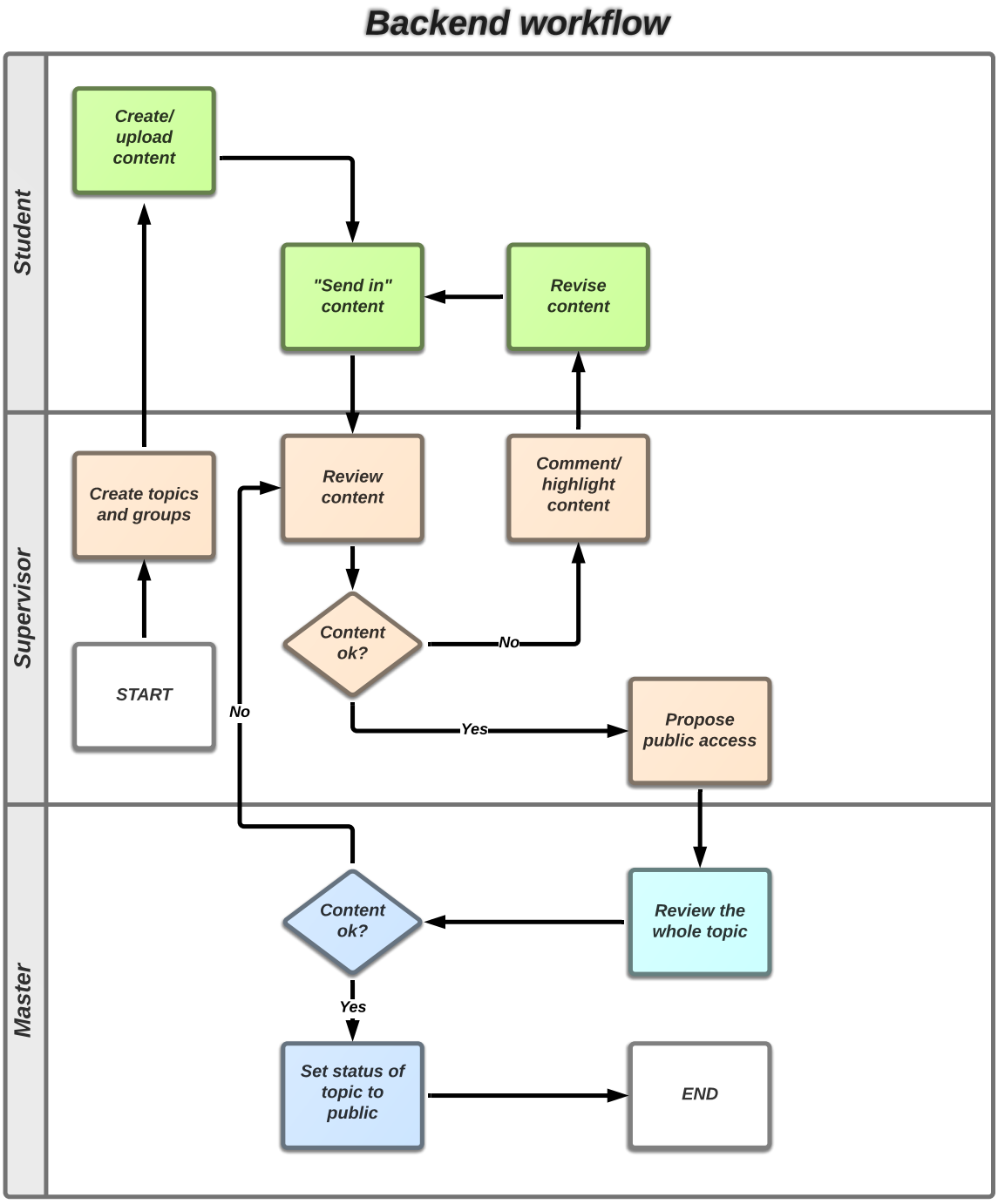

Figure 14: The diagram shows the work-flow within the backend system with the three roles that are involved in the work-flow

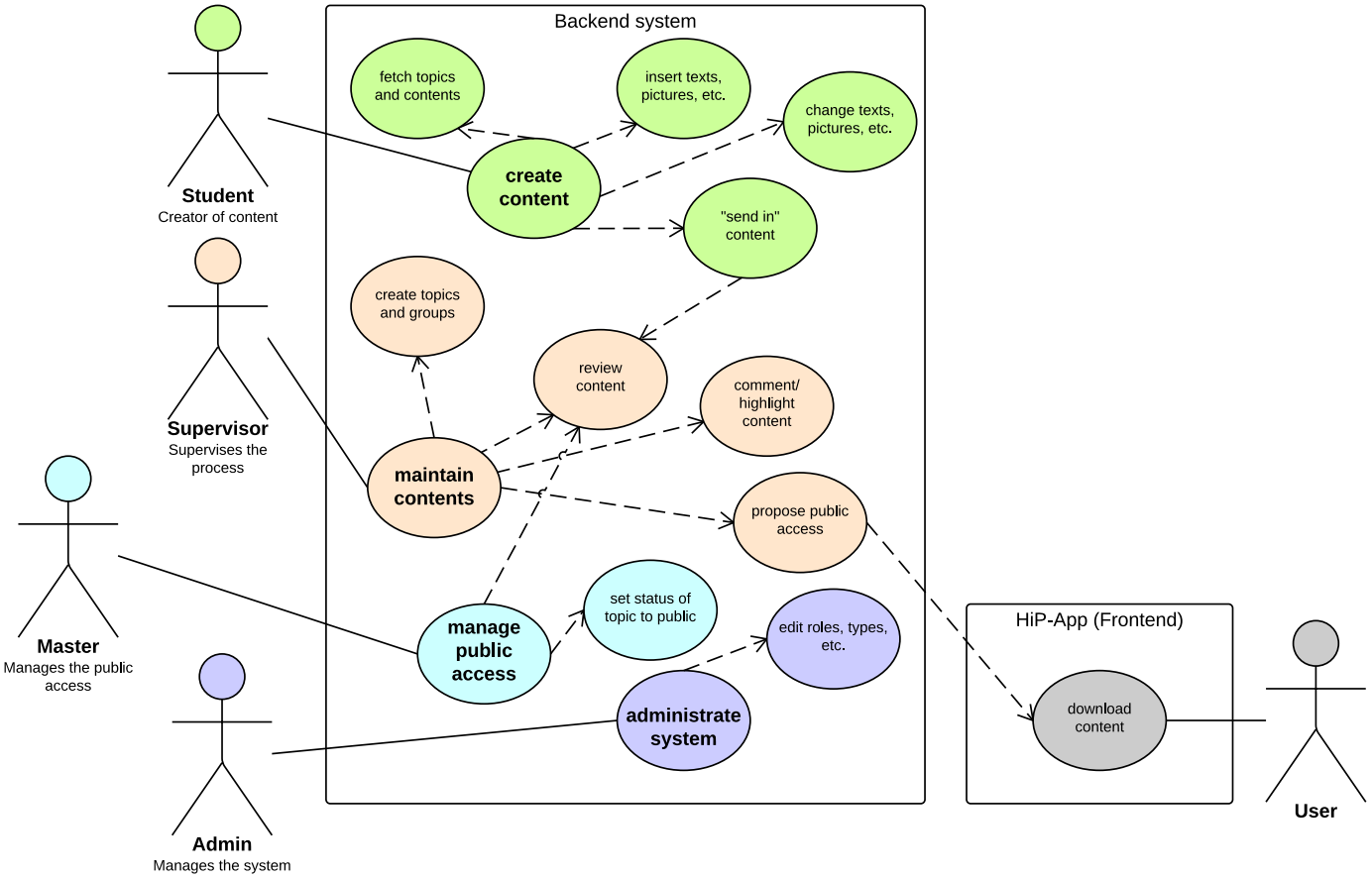

Figure 15: An UML2! use case diagram showing the different users of the HiP system

Table 5: Showing the derived requirements of the backend for the supervisor role, which are sorted by priority

ID	Description	Acceptance criteria	Priority
BS1	The supervisor should draft **guidelines** and assistance (e.g., Button with question-mark)	- At least one information resp. help function per functionality	1
BS2	The supervisor should be able to see **which data is missing**	- The feedback should be visual - The feedback should be transparent to upper layers of the UI	1
BS3	The supervisor should be able to see **data that is ready** for review	- Try without content that is ready for review - Try with content that is ready for review	1
BS4	The supervisor should be able to **assign exhibits** to students	- Try assigning an exhibit to one student - Try assigning an exhibit to more than one student	1
BS5	The supervisor should be able to **trace content** back to specific students	- Show visual connection between student and content	1
BS6	The supervisor should be able to **define topics** and exhibits	- Try defining a topic more than once	1
BS7	The supervisor should be able to **comment and discuss** the given content of the students	- Try commenting empty content - Try commenting a lot of content	1

Continued on next page

Table 5 – continued from previous page

ID	Description	Acceptance criteria	Priority
BS8	The supervisor should be able to **mark errors** in the content	- Try marking an error twice	1
BS9	The supervisor should get **e-mail notifications** about new content handed in by students	- The message should leave the system in less than 2 minutes in 90% of the time	1
BS10	The supervisor should be able to **copy topics** and categories (e.g., usage of templates for different typical cases, duplication, etc.)	- Try copy an empty topic - The copied topic should be easily changeable to adapt it to the new usage	2
BS11	The supervisor should be able to define **validation-constraints** (e.g., character limitation)	- Try with error within the validation constraints	2
BS12	The supervisor is able to see the **amount** of **texts and pictures** in a hidden topic	- Try without any content included - Try with a lot of content included	2
BS13	The supervisor should be able to **work offline**	- Try disconnecting a running session	3

Table 6: Showing the derived requirements of the Backend for the student role, which are sorted by priority

ID	Description	Acceptance criteria	Priority
BSt1	The students are only able to **send in specific content** (field / topic)	- Try sending content to another topic	1
BSt2	The students should get an **e-mail notifiation** about new content in their topic (e.g., send in via fellow students)	- The e-mail should be received in less than 2 minutes in 90% of the time	1
BSt3	The students should be able to send in **metadata**	- Try with errors within the meta-data	1
BSt4	The students should be able to **overview** the **possible links** within their topic (e.g., GPS-information)	- Try without any links - Try with a lot of links	1
BSt5	The students should be able to **send in content**	- Try sending empty content - Try sending a lot of content	1
BSt6	The students should be able to **propose** topics and content	- Try proposing an existing topic	1
BSt7	The students should only have **access** to the backend **for a specific time**	- Try logging in after the temporary account has been deleted	1
BSt8	The students should have **access to all** temporary content (i.e., not reviewed content)	- Try accessing currently empty content	1

Continued on next page

Table 6 – continued from previous page

ID	Description	Acceptance criteria	Priority
BSt9	The students should be able to create **interdisciplinary groups** and communicate within these	- Try creating a group without users - Try to send an empty message to the group - Try to send a very long message to the group	1
BSt10	The students should be able to see their content in a **preview mode** that simulates the frontend	- Try showing an empty topic - Try showing a huge topic	2
BSt11	The students should be able to see content of **other groups in a preview** mode that simulates the frontend	- Try showing an empty topic - Try showing a huge topic	2
BSt12	The students should be able to comment and **discuss** the content of their group or other groups	- Try to send an empty comment - Try to send a huge comment	2
BSt13	The students should be able to **hide** their unfinished work to the supervisor	- Try hiding without having any content	2

Table 7: Showing the derived requirements of the Backend for the master role, which are sorted by priority

ID	Description	Acceptance criteria	Priority
BM1	The master should be able to **recover data** by using a back-up system	- The recovery should not take longer than one hour	1
BM2	The master role can be assigned to a **couple of users** at the same time	- Try to assign the master role to nobody	2
BM3	The master is able to do the final **acceptance**	- Try to accept an empty topic - Try to accept a huge topic	2

Table 8: Showing the derived requirements of the Backend, which are sorted by priority

ID	Description	Acceptance criteria	Priority
BMi1	The data of the system is **stored** on IMT-Server	- The data should be easily transferable	1
BMi2	The system can be **updated** and maintained in the future (e.g., project-groups, SHK, etc.)		1
BMi3	The content should not be limited to specific **layouts, views** (e.g., languages) and templates		1
BMi4	The system should be **expandable** (e.g., new content, filters, etc.)		1
BMi5	The system should be **safe** with respect to hackers resp. data manipulation	-The system should be safe with respect to the economic view/ definition of safety	1
BMi6	The system offers features to **manage** groups	- Try managing a group with an empty name	2

Table 9: Showing the derived requirements of the Frontend, which are sorted by priority

ID	Description	Acceptance criteria	Priority
F1	The user should be able to **navigate** to the different locations shown in the HiP-application	- The navigation should response fast - Try navigating to the current position	1
F1.A	The user should be able to navigate to the different locations and discover these locations on his own	See F1	1
F1.B	The user should be able to navigate to the different locations and use round tour information of the application	See F1	1
F1.B	The user should be able to navigate to the different locations while using filters (e.g., epochs)	See F1	1
F2	The user should be able to create **thematic routes**	- Try creating a route without assigning a theme	1
F3	The user should get a **list of locations/exhibits** in Paderborn	- Try opening an empty list	1
F4	The user should **see linkings** within an exhibit different exhibits (e.g., Liborischrein -> Hle -> Scriptorium)	- Try opening a topic without links - Try opening a topic with a lot of links	1

Continued on next page

Table 9 – continued from previous page

ID	Description	Acceptance criteria	Priority
F5	The user should be able to **deselect** specific categories	- Try deselect only one - Try deselect many	1
F6	The user should be able to **filter** exhibits on the map (e.g., locations, historical figures, etc.)	- Try using multiple filters	1
F7	The user is able to **overlay** the current map of the city with historical maps	- Try overlay one map with a hist. one - Try overlay a couple of maps	1
F8	The user is able to **see himself and historical** places on the map	- Try in an area without hist. places - Try in an area with a lot of hist. places	1
F9	The user should not **exceed his storage** on the smartphone	- Clear cache should be possible	1
F10	The user should not **exceed his data-volume** on the smartphone	- Pictures and videos have to be small	1
F11	The user should be able to use the application easily (**good usability**)	- Interface should not include too many functions per view	1
F12	The user should be able to switch between **different contents** (e.g., Video, 3D, etc.) fast	- At most two clicks/touches between the different contents	1
F13	The user should be able to see ***invisible***	- Try with more than one invisible	1

Continued on next page

Table 9 – continued from previous page

ID	Description	Acceptance criteria	Priority
	objects within the details-tab (e.g., something placed inside an altar)	object at the same time	
F14	The user should be able to use **tablets** and smartphones	- The UI should adapt to the screen size resp. resolution	1
F15	The user should only get details about an exhibit while he is **next to it or afterwards**	- Try to get details beforehand	1
F16	The user should be able to get **texts, graphics/ pictures** and links about an exhibit	- Try without any texts, etc. - Try with a lot of texts, etc.	1
F17	The user should be able to get **audio, video** and **3D-views/models** about an exhibit	- Try without any videos, etc. - Try with a lot of videos, etc.	2
F18	The user can create and join **treasure hunts** respectively geo-caching features	- Try join an treasure hunt without a name	2
F19	The user should get informed about **exhibits** and **locations that are next to him**	- The information should be send immediately as the user arrives at the position	2
F20	The user should be able to get navigated with **AR-rabbits**	See F1	2
F21	The user should be able to get navigated inside of a **building**	- The navigation should be accurate	2
F22	The user should be able to choose between		2

Continued on next page

Table 9 – continued from previous page

ID	Description	Acceptance criteria	Priority
	different **starting possibilities** (i.e., tour, discovery and historical topics)		
F23	The user should be able to **hear the content** via an audio-guide	- The audio files should be small (see, F9, F10)	2
F24	The user should be able to get exhibits as **comparison by using AR**	- Try opening more than one exhibit as comparison	2
F25	The user should be able to **create own notes and comments**	- Try creating an empty note/comment - Try creating a huge note/comment	2
F26	The user should be able to **share content** via social media	- Sharing should not need more than two clicks	2
F27	The user should be able to **export content** as PDF and create book-marks	- The export should not take longer than 30 sec in 90% of the time	2
F28	The user should be able to get the content in **different languages** (i.e., english, french, turkish)	- Adding new languages should be easy	2
F29	The user should be able to choose between different **criteria** with respect to the audience (e.g., different ages of people)	- Try selecting one criterion - Try selecting more than one criterion	2

Listing A.1: Complete upload Action of the FileController

```
1  def upload(topicID: String) = Action(parse.multipartFormData) {
       request =>
2      request.body.file("file") match {
3        case Some(photo) =>
4          val TARGET_W = 64; // width of the thumbnail
5          val TARGET_H = 64; // height of the thumbnail
6
7          val filename = photo.filename
8          val contentType = photo.contentType
9
10         val newFile = new File("/tmp/picture/uploaded")
11
12         if (newFile.exists())
13           newFile.delete()
14
15         photo.ref.moveTo(newFile)
16
17         val gridFS = new GridFS(db, "media")
18         val fileToSave = DefaultFileToSave(filename,
               contentType)
19
20         /* create thumbnail */
21         val fileToSaveThumb = DefaultFileToSave("thumb_"+
               filename, contentType)
22
23         val os = new ByteArrayOutputStream()
24
25         /* load image for scaling operation (needed to derive
               thumbnail) */
26         var before = ImageIO.read(newFile)
27
28         /* create scale operation */
29         val wScale  = TARGET_W / before.getWidth().asInstanceOf
               [Double]
30         val hScale  = TARGET_H / before.getHeight().
               asInstanceOf[Double]
31
32         var at = new AffineTransform()
33         at.scale(wScale, hScale)
34         var scaleOp = new AffineTransformOp(at,
               AffineTransformOp.TYPE_BILINEAR)
35
36         /* create object that will contain the scaled image */
37         val w = (before.getWidth() * wScale).asInstanceOf[Int]
38         val h = (before.getHeight() * hScale).asInstanceOf[Int]
39         var after = new BufferedImage(w, h, BufferedImage.
               TYPE_INT_ARGB)
40
```

```
41          /* use scale operation */
42          scaleOp.filter(before, after)
43
44          /* write image to output stream */
45          ImageIO.write(after,"png", os)
46
47          /* derive FileInputStream */
48          val fis = new ByteArrayInputStream(os.toByteArray())
49
50          /* write both files */
51          gridFS.writeFromInputStream(fileToSave, new
                FileInputStream(newFile))
52          gridFS.writeFromInputStream(fileToSaveThumb, fis)
53
54          /* include the additional data */
55          val cleanedID = fileToSave.id.toString.split('"')(1)
56          val cleanedIDThumb = fileToSaveThumb.id.toString.split(
                '"')(1)
57
58          metaCollection.insert(Json.obj(
59            "uID"    ->  cleanedID,
60            "topic" ->  topicID,
61            "thumbnailID" -> cleanedIDThumb,
62            "kvStore" -> "-1"
63          ))
64
65          Ok("File uploaded")
66        case None => BadRequest("no media file")
67      }
68    }
```

BIBLIOGRAPHY

Alliance, A. (2015). Guide to agile practices. Website, http://guide.agilealliance.org/guide/tdd.html, 14.02.2015.

AngularJS (2015a). Comprehensive directive api. Website, https://docs.angularjs.org/api/ng/service/$compile#directive-definition-object, last checked 18.02.2015.

AngularJS (2015b). karma-runner/karma. Website, https://github.com/karma-runner/karma, 14.02.2015.

Azuma, R. T. (1997). A survey of augmented reality a survey of augmented reality a survey of augmented reality. *In Presence: Teleoperators and Virtual Environments*, 6(4):355–385.

Beck, K. (1999). Embracing change with extreme programming. *Computer*, 32(10):70–77.

Beck, K. (2003). *Test-driven development: by example*. Addison-Wesley Professional.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., and Thomas, D. (2001). Manifesto for agile software development. Website, http://www.agilemanifesto.org/, 14.02.2015.

Berczuk, S. (2007). Back to basics: The role of agile principles in success with an distributed scrum team. *Agile Conference (AGILE), 2007*, pages 382 – 388.

Bondo, J., Barnard, D., and Burcow, D. (2010). *iPhone User Interface Design Projects*. Apress.

Bradley, J., Sakimura, N., and Jones, M. (2014). Json web token (jwt).

Ceschi, M., Sillitti, A., Succi, G., and De Panfilis, S. (2005). Project management in plan-based and agile companies. *Software, IEEE*, 22(3):21–27.

Cohn, M. (2004). *User stories applied*. Addison-Wesley Professional.

Cukier, D. (2013). Devops patterns to scale web applications using cloud services. In *Proceedings of the 2013 companion publication for conference on Systems, programming, & applications: software for humanity*, pages 143–152. ACM.

Danube (2014). Scrum effort estimation and story points. Website, http://scrummethodology.com/scrum-effort-estimation-and-story-points/, 12.02.2015.

Dev.metaio.com (2015). Overview | metaio developer portal.

Duvall, P. (2012). Breaking down barriers and reducing cycle times with devops and continuous delivery.

Eckerson, W. W. (1995). Three tier client/server architecture: Achieving scalability, performance, and efficiency in client server applications. *Open Information Systems*, 3(20):46–50.

Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, USA. AAI9980887.

George, B. and Williams, L. (2004). A structured experiment of test-driven development. *Information and Software Technology*, 46(5):337 – 342. Special Issue on Software Engineering, Applications, Practices and Tools from the {ACM} Symposium on Applied Computing 2003.

Google (2014). Google maps javascript api version 3. Website, https://developers.google.com/maps/documentation/javascript/directions, 14.02.2015.

Google (2015). Polyline. Website, https://developer.android.com/reference/com/google/android/gms/maps/model/Polyline.html, 14.02.2015.

Hampel, T. (2001). *Virtuelle Wissensräume*. PhD thesis, Universität Paderborn.

IEEE (1990). Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84.

iText Software Corp (2015). itext core functionality. Website, http://www.itextpdf.com/functionality, 14.02.2015.

Janzen, D. S. and Saiedian, H. (2005). Test-driven development: Concepts, taxonomy, and future direction. *Computer Science and Software Engineering*, page 33.

Jones, C. (2003). Why flawed software projects are not cancelled in time. *Cutter IT Journal*, 16(12):12–17.

Junaio (2014). Why to become a junaio developer. Slideshare presentation, http://www.slideshare.net/metaio_AR/why-to-become-a-junaio-developer, last checked 14.09.2014.

Keaveney, S. Conboy, K. (2011). *Cost Estimation in Agile Development Projects*. National University of Ireland.

Mast, R. (2013). Architektur in agilen teams: Was softwarearchitekten von jazzmusikern lernen können. *OBJEKTspektrum*.

Maximilien, E. Michael. Williams, L. (2003). Assessing test-driven development at ibm. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 564–569. IEEE.

Maxxor (2015). Software development process | maxxor. Website, https://www.maxxor.com/software-development-process, last checked 14.02.2015.

Mesbah, A. and van Deursen, A. (2007). Migrating multi-page web applications to single-page ajax interfaces. *Software Maintenance and Reengineering, 2007. CSMR '07.*, pages 181 – 190.

MongoDB-Inc. (2015). Gridfs. Website, http://docs.mongodb.org/manual/core/gridfs/, last checked 16.02.2015.

Mueller, E., Wickett, J., and Gaekwad, K. (2011). What is devops? Website, http://theagileadmin.com/what-is-devops/, last checked 21.02.2015.

Nerur, S., Mahapatra, R., and Mangalaraj, G. (2005). Challenges of migrating to agile methodologies. *Communications of the ACM*, 48(5):72–78.

North, D. (2012). Bdd is like tdd if... Website, http://dannorth.net/2012/05/31/bdd-is-like-tdd-if/, 14.02.2015.

OpenSource (2012). Osmand. Website, https://code.google.com/p/osmand/, 14.02.2015.

Paulk, M. C. (2002). Agile methodologies and process discipline. *Institute for Software Research*, page 3.

Rodriguez, A. (2008). Restful web services: The basics. *Online article in IBM DeveloperWorks Technical Library*, 36.

Schwaber, K. and Beedle, M. (2002). *Agile Software Development with Scrum*. Pearson.

Statista (2014). Global apple iphone sales from 3rd quarter 2007 to 3rd quarter 2014 (in million units). Website, http://www.statista.com/statistics/263401/global-apple-iphone-sales-since-3rd-quarter-2007/, last checked 13.09.2014.

Sutherland, J. Jakobsen, C. (2009). Scrum and cmmi – going from good to great. *Agile Conference*.

Trelle, T. (2014). Mongodb. *JavaMagazine*, 5:56–62.

Vogel, L. (2010). Creating pdf with java and itext. Website, http://www.vogella.com/tutorials/JavaPDF/article.html, 14.02.2015.

Wikimedia-Foundation (2014). About us. Website, https://www.wikimedia.de/wiki/%C3%9Cber_uns, last checked 12.09.2014.

Wikitude (2014). Augmented reality for multiple platforms. Website, http://www.wikitude.com/products/wikitude-sdk/, last checked 14.09.2014.

Williams, L., Maximilien, E. M., and Vouk, M. (2003). Test-driven development as a defect-reduction practice. In *Proceedings of the 14th International Symposium on Software Reliability Engineering*, ISSRE '03, pages 34–, Washington, DC, USA. IEEE Computer Society.

STATUTORY DECLARATION

I hereby declare that I have developed and written this thesis on my own, and no external sources were used except as acknowledged in the text and footnotes. The thesis in this form or in any other form has not been submitted to an examination body and has not been published.

Paderborn, February 28, 2015

Jörg Amelunxen