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, April 17, 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

Within this thesis, we have developed a backend for a web application, using an agile development approach. This backend allows students to release their house- and study work within a smartphone app to the public, after it has been corrected by the supervisor. Besides the actual development of the application, which is based on 2.0 Play and AngularJS, a part of the topic of this thesis is the methodology within an agile approach itself.

ZUSAMMENFASSUNG

Innerhalb dieser Masterarbeit haben wir mit einem agilen Entwicklungsansatz ein Backend für eine Webanwendung entwickelt, die es Studierendenen ermöglicht z.B. ihre Haus- und Studienarbeiten später innerhalb einer Smartphone app für die Öffentlichkeit freizugeben, nachdem diese durch die Betreuer korrigiert worden sind. Neben der eigentlichen Entwicklung der Anwendung, die sich auf Play 2.0 und AngularJS stützt, behandelt ein Teil der Arbeit das methodische Vorgehen in einem agilen Ansatz selbst.

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. Thank you, Dr. Simon Oberthür for doing a very good job as a supervisor. You were there, when I needed help; but on the other hand, I had a enough freedom to materialize my ideas and thoughts.

Furthermore, I would like to thank all people who proofread my thesis and Björn Senft for offering me the usability review. The review was very important and increased the quality of the backend recognizable.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ABBREVIATIONS

AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
AR	Augmented Reality
AREL	Augmented Reality Experience Language
BDD	Behavior Driven Development
BLE	Bluetooth Low Energy
BSON	Binary JSON
CD	Continuous Delivery
CMMI	Capability Maturity Model Integration
CMS	Content Management System
CPU	Central Processing Unit
CSS	Cascading Style Sheets
DOM	Document Object Model
GPS	Global Positioning System
GUI	Graphical User Interface
HiP	History in Paderborn
HMAC	Hash-based message authentication code
HTML	Hypertext Markup Language
HTML5	Hypertext Markup Language V5
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
JS	Javascript
JSON	JavaScript Object Notation

JWS	JSON Web Signature
JWT	JSON Web Token
KVS	Key Value Store
MVC	Model-View-Controller
MVVM	Model View ViewModel
NCSU	North Carolina State University
NPM	Node Package Manager
OSM	Open Street Map
OsmAnd	Open Street Map Android
PaaS	Platform as a service
PDF	Portable Document Format
RAM	Random-access memory
REST	Representational State Transfer
SHA256	Secure Hash Algorithm 256
SOAP	Simple Object Access Protocol
SPA	Single Page Application
TDD	Test-Driven Development
UI	User Interface
UML	Unified Modeling Language
UML2	Unified Modeling Language 2
URI	Uniform Resource Identifier
WebGL	Web Graphics Library
WSDL	Web Services Description Language
XML	Extensible Markup Language

1

INTRODUCTION IN THE THESIS

This first chapter will introduce the benefits of the system that we will develope 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 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 providing 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 3, 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 sketched within this thesis, 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. Furthermore, we would prevent the problem that people are sitting next to an object (e.g., within a museum) but are using the app without getting in touch with the real-world object because the AR functionality induces activity of the user.

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

This master thesis contains six chapters. The first chapter contained an introduction and ends with this 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 Modeling 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))*

The following two chapters will contain details about the planned system. We dedicate a whole chapter to this part of the development process because an idea about the system, its behavior and the background of the used technologies is very important and the foundation of the complete following work (just like the quotation above shows).

In general, the planned application should be able to handle the complete workflow for a university course situation. So, the backend should offer features that enable groups of students the work on specific topics. The domain model showing this situation for an (offline-) general university course is shown in Figure 1. As one can see, students are working together in groups to create content for a specific topic. The whole process is supervised by the supervising employee of the university. We will project this way of working into our backend application.

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 have been 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 of them.

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

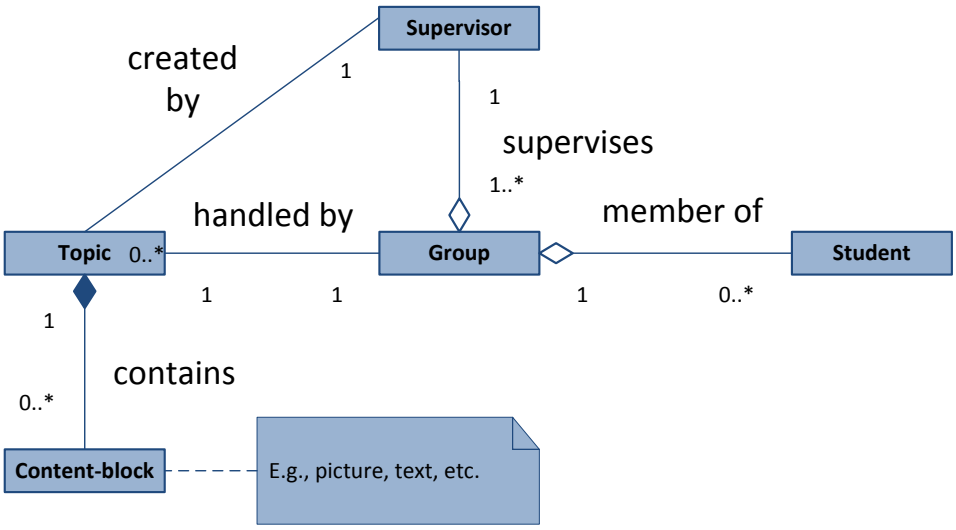

Figure 1: The domain model of the university courses.

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 organizational 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. The HiP-application will be developed in a Scrum-like fashion to achieve a high efficiency in the small timeframe and to be able to compare the experience with the Scrum like process with a classical development process. However, a full Scrum approach is not possible because the development team is very small (i.e., we will combine different roles in one person). Nonetheless, we will include the ideas and concepts of the Scrum process. 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 sprints with a length of about 2 weeks. In any case, after every sprint the product should be more re-

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

fined (Schwaber and Beedle (2002)). Of course, 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 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 the short timeframe of the master thesis. The close customer relationship can for example be found in the fact the the customer is often invited to 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.

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 and agile development in general, 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

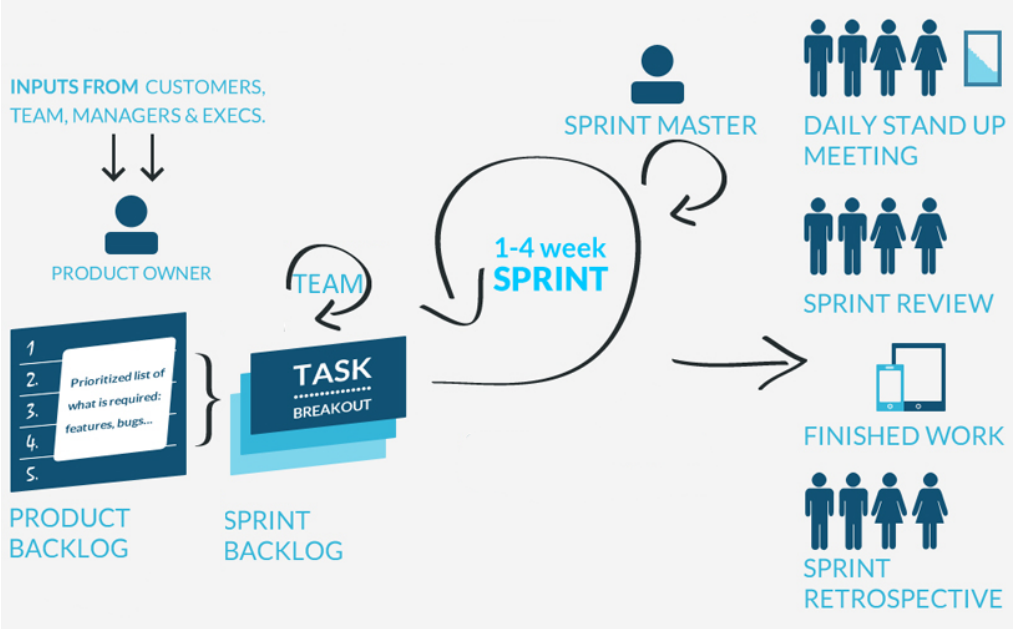

Figure 2: The Scrum development process (Simplified, original work from Maxxor (2015)).

the requirements if needed (Beck et al. (2001)). However, changing requirements within a currently running software development are a common cause of problems with respect 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 (Ceschi et al. (2005)). This means, 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

To get the idea of burn-down charts, we need a first intuition about story points. Story points 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 an unit of needed time* - .

A burn-down chart shows the amount of remaining story points 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 timeframe 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, the developers are to slow in the current sprint and in the case the sprint curve is below the *OPT* line, the developers are ahead of the time.

Obviously, burn-down charts do not have problems with changing requirements outside of the current sprint because they only track the information within 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. With this knowledge about the management of tasks and time estimation, we can take a closer look at story points.

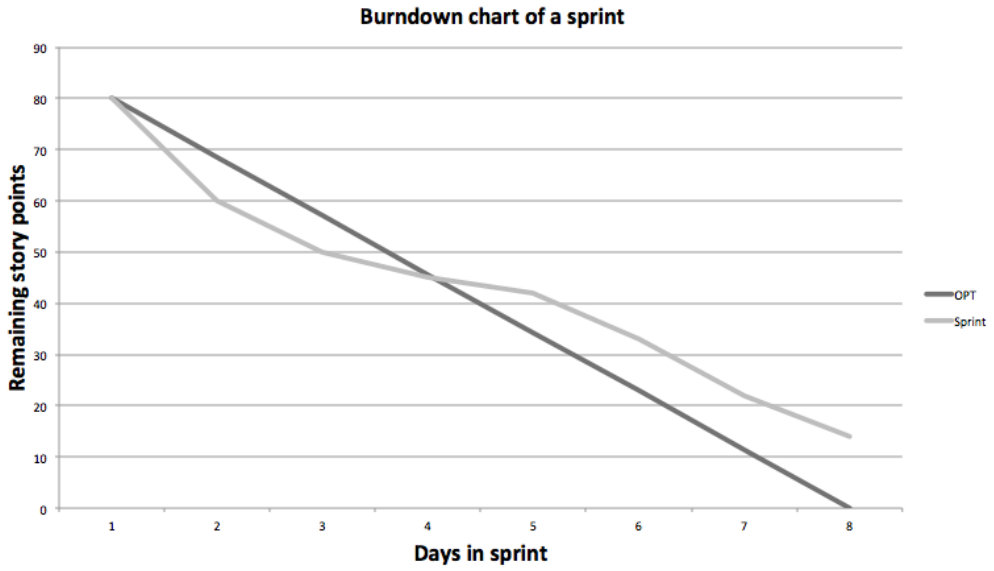

Figure 3: A burn-down diagram showing an example sprint.

2.2.2 Story points

In the previous section we have implicitly used the unit *story points*. Story points are tightly coupled with the idea of agile development because they represent an unit of needed time that 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 performance of the team. 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 team-member 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)).

Now, as we have seen how to manage the reduction of requirements with burndown charts and story points, we will take a look at the way we can specify user requirements in the first place; by using user stories.

2.2.3 User stories

As Pichler (2010) describes, an user story explains how a customer or an user uses the final the product in a short story. To create respectively discover these stories, one can use personas. Such a persona is a fictional character that represents a specific user type that might use a website or a product. A persona for our HiP backend could for example be *Steve*, the 24 year old student of english history that needs to create his homework with the backend. Obviously, Steve would represent the role *student* from the view of the backend.

Another important fact about user stories is that they are closely related to agile development. This relation can, for example, be seen at the fact that each user story is expected to result in a contribution to the value of the general product. This contribution should be added regardless of the order of the actual implementation (Alliance (2013)). Of course, this is a necessary condition for a user story, if one thinks about the way the development team works on the backlog of the product. The entries within the backlog are sorted into sprints and each entry within a sprint can be implemented as an autonomous unit by the development team. So, in general, one do not have any dependencies on entries within the sprint backlog.

2.3 DEVOPS

Besides the cost estimation and requirement engineering, 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, the term *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 one learns about the induced problems much faster because one gets much faster feedback from the enduser. Obviously, it is easier to fix a bug in such a situation, as in the case that the customer detects a bug in a 'new' release, which is four month old on the development machines. This results also in the benefit that the 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)). One of these approaches is the fact that, as soon as one starts thinking in much smaller autonomous chunks, it becomes much easier to migrate the developed system into a cloud service in a Platform as a service (PaaS) like fashion (Cukier (2013)).

One of the tools you need to create a DevOps process (respectively a DevOps culture), is Continuous Delivery.

2.3.1 Continuous Delivery and Continuous deployment

Continuous Delivery (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 *continuous 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

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. The complete process is also shown in Figure 4. As one can see within this figure, the process is quite straight forward and can easily be automated by using tools and platforms like *Bamboo* and *GitHub*.

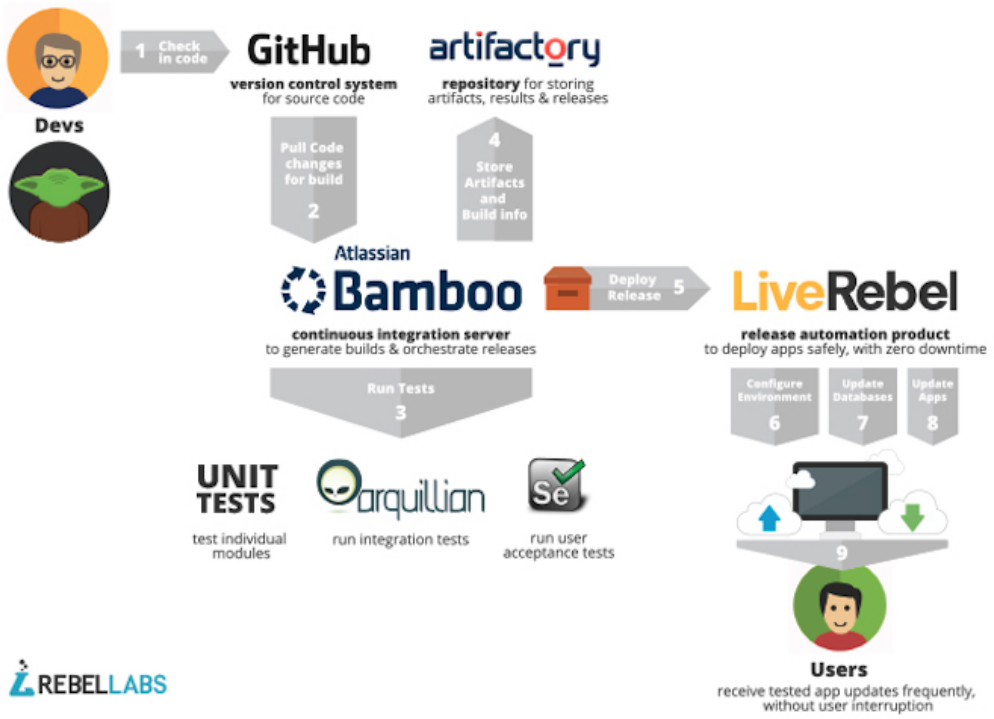

Figure 4: The CD process showing some tools that can be used to create a CD pipeline (taken from Wattson (2013))

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 timeframe 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 HTTP-interface and handles the routing from HTTP-requests to application code. On the frontend-side, AngularJS will be used to create a

fast and responsive web-User Interface (UI) 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 an 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 automati-

cally created from the Scala/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 in Play 2.0 is used to do the data handling, which is quite easy because the Play framework maps automatically the documents stored in the MongoDB to concrete Scala objects.
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 Hypertext Markup Language (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 Listing 2.2. In line 1 of Listing 2.3, we declare the used parameters (i.e., a String variable) that we have used to pass the String "Your new application is ready". The *@main* command in line 3 calls another template, which includes everything besides 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. In more detail, it offers an interface to get the data from users, who are currently logged into the system. Furthermore, it offers out of the box support for web-services like Twitter, Facebook, Google, LinkedIn and GitHub. If one do not want to rely on external services, *SecureSocial 2* provides a Username/Password mechanism with signup, login and reset password functionality. We will use this last technique within the HiP backend.

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).

The core ideas of REST are:

- A resource is identified by a persistent identifier (i.e., the URI).
- A resource (on the server side) is manipulated by using the fitting HTTP methods: POST, GET, PUT, and DELETE. So, one should not use HTTP GET to change something on the server.
- The actual representation, which gets downloaded as a resource from the server, is dependent on the request itself and not the used identifier. As stated in the first point of the list, every identifier identifies exactly one concrete resource. So, one should use HTTP Accept headers to control the needed representation (e..g, Extensible Markup Language (XML), JSON, etc.).

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 (i.e., Facebook, Twitter, etc.) are taking up on REST (Rodriguez (2008)). Besides the REST support, 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

MongoDB is a document-oriented database, which stores data as JSON objects. Thus, data entries within the MongoDB are called documents and are in essence ordered sets of key-value pairs (Trelle (2014)). However, values can

also be complete documents and arrays. So, one can store complex hierarchical structures within a MongoDB. Similarly, binary data (e.g., pictures) gets stored in a Binary JSON (BSON)-format. This format is a binary JSON format that includes datatypes and better traversability.

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

Every document within the database needs 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 an *_id* field, the field gets automatically generated.

An insert into a MongoDB is easily done and shown in Listing 2.4.

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 2.4 by matching against his lastname, one would need to do the steps shown in Listing 2.5. The object that is used to match the query is created in line 3. The query itself is started in line 7 of Listing 2.5.

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);
```

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 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 Model View ViewModel (MVVM) architecture and tries to create the same behavior by doing changes to the ViewModel. The ViewModel sits behind the concrete UI layer and exposes data needed by a View from a Model. Because of that, the ViewModel can be viewed as the source of data for the Views to get data and call functions.

Listing 2.6: Simple example that shows the use of an AJAX request that shows the response 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 shows a simple example, where an expression is used to call a method of a controller object.

Listing 2.7: Simple example that shows the use of expressions

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

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. This feature 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. The main drawback of the two-way data binding is a lower performance. This problem is also known by the developers of AngularJS. Because of that, Angular 1.3 introduced faster one-time data binding that allows for a model to be updated once from the value set by the controller (Peterson (2015)). We will need to keep this in mind, to create fast code with AngularJS.

Furthermore, AngularJS offers directives like *ng-class* that add a specific class to a DOM-element dynamically 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. It is, for example, possible to create a directive called *chat-box* that can directly be 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. A direct dependency on Google Chrome is not possible because we will not have any influence on the browser that is used by students and supervisors. 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 creation 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 by a lot of common web-applications. Thus, we enhance the external consistency of the HiP-application in respect of other web-applications (Lidwell et al. (2010)), which may be well known to the user.

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

Furthermore, Twitter 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.

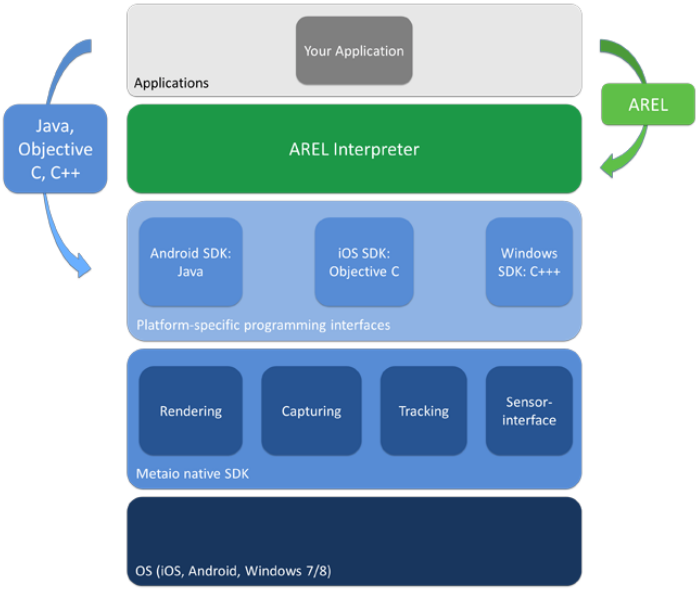

Figure 5: The figure shows the placement of the AREL interpreter within the platform stack. (Taken from Dev.metaio.com (2015))

Furthermore, Metaio has developed a JavaScript binding of the SDK used for AR-applications called Augmented Reality Experience Language (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 5 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 AREL layer

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

First of all, we need a static content definition, which is an 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 Graphical User Interface (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.

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 and JSModeler

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. However, we will use libraries like *JSModeler* to abstract from the low-level OpenGL commands and simplify the development process.

JSModeler provides a 3d object viewer for presenting 3d models and small scenes on a web page. It is able to import models with the formats 3DS, OBJ and STL. Furthermore, it is able to manipulate the points within the internal buffer, so we will be able to manipulate rendered objects.

After we have now taken a look at the used frameworks and technologies, we will now change our view to the used techniques and tools.

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 that 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 Nerur et al. (2005), 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, the relation between writing test cases and designing a system seems to be a bit odd. However, as Janzen and Saiedian (2005) point 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) state that the positive aspects of the usage of TDD has also been shown in studies of the North Carolina State University (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 Alliance (2015) 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, "Say 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

As we have seen in the last section, 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, Behavior Driven Development (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 and GitHub

We will use Git, which is a commonly used distributed revision control and source code management system, for the versioning of our project 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. An example of such supported estimation tools are burn-down charts that have 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 respectively frameworks are elaborated and evaluated, include a prototypical implementation of the needed components of the backend system.

But for now, we will start with the same first step as every development process; the requirements engineering phase.

3.1 REQUIREMENTS ENGINEERING

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*. Within this meeting, we talked about the main ideas of the application and tried to grasp the needed functionality. In the end, we ended up with a couple of cards with written user stories. Some of these cards are shown in Figure 6. The cards were also sorted within the meeting to create clusters of connected requirements.

After this meeting, 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

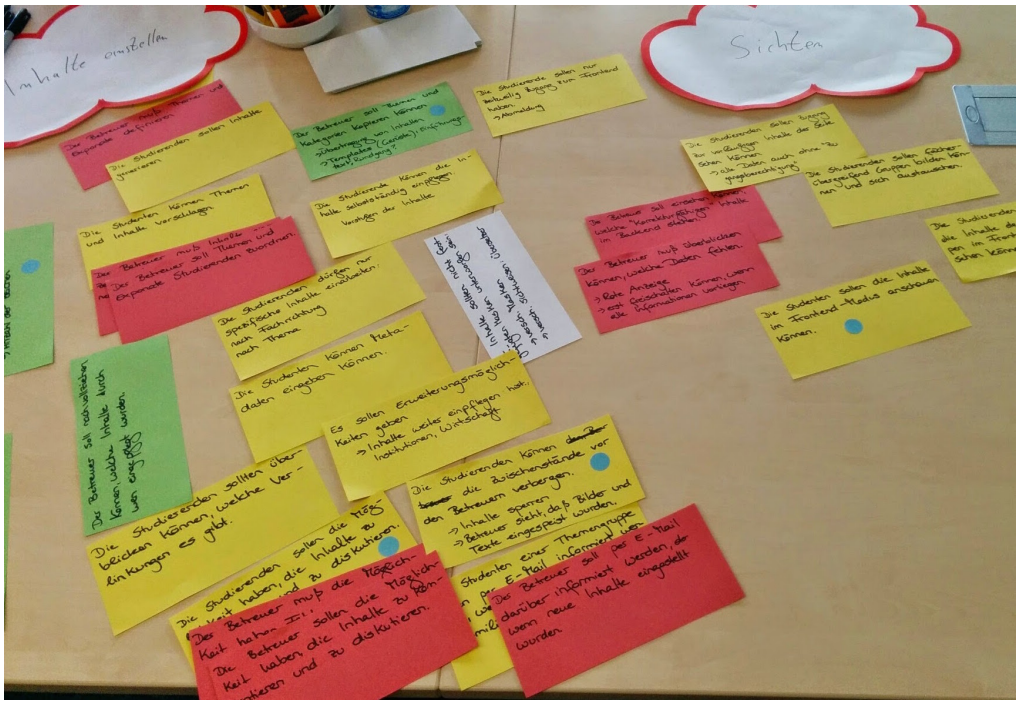

Figure 6: Some of the requirement cards that has been sorted within the requirements engineering meeting.

in tables 5, 6, 7, 8 and 9. These requirements can directly be used to derive some the needed test cases for the TDD approach.

One of these main features that has been elaborated in this meeting, was the need for four different roles within the complete system. We will now take a more detailed look at these roles within the following section.

3.2 USE CASES

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

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 a 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

The four different roles are also shown in Figure 22 as an UML2 use case diagram, which is placed in the appendix. This 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 outline two use case tables as an example for the usage of the system. However, a complete list of use case scenarios would be to large for this thesis because this could fill a book on its own.

Table 1: Use Case Scenario: Student changes content of 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 in case 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 is shown within Table 2. It shows that it will be quite easy for a supervisor to create a new group. The only steps he needs to take is to log into the system, open the correct view and input all needed information (e.g., members of the group).

However, even these two simple use cases includes complex interaction between the different controllers and views within the system. This is shown in

the UML sequence diagrams within Figure 7 and Figure 8. Note that both UML sequence diagrams does not contain the actual storing within the database to keep them on a manageable level of complexity/details. So, both diagrams contain only *«UI»* and *«Controller»* elements. Both UML sequence diagrams show AngularJS partials as *«UI»* elements. These partials are HTML parts that can be exchange and offer different UI elements. The connection to the backend is done by the *Play framework routing* controller, which redirects the HTTP requests to the fitting controller.

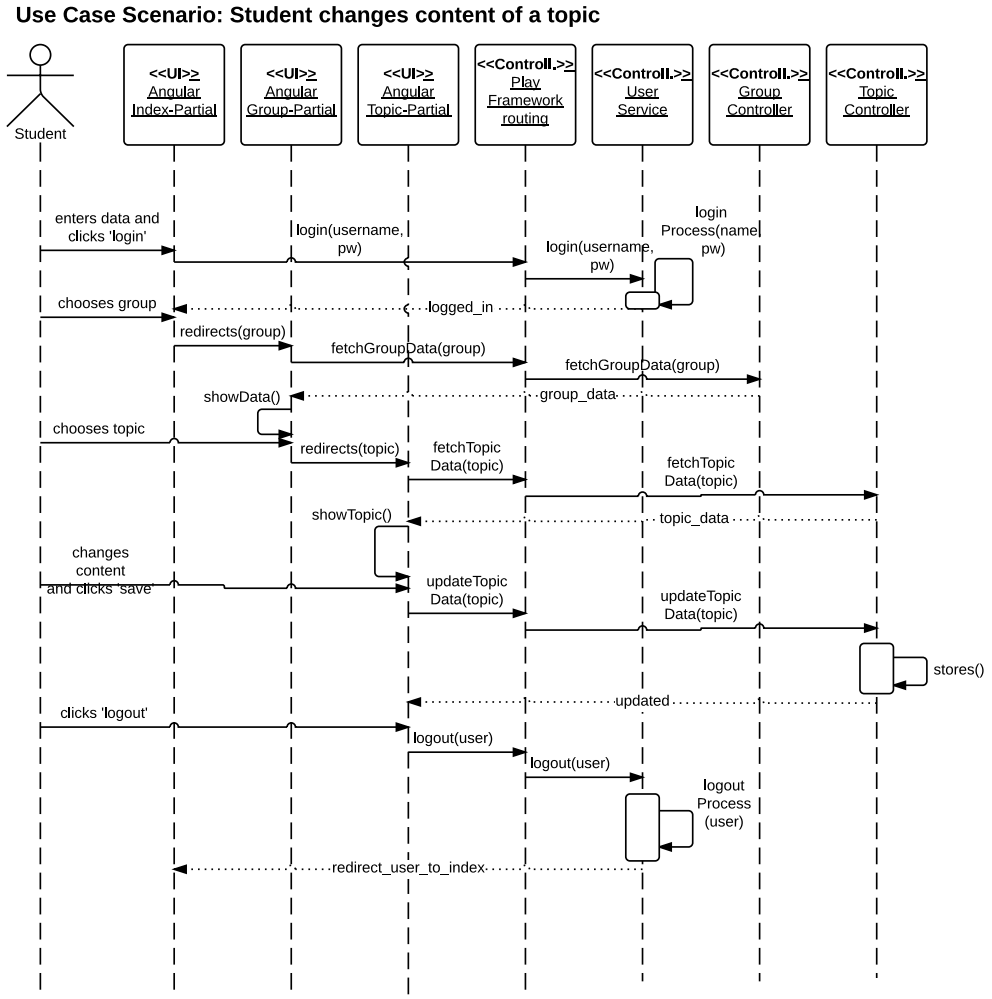

Figure 7: The corresponding UML sequence diagram for use case ‚ÄöäÄ*Student changes content of a topic*‚ÄöäÄò.

Figure 7 shows the interaction that is needed to exchange content within a specific topic. The student is able to login at the *index partial*, which redirects the login to the backend via the *Play framework routing*. After that, the student gets redirected to a new partial (i.e., *the group partial*, where he is able to select the topic he wants to edit). The third involved partial is the *topic partial* that shows the chosen topic and offers UI topic editing functions.

In a similar way, Figure 8 shows the process and the interaction for the creation of a new group. Since this use case does not need any information about a topic, we do not need the topic partial and the topic controller. Most of the interaction is similar to the shown diagram in Figure 7. The group controller handles the creation of the actual group and sends the storage request to the database within the *storing()* function (as stated before, the actual storing is not included in these sequence diagrams).

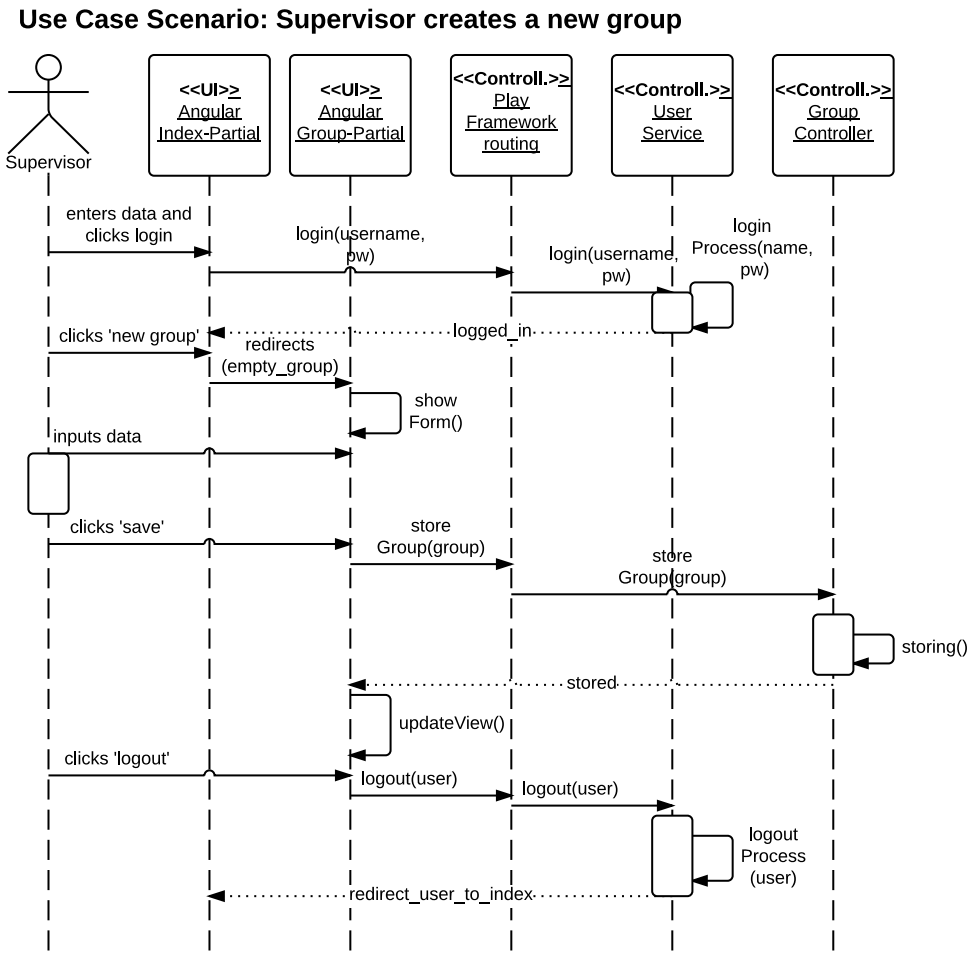

Figure 8: The corresponding UML sequence diagram for use case ‚Äö*Supervisor creates a new group*‚Äò.

Now, after we have seen what the application is about and got a first idea about used components within the UML sequence diagrams, we will now take a closer look at the planned architecture.

3.3 ARCHITECTURE OF THE APPLICATION

The architecture design phase within 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 on its whole, before the actual programming phase is started (Microsoft (2009)). This is, obviously, not applicable within an agile development approach as not all requirements are stated in the beginning of the development process. As Mast (2013) points out, agile architecture design has the following important attributes:

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 approach 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.

That said, we will choose a 2-tier 3-layer architecture for the development for the application. This is, according to Eckerson (1995), a quite common architecture for Client/Server respectively web applications. Within our application, the 3-layer architecture is nice because it enables us to exchange the first layer (and thus the first 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 9. The figure shows also the storage layer, 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 layer on the second 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, some parts of the application layer need

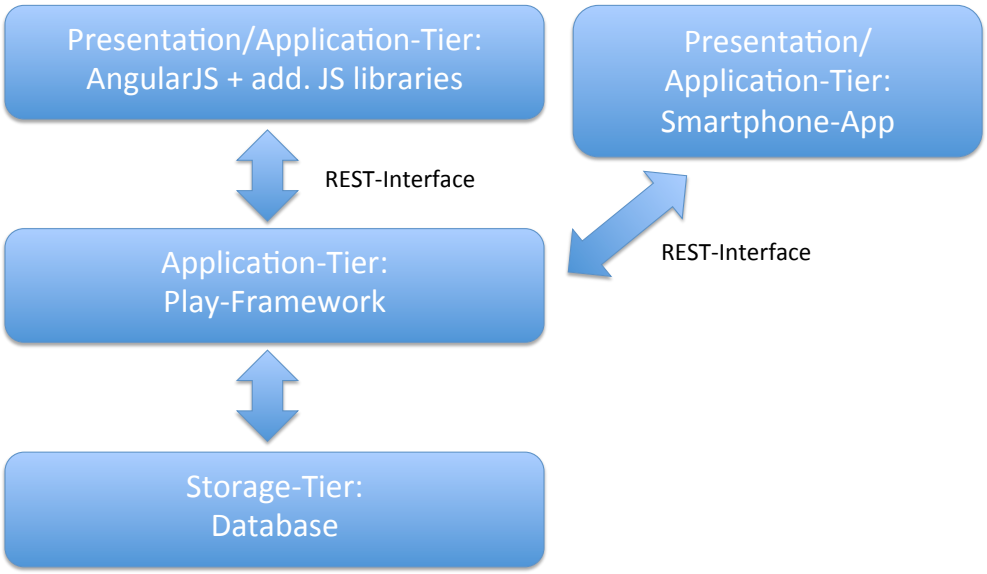

Figure 9: The general 3-tier architecture of the HiP-application with both presentation tiers (i.e., web frontend and smartphone frontend).

to be included on the client side within the presentation layer. Because of that, the presentation layer will be much more complex as it is shown in this brief draft since AngularJS, which is used in this layer, relies on a MVVM architecture on its own. 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. Nevertheless, we need to take a look at the usage of the different components (e.g., frameworks, libraries and tools) and how they 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 layer (i.e., the presentation layer). To offer all needed features of the backend in this short time frame, we will need to include a couple of frame-

works and libraries. A brief overview about the components and how they work together is shown in Figure 10.

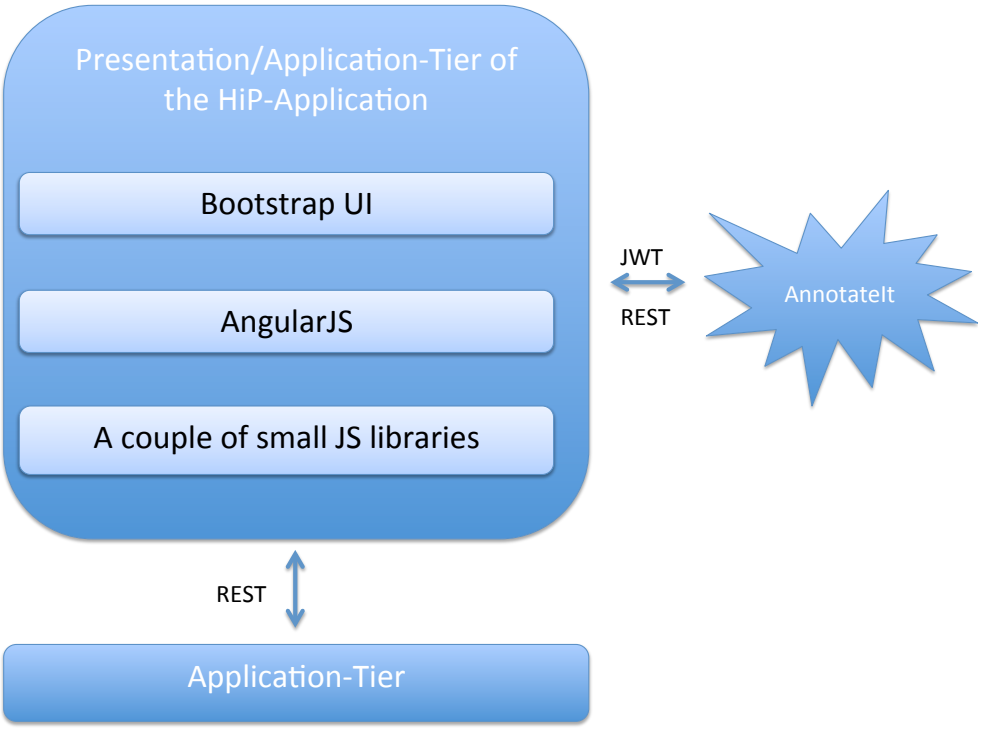

Figure 10: The usage of the different components and how they work together.

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 JSON Web Token (JWT). This JWT is a digital claim encoded as a JSON object that is digitally signed using a JSON Web Signature (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 data 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 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 21, which can be found in the appendix. The figure shows the complete workflow within the backend system as a 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 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 11. These annotations should also be assessable and (un-)lockable for the supervisor.

The following subsections will take a closer look at specific parts of the backend to create a better foundation for the cost estimation at the end of this section.

3.5.1 Input data/content via the CMS in the system

The storage of the data respectively content of the topics and groups will be mainly done by using the MongoDB. For example, a topic will be an object with information like version number, content, etc. Besides the general topics, the system will also able to handle subtopics, which are more detailed/refined

Figure 11: 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

topics that belong to the main topic. For example, if the main topic handles a specific church, than typical subtopics could be *creation and history*.

So, when an user changes something at a topic, the data is changed on the model within the view and send back to the model in the backend. This can be done quite easy 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. However, this way of data handling (i.e., modifying on the client side and sending data back as soon as the user hits the *save* button) induces problems if more than one user works at the same time at the same topic. Imagine user A and B start to work at the same topic at the same point in time. Then both download the same content on their client side but if A clicks *save* before B, then B will override all changes from A. We will prevent this problem by creating a *lock* mechanism on a topic. So, if A works on the topic, B is not able to start editing in the first place. The now induced efficiency problem (i.e., only one user can work at a topic) will be reduced by offering features for creating different text blocks that are

autonomously locked. So, A is not locking the topic *Südring center* but only a part of it (e.g., *Südring center - Introduction*).

The following pseudo-code in Listing 3.1 shows the algorithm that is used to lock the topics.

Listing 3.1: Pseudocode of the locking algorithm that is used for preventing multi-user from overwriting content.

```
1  onChange(){
2    doIOwnTheLock <- (internalLock.lastChange <= 5min ago)
3    if(doIOwnTheLock)
4      // we are fine -> working is possible
5    else{
6      currentLock <- db.locks.lock(topicUID)
7
8      topicIsLocked <- (currentLock.lastChange <= 5min ago)
9
10     if(topicIsLocked){
11       // we cannot modify the topic
12       // update lock again in 5 minutes
13     }else{
14       newTimestamp <- Date.currentTime
15       db.locks.lock(newTimestamp)
16       internalLock.lastChange = newTimestamp
17     }
18   }
19 }
```

The main idea of this algorithm is that we check if we are able to create a lock (resp. even own the lock) on the current topic before we do any changes (line 3). On a technical level the lock is a timestamp that describes the last time when the topic has been changed. For checking the availability of the lock, we look at the timestamp and compare it with the current time. If the last edit is less than 5 minutes old, we wait 5 minutes and try again. If the lock is older than 5 minutes, we set it to the current time and save the used value in an internal variable for comparison. This way, we are able to check locally on the client side, if the current client is the client that owns the lock.

Besides this locking algorithm, 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 docu-

ments 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.

But including data in the system is not enough for working with topics. We need also ways to offer review features for supervising users.

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. This can easily be handled by the JS library *AnnotateJS* and will be explained in more detail within chapter 4.

The attachment of information and comments to a complete topic can easily be created by using the chat system and a new chat room, which is exclusively opened for that topic object.

If the supervisor is satisfied with the given quality, he should be able to mark the topic as 'ready for publish'. On a technical side, this boils down to setting a different flag at the topic object. So, the needed time for this feature should be low.

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

The augmentation editor will be needed to edit the objects (i.e., point clouds) that have been scanned with a smartphone by a student. However, because the needed file format is proprietary and the owning company Metaio does not publish any information about the format, we will not include a complete modeling editor within the backend. More details about the problematic file format are described in section 3.6.1. We will leave this problem open for the project group. Maybe they are able to reverse-engineer needed information or exchange the used technology (e.g., using *Wikitude* instead of *Junaio*).

Any way, the editor should in the end 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 11 shows a mockup of such a system. The mockup shows the rendered point-cloud and two buttons to edit the augmentations on this point-cloud. This information can afterwards be shown within the fronted application.

Because of the described problems, we will include a mockup system within this master thesis that should be able to render 3D-objects without editing features. To create this, we will use the library *JSModeler*, which is a great start for such a system.

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 only a rough estimation. Nonetheless, this rough estimation should be sufficient to get an idea about the needed time. However, note that some 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: 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 (e.g., in topics)	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

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.

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 12.

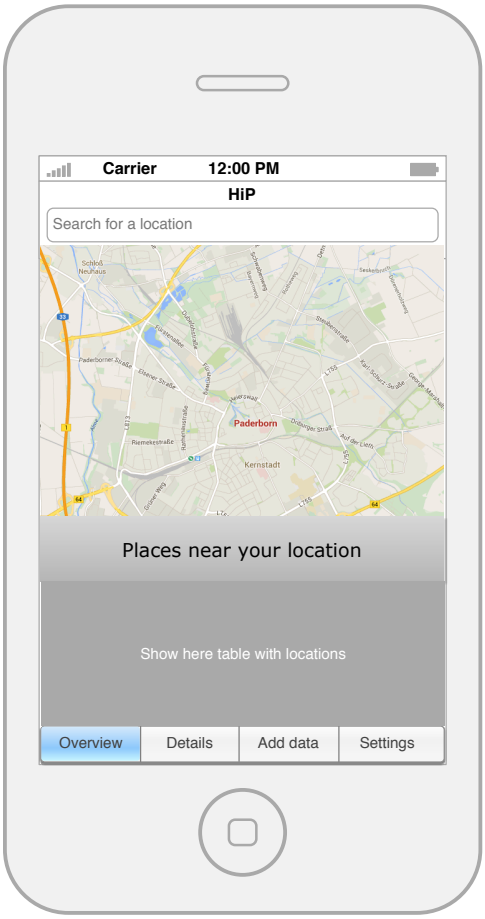

Figure 12: A mockup showing the main page of the frontend application showing a map of paderborn and a general overview about the UI-elements.

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 univer-

sity 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. This backend process has already been described in section 3.5.3. An mockup of this view is shown in Figure 13. The mockup shows an annotated object, which is now found within the AR-environment with the attached annotations. The tabs at the bottom of the screen are used to switch between different menus of the smartphone application.

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 within the draft.

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 to track the object with Junaio. The scanning process itself is done by the *Metaio Toolbox*. A screenshot of the *Metaio Toolbox* is shown in Figure 14. When an object gets scanned with the Toolbox, the file is stored on the smartphone's filesystem and can be shared by using email attachments.

However, because of the problematic file format, there may be a shift the the Junaio eco-system to alternative environments. The problems with the *creator3dmap* are described in the following section.

Data format for AR files

The scanned objects are saved within the *3dmap* and *creator3dmap* format. Both files are essentially *ZIP* files that contain metadata, pictures and binary data for the point-clouds. The original idea within this thesis was to use these files to render the point-clouds in the backend and offer features for manipulation. However, because we would need the file specification to be able to read the

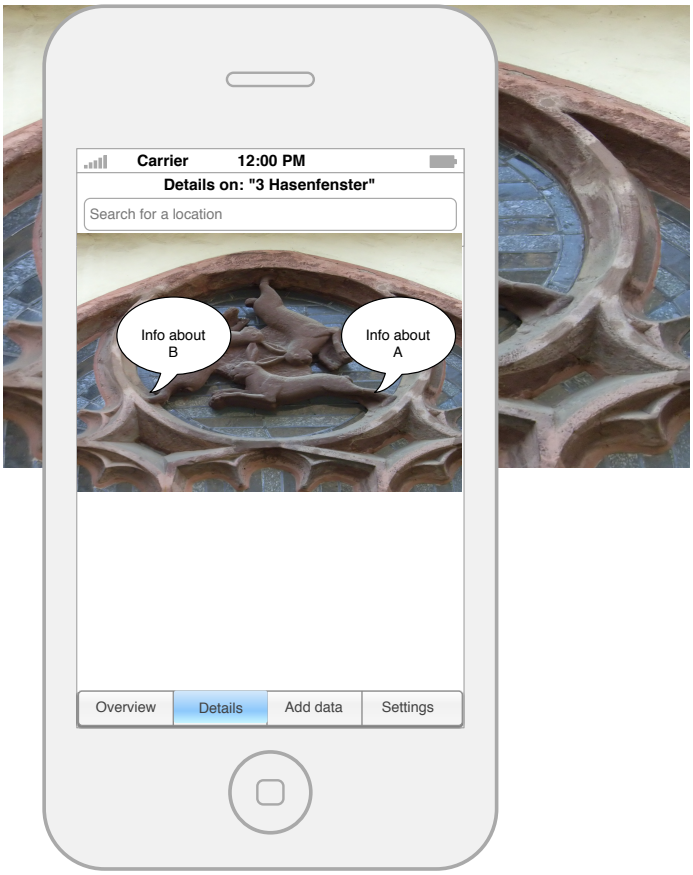

Figure 13: A mockup showing the details page of the "Dreihasenfenster" while the camera of the smartphone is pointing to the window itself.

binary data contained in the ZIP files, we asked for such a specification at the company Metaio. Sadly, we never received an answer.

Thus, this part of the thesis is postponed and left open for the Project Group.

3.6.2 Show close "interesting places" within a map

In the end, every topic will contain Global Positioning System (GPS) coordinates for exhibits and locations, which has been added by students. We can make use of this information to get a location specific map and navigation features on the smartphone.

Listing 3.2: Example for using the GPS coordinates within an Android application

```
1 import com.google.android.gms.maps.*;
```

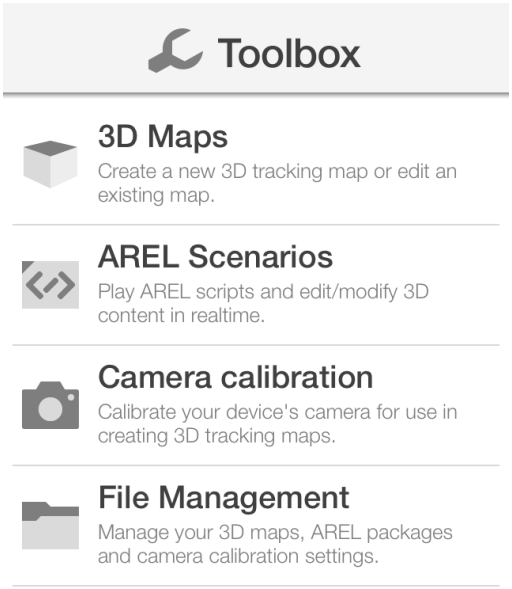

Figure 14: The main menu of the *Metaio Toolbox* showing the main features of the app.

```
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*. As Listing 3.2 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 service 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 3.3: Example for construction of the DirectionsRequest JSON object that is needed to use the directions service

```
1  {
2    origin: LatLng | String,
3    destination: LatLng | String,
4    travelMode: TravelMode,
5    transitOptions: TransitOptions,
6    unitSystem: UnitSystem,
7    waypoints[]: DirectionsWaypoint,
8    optimizeWaypoints: Boolean,
9    provideRouteAlternatives: Boolean,
10   avoidHighways: Boolean,
11   avoidTolls: Boolean
12   region: String
13 }
```

After doing that, we can access the direction service by posting a JSON object to the Google service and use the returned data to render the navigation path within our application. The construction of such a JSON object is shown in Listing 3.3.

As we can see, this 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 re-

turned data. This is, for example, possible by drawing lines on the map using the *google.maps.Polyline* class. This is shown in Listing 3.4.

Listing 3.4: 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'
  });

  flightPath.setMap(map);
}

google.maps.event.addDomListener(window, 'load', initialize);
```

As we can see in Listing 3.4, we can directly draw on the map, which is really fast. The listing shows the code in Javascript. The flightPath object is directly created as a Polyline object with the given coordinates. After that, it can directly be placed on the map. Although the shown code is JS, the Polyline class can, however, be accessed natively with the Android SDK (Google (2015)). Obviously, 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 just needs 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 Open Street Map Android (OsmAnd) project and use the code as a basis for the HiP-Navigation feature. OsmAnd is a map and navigation application that uses

the Open Street Map (OSM) data. The application offers routing with optical and voice guidance and 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. An example for such an overlay is one 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.

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.

However, by using GPS we are only able to creating a good location awareness for outdoor situations. As soon as the user of the smartphone enters building, we need to think about other ways to get a precise estimation about the position of the user. The smartphone application could include this by using iBeacons. An iBeacon is a new approach for creating very precise location estimations that has been developed by Apple. Technically, Bluetooth Low Energy (BLE) is used to connect to a device that supports the iBeacon technology within a region around the device (i.e., the object that has the iBeacon device attached). This allows a smartphone to determine whether it has entered or left the region and to create an estimation about the proximity to the beacon (Apple (2014)). Natively, only iOS since version 7.0 offers support for the usage of iBeacon.

However, there exist a couple of libraries on the Android platform that offer solutions for creating iBeacon functionality for Android. Two of these libraries are created by Radius Networks and Sensorberg. Sensorberg's solution was for example used to create the iBeacon based indoor navigation at the Cebit 2014 (Kaufmann (2014)). However, Sensorberg focuses on the distribution of own hardware in combination with its library. So, the library of Radius Networks called *Android Beacon Library*, which offers mainly its library licensed under the Apache License Version 2.0, is more interesting for our project. The library allows Android devices to use beacons in a similar way like iOS devices. This means, every app using this library on the Android smartphone can request to get notifications when the smartphone reaches a close range to a beacon respectively leaves it (RadiusNetworks (2015a)).

The usage of the library is very easy and fast. One only needs to derive a new class from a class that is included within the library and overwrite a couple of methods. For example, by a simple overwrite of the method *didEnterRegion(Region region)* one can handle the reaction of the smartphone application when a new iBeacon has been found (RadiusNetworks (2015b)).

Because we are now able to fetch the location of the user indoor and outdoor, we will now take a look at possibilities to use this information.

3.6.4 Fetching topics and PDF export

The requirements states that as soon as an user arrives at a specific location, he should be 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 as soon as he arrives at the position.

Listing 3.5: 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 3.5 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 Portable Document Format (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 fast and easy within the smartphone application.

3.6.5 Rendering AR-data with Junaio

The rendering of the AR data is another very important part of the HiP frontend. 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. The static content definition contains the used models for that specific scene.

The concrete costs of this part of the frontend are very hard to estimate because we lack a couple of information about the way we can integrate the data we get from the Metaio Toolbox. Nevertheless, we will try to create a feasible estimation within the next subsection.

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 shows the minimal features of the smartphone frontend, which sum up to 34 story points. So, the frontend system should cost about 222 hours to create a prototypical but running system.

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
Include iBeacon support	The user should be able to get information about iBeacons on its location	3
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
		$\sum$ 37

4

IMPLEMENTATION DETAILS

This chapter will show details about the implementation of the HiP application. However, note that the application is too complex to be explained in full detail in this chapter. Because of that, we will start by explaining the implementation in general on an abstract level and show only specific implementation parts, afterwards. The shown implementation parts are chosen because we think that these parts are important within the system and deepen the understanding of the application.

Furthermore, we will show details that has been changed from the draft described in section 3 because the (changed respectively new) requirements came up within the implementation phase in the agile process. Of course, these changes are 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*. These changes have also induced a couple of architectural problems because some ideas and structures that has been used for the current version of the HiP application are already deprecated but this problem will be explained in more detail in chapter 6.

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 handled by the Play framework with a single line of code within the configuration file of the Play framework (i.e., the application.conf file):

```
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:Port/hip"
```

After Play 2.0 has created the connection to the MongoDB we can use the connection within the framework by extending a controller with the *MongoController* trait (a Scala trait is similar to an interface in Java). By doing this, we have created a couple of Scala controllers that form the second tier of the application. The main function of these controllers is the handling of data on the way to the database. Nevertheless, as we will see, the second tier includes also functions for creating thumbnails for uploaded pictures and the creation of JWT.

Besides these features on the server side, 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 soon 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, one 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 that are interconnected.

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 big parts of application logic. They can be created by controllers and are working completely autonomous. This is good for the creation of a system that stays easy to maintain. 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 a 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.

So, now we have an idea about the four main parts of the client side. However, we need a more structured view about the way these parts are connected to understand the details of the application.

Figure 15 shows the main dependencies between the just described main parts of the backend as an UML package diagram.

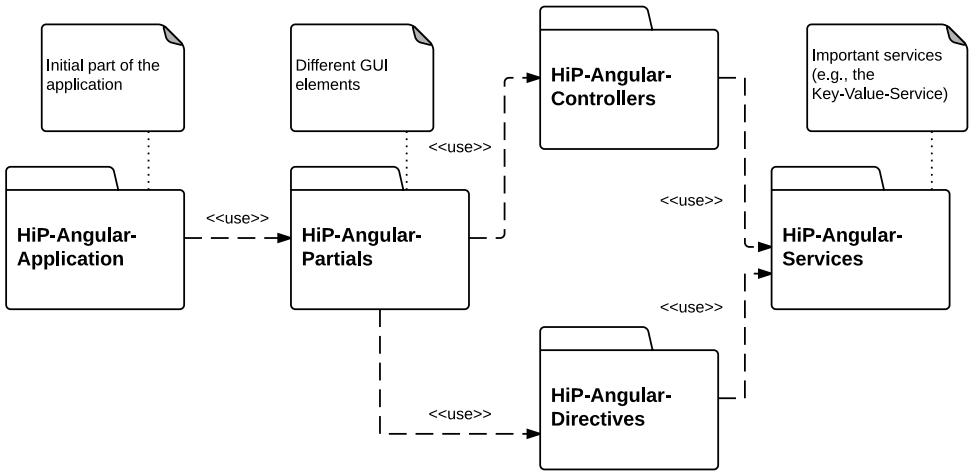

Figure 15: The UML package diagram shows the general dependencies between the main parts of the backend.

The figure shows that the main application is only using partials, which make use of controllers and directives. Both are using services to use encapsulated application logic.

A more technical view on the complete system can be found in Figure 23 within the Appendix as a UML deployment diagram. Because this technical view is not needed to understand the system itself but is useful to underline the main ideas of the architecture it has been shifted to the appendix.

After we have now gained some insights into the general architecture of the system, we will get a more detailed view on some parts of the current implementation. Our journey though the system will start at the server side within Actions for the Play 2.0 framework.

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 plugin creates ** HTML tags with the (css) 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 storing 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*, 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 description:

1. Alice registers the HiP backend by AnnotateIt, and receives a *consumer key/secret key* pair.
2. Bob (an 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 that Bob is a real user from HiP. Thus, it stores his annotation.

An example for such an JWT authentication token is shown in Figure 16.

As one can see, the token contains a header, which specifies the used algorithm, a body, which contains the payload and an authenticator, which is a Hash-based message authentication code (HMAC) based on Secure Hash Algorithm 256 (SHA256). The payload contains mainly the userID and consumer key of the application. So, this token proves that the request 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

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

Figure 16: An example for a JWT authentication token.
On the left side: The JWT in Base64 encoding. On the right side: JWT as the decoded token.

```
6
7         // Create HMAC signer
8         val signer = new MACSigner(sharedSecret)
9
10        // get current time
11        [...]
12
13        // Prepare JWT with claims set
14        val claimsSet = new JWTClaimsSet()
15        claimsSet setCustomClaim("consumerKey",   "
              ed83da60ae5e4d159729eef16a207525")
16        claimsSet setCustomClaim("userId",  [...]);
17        claimsSet setCustomClaim("issuedAt", nowAsISO)
18        claimsSet setCustomClaim("ttl", "86400")
19
20        var signedJWT = new SignedJWT(new JWSHeader(
              JWSAlgorithm.HS256), claimsSet)
21
22        // Apply the HMAC
23        signedJWT.sign(signer)
24
25        val s = signedJWT.serialize
26
27        Ok(s)
```

```
28          }
29          case _ => {
30            Ok(views.html.loginplease())
31          }
32        }
33    }
```

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 an *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 loginplease.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 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 in cases where 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 [...]
```

At this point, 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 Listing A.1 within the appendix.

After we have now seen a couple of important Actions within the Scala controllers at the server side, we will start to look at the code on the client side. To do this, we will start by taking a closer 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 written 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 17. 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 17: 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

```
controllersModule.directive('mediaGallery', function() {
    return {
        restrict: 'E',
        scope: {
            files: '=files',
            picturetooltip: '=picturetooltip',
            deletetext: '=deletetext',
            copyto: '=copyto',
            opentext: '=opentext',
            sendmetadata: '=sendmetadata',
            currentType: '=currenttype',
            updatetype: '=updatetype',
            lc: '=languagecontroller'
        },
        templateUrl: '/assets/directives/mediaGallery.html'
    };
});
```

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, which is a subset of the String `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 *isolated* 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 depend 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 18. The figure shows a template box from the view of a supervisor.

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.

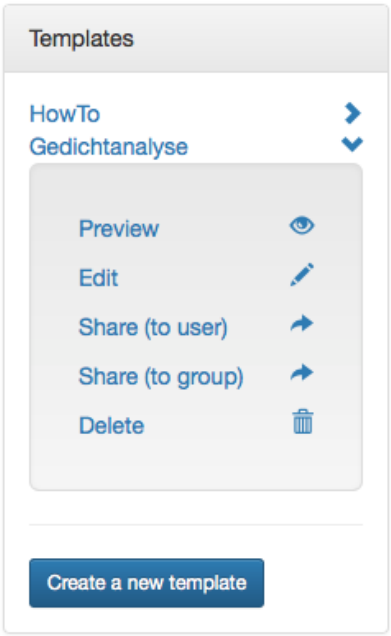

Figure 18: An example template box with two templates. The menu of the lower template has been opened by using the triangle on the right side.

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 manage-

able 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 19.

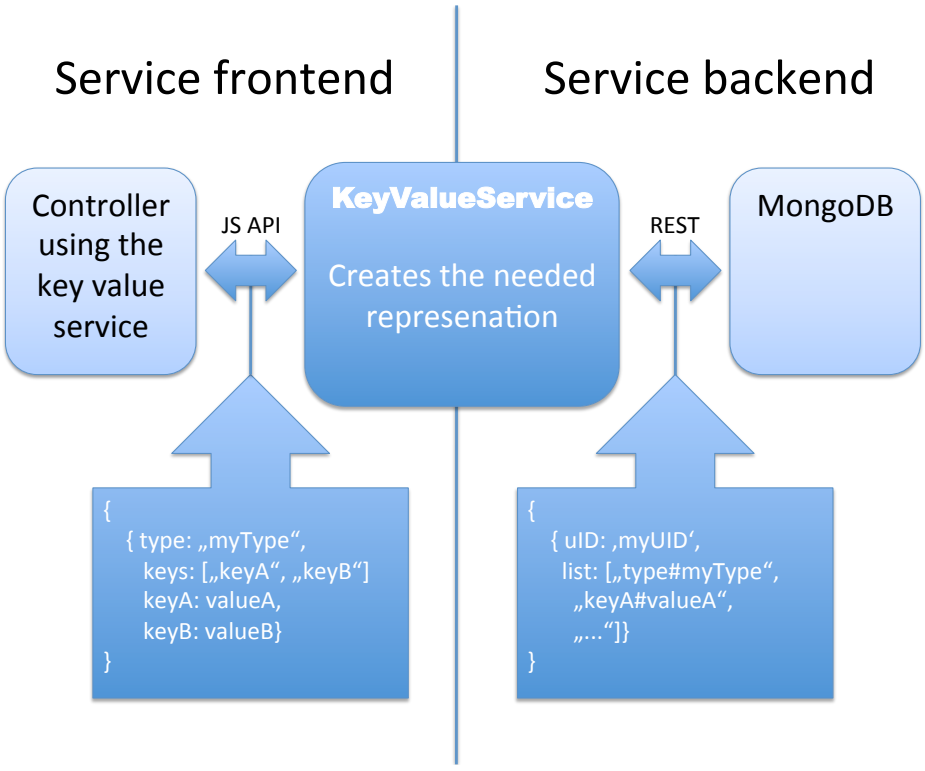

Figure 19: The frontend and backend of the key value service with the two different data formats

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 new situations and can also be seen in the fact that the service is used within the HiP backend to store textual templates and meta-data for pictures. 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 program 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 and split the key and value in line 5. The resulting array gets stored 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 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 Key Value Store (KVS) so reusable is the idea of typed key-value stores. A type specifies 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.

4.6 INTERFACE TO THE SMARTPHONE APP

As we have said before, the development of the smartphone application started as a bachelor thesis and will also be taken over by the project group. The app accepts data as JS objects. Listing 4.9 shows the structure of these objects for topics.

Listing 4.9: The format of topic files within the smartphone application.

```
1 {
2     "data": [
```

```
3           {
4               "categories": String,
5               "description": String,
6               "id": String,
7               "lat": Double,
8               "lng": Double,
9               "name": String,
10              "tags": String
11          },
12          [...]
13      ]
14  }
```

As one can see, the format consists of JS objects that are included in the array, which is the value for the key *data*.

Because the HiP backend uses another representation of the data (which is also contained within the MongoDB), we offer another URI to retrieve the file for the smartphone. Internally, we translate the representation that is stored within the MongoDB into the other representation for the smartphone. Note that this is not a break with the REST principle that we should not use different URI to specify the format. In this case, we do not only exchange the format (e.g., XML to JSON) of the same data. We store other (i.e., less) fields within the smartphone object as in the original document that is stored in the MongoDB. So, after all, we create a new object with less fields (which has another format) and every object has its own identifier.

In a similar way, the interface for pictures translates the internally used identifier to the frontend-app style. Internally, a picture is an autonomous object that is identified by its unique identifier. Furthermore, every pictures contains meta information about the topic it is used in. This way, we can easily include as many pictures as we want in one topic. Timo Boegeholz, who wrote the first frontend application in his bachelor thesis, used another approach and identifies pictures with the id of the topic that it contains plus the suffix *.jpg*. So, we provided a new interface that accepts calls like

hiproot/admin/app/pictures/topicID.jpg

and redirects the call to the internally used interface

hiproot/admin/pictures/mainPictureID.jpg.

The main picture of a topic has to be chosen in the hip backend. Figure 20 shows a screenshot of the HiP frontend that uses the provided data of the backend.

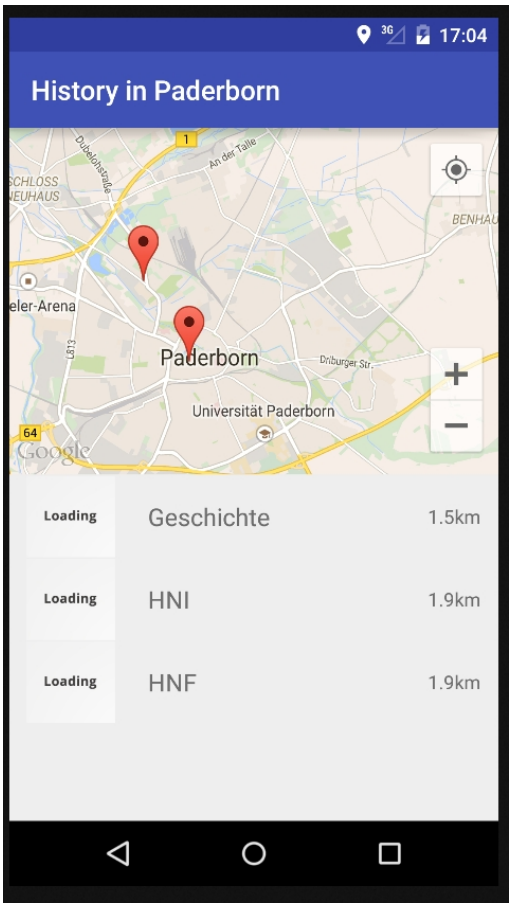

Figure 20: A screenshot of the smartphone frontend with data provided by the current version of the HiP backend.

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 chapter 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.

The following section will contain the results of the final test suites for the current version of the HiP application. Thus, this summary does not cope with the importance of testing within the development process but it cannot be shown or expressed in a better way within this written thesis.

Nevertheless, testing is only a small fragment with respect to a complete quality assurance according to the quality model described within the ISO/IEC 9126, which will be described in the next section. However, functional quality assurance (i.e., testing) was most that could be achieved in the time frame of this master thesis.

5.1 QUALITY ASSURANCE AND QUALITY MODELS

Software quality is a term that is hard to grasp and could potentially include very different things. To create a common understanding of quality and to create a formalized expression of quality, *quality models* have been introduced (Waghmode and Jamsandekar (2013)).

The term *quality model* has been defined by Deissenboeck et. al. as *'a model with the objective to describe, assess and/or predict quality'* (Deissenboeck et al. (2009)). So, as we have said, it is the needed foundation for every common understanding or evaluation of the fuzzy term *quality*.

Although, a multitude of quality models have been proposed and applied within the last years, we will focus on the quality model that has been in-

troduced within the ISO/IEC 9126. This quality model itself is mostly used to define the term *quality*. Furthermore, the ISO/IEC 9126 offers metrics to *measure* the defined quality of the product. However, we will only scratch the surface of this topic within this thesis because software quality models are quite complex.

To define the term quality, the ISO/IEC 9126 defines six characteristics, which are again divided into sub-characteristics. The six main characteristics are *Functionality*, *Reliability*, *Usability*, *Efficience*, *Maintainablity* and *Portability* (Jung et al. (2004)). We will not get into details here because we try to keep this section short and the terms should be sufficient to get an intuitive idea about these quality characteristics. The defined sub-characteristics of these main characteristics can be evaluated by the metrics that are proposed within the ISO/IEC 9126.

To get a better understand of these measurement metrics, we will take a closer look at one sub-characteristic of *Reliability*, which is called *Maturity*. The ISO/IEC 9126 defines a maturity metric as *'an external maturity metric should be able to measure such attributes as the software freedom of failures caused by faults existing in the software itself'* (ISO/IEC (2001)).

As an example, we will take a look at the metric *test coverage* (ISO/IEC (2001)), which we will also use in the next section to measure the quality of our test suites.

Name: Test coverage

Purpose: How much of required test cases have been executed during testing?

Method: Count the number of test cases performed during testing and compare the number of test cases required to obtain adequate test coverage.

Measurement: $X = A/B$, where A =Number of actually performed test cases and B = $Numberofestimatedtestcasestobeperformedtocoverallrequirements$.

Interpretation: $0 <= X <= 1$: Closer X to 1.0 is the better test coverage.

Scale: Absolute

Audience: Developer, Tester, SQA

As one can see, the ISO/IEC 9126 defines a metric with a name, shows the purpose and explains how the metric can be used and evaluated by the target audience. By using these metrics, we can evaluate the quality of the sub-characteristic. The important thing is that the main-characteristics are created from these sub-characteristics. So, we evaluate the quality of the main characteristics. Thus, we evaluate by definition the quality of our whole software

product by using this metrics because we defined software quality as the quality of the main characteristics.

So, after we have now seen this theoretical approach to software quality assurance, we will now take a closer look at the used test suites. As we have described in chapter 2, the tests have been developed with the help of the Jasmine framework.

5.2 TEST ENVIRONMENT

The Jasmine test suites were run within Karma on Mac OS X operating system with Google Chrome. The hardware configuration was a dual core Central Processing Unit (CPU) and 8096MB Random-access memory (RAM).

5.3 TESTING RESULTS

A typical test case within our test suites looks like the test case shown in Listing 5.1.

Listing 5.1: Simple test case for the type service

```
1  it('is able to fetch all types', function () {
2      var check = function(type){
3          expect(type.length).toBe(2);
4      };
5      service.getTypes(check);
6
7      $httpBackend.expect("GET","/admin/types").respond(200,
           typeList);
8      $httpBackend.flush();
9  });
```

As one can see, this is a unit test case written for the Jasmine framework. Line 1 shows the header of the test case, which can be read like a typical english sentence. The body of the test case contains the call of the actual function (line 5) and the matching against the expected value (line 3). Furthermore, one can easily check the commands that have been send via the REST interface with the mockup of the HTTP module of AngularJS, which is called httpBackend in the listing.

Because we restricted ourselves to unit tests and did not create any integration tests, we were not able to reach every line of code within our controllers and services within the unit test cases (a lot of code is only for handling events and user input). This can also be seen within the implementation of the acceptance criteria, which has been created from the requirements engineering (like the requirements in chapter 3, these criteria are shown in tables 5, 6, 7, 8 and 9). Some of these acceptance criteria could only be automatically tested with integration tests and are, thus, not tested within this thesis.

However, we ended up with 118 test cases with a statement-coverage of 71 percent. To express the testing result more formal, we will make use of the already defined metric called *test coverage*. Remember, the the metric uses the ratio between the actual amount of test cases and the estimated amount of test cases needed to cover all requirements. We estimate this number to be about 165 test cases (assuming the amount of code per test case that we have achieved with the current number of concrete test cases). So, our ratio ends up to be: $\frac{118}{165} = 0.71$. It is not surprising that the value is 0.71 since we express essentially the same data as in the simple state-coverage. In the end, the value seems to be quite fine, assuming we should get as close as 1.0 as possible, for a first prototype.

Note that unit testing was not the only part of quality assurance that we have used for this project. The next section will describe the usability improvements of the HiP application.

5.4 USABILITY STUDY OF THE BACKEND SYSTEM

According to Lin et. al., usability is an important part of every software system and the importance of usability for such systems is still rising (Lin et al. (1997)). Furthermore, *Usability* is a main characteristic within the quality model of the ISO/IEC 9126.

The ISO/IEC 9126 defines the term usability as 'a set of attributes that bear on the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users' (Bevan (1997)). The sub-characteristics of *Usability*) are defined as *Understandability*, *Learnability*, *Operability*, *Attractiveness* and *Usability Compliance* (Bevan (1997)). In general, it is very hard to evaluate usability problems by developers for systems they have written because they have a very different view as the end-user. This problem gets enhanced by the fact that usability problems are in many cases hard to grasp by objective met-

rics and are influenced by personal and cultural backgrounds (Herman (1996)). The most objective way to evaluate a lot of usability problems are checklists and questionnaires (e.g., did we include tooltips to increase the *Learnability* of the application? (Shamsuddin et al. (2014))).

Because of these problems, we tried to get professional feedback by an external usability expert to circumvent the problem that we need to evaluate our own system. Kindly, M. Sc. Björn Senft offered us an evaluation of the HiP application backend within an informal review session to find common usability problems.

Together with one member of the currently started project group, which will take over the development project, we sat together and made a complete 3 hour walkthrough of the application. In the end, we came up with a great list of small and not that small usability problems. Examples of such problems were, the missing of instant feedback for a couple of user forms and misleading color conventions. Most of these problems have been fixed in the last weeks of this master-thesis. However, some of these problems remain open and are listet in Table 10 within the appendix. The listed table contains information about the problems itself, if they have been resolved for the last version of this thesis and an expectation about the priority of the listed problem.

As we will see in section 6.3.2, the open problems may be handled by the project group.

6

DISCUSSION AND FUTURE WORK

This chapter will begin with a short explanation about the future development (respectively a transition to a project group) of the HiP system and will close this thesis with a conclusion and a brief view into the future, afterwards.

6.1 HANDING OF THE PROJECT

The fact that the backend has been developed in an agile fashion has a major impact on the way we are able to create a transition to a new development team. The transition is quite easy in this case because there are open problems/feature requests within the product backlog, which can easily be solved in small pieces by other developers. To accelerate the learning process for the members of the upcoming project group, we gave a short introduction talk and prepared installation instructions. This offers the members the opportunity to gain some insights into the current version of the HiP backend. As the members of the project group dived more and more into the application code, they were supported by emails if they had any questions on code or architectural decisions. More ideas about future work will be briefly explained in section 6.3.2.

6.2 ARISEN PROBLEMS WITHIN THIS THESIS

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

6.2.1 The agile process

The development of 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) the development team is unsure about the importance of specific features and problems (e.g., the needed keys for meta-data of pictures). But this information can easily gathered within the meetings to create an application that fits the customer needs.

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, 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 at specific deadlines.

Furthermore, the new requirements *leaded* the system (and especially the technical concepts, like topics and subtopics) into new directions while the system was under development, which ended up in the problem that the current version of the system includes already deprecated parts of code and concepts. An example for this evolution is the fact that we introduced topic-blocks in the middle of the development process, which can be edited in an autonomous ways. These topic-blocks may contain for example introductions, summaries, etc. Because the semantical line between these topic-blocks and subtopics is very fuzzy, we may need to remove the concepts of subtopics within future versions of the HiP application. On a technical side, topic blocks and subtopics are already handled exactly the same. But more details about the concrete development, will be given in the next section.

6.2.2 The development of the application

On the technical 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 for 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 within the final

product 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 in the case that code 'at the core of the system' (i.e., code parts that are used a lot by other functions) 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, which 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.

Last but not least, the main problem within the complete developing phase was that we could not use the Metaio eco-system for the AR-data as we had expected. This leaded to a couple of time-shifts, workarounds and not fulfilled requirements (i.e., the creation of the 3D-modeling editor).

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.

With a lot of feedback within the biweekly meetings this goal has been achieved and the application is able to handle even complex editing, reviewing and organization of information. Supervisors are able to create new groups, assign topics to them and edit specific constraints to create an easy filtering process for the work of the students. The students are able to work on these topics, communicate in a couple of ways, add/modify meta-data and review there

own work. If a topic is finished, it can be 'published' within the frontend by a master user. Admins are needed to create new data-types, change user-rights and edit language keys.

So, all in all, we achieved a lot more, as it has been specified within the first draft respectively within the first requirements engineering meeting that was used to create the story cards.

6.3.2 Future work

Although we have implemented a lot of functions within the backend, there is a lot more functionality open within our Scrum backlog. In general, future work can be done on three major areas:

- The smartphone application. As we have seen, one view of this app gets currently developed by a bachelor student within his bachelor thesis. However, there are a lot open requirements that need to be implemented and has sketched within this thesis.
- The remaining functions of the backend.
- The shift to a *DevOps ready* developing environment.

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 as it has been explained within this thesis. This will be a major improvement for the delivery speed and reduce the needed effort within every release. Such an improvement is especially important for a small project group, which tends to lack of free time. Another possible improvement would be the implementation of some of the roughly described patterns for DevOps architectures (Cukier (2013)).

Furthermore, the UI of the current system is able to handle a couple of seminars with students and supervisors at the same time. Nevertheless, one should think about more ways to scale the system in a way to be able to handle much bigger groups of students and supervisors, without loosing to much usability. Another important part would include the fixing of the remaining usability problems, that are open from the usability expert review by Björn Senft.

A

APPENDIX

The appendix contains 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 Node Package Manager (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 21 shows the work-flow within the backend system as a flow diagram.

Furthermore, Figure 22 shows an UML2 use case diagram showing the different users (i.e., user roles) of the application and Figure 23 shows a more technical view on the deployment situation of the system.

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

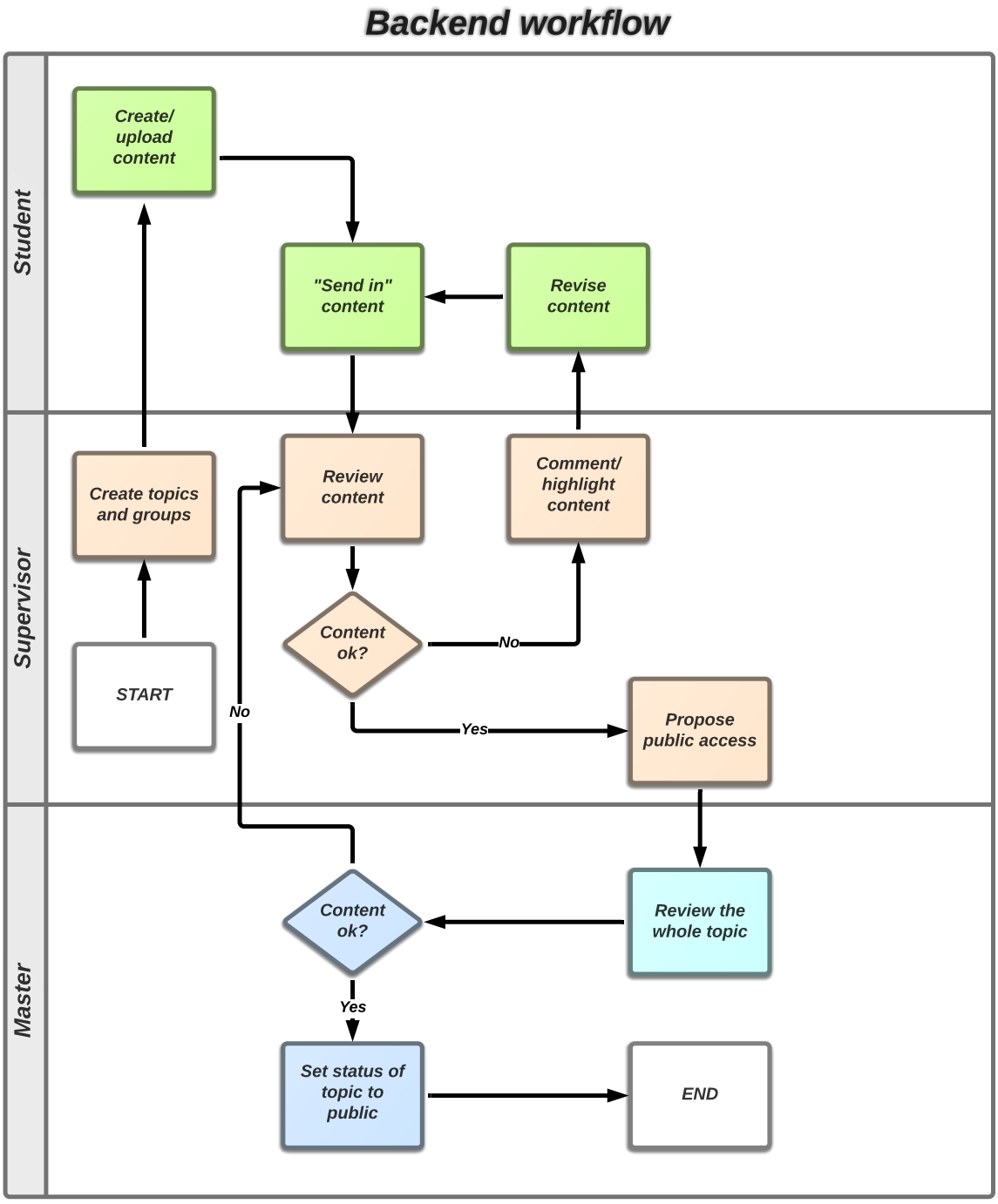

Figure 21: The diagram shows the work-flow within the backend system with the three roles that are involved in the work-flow.

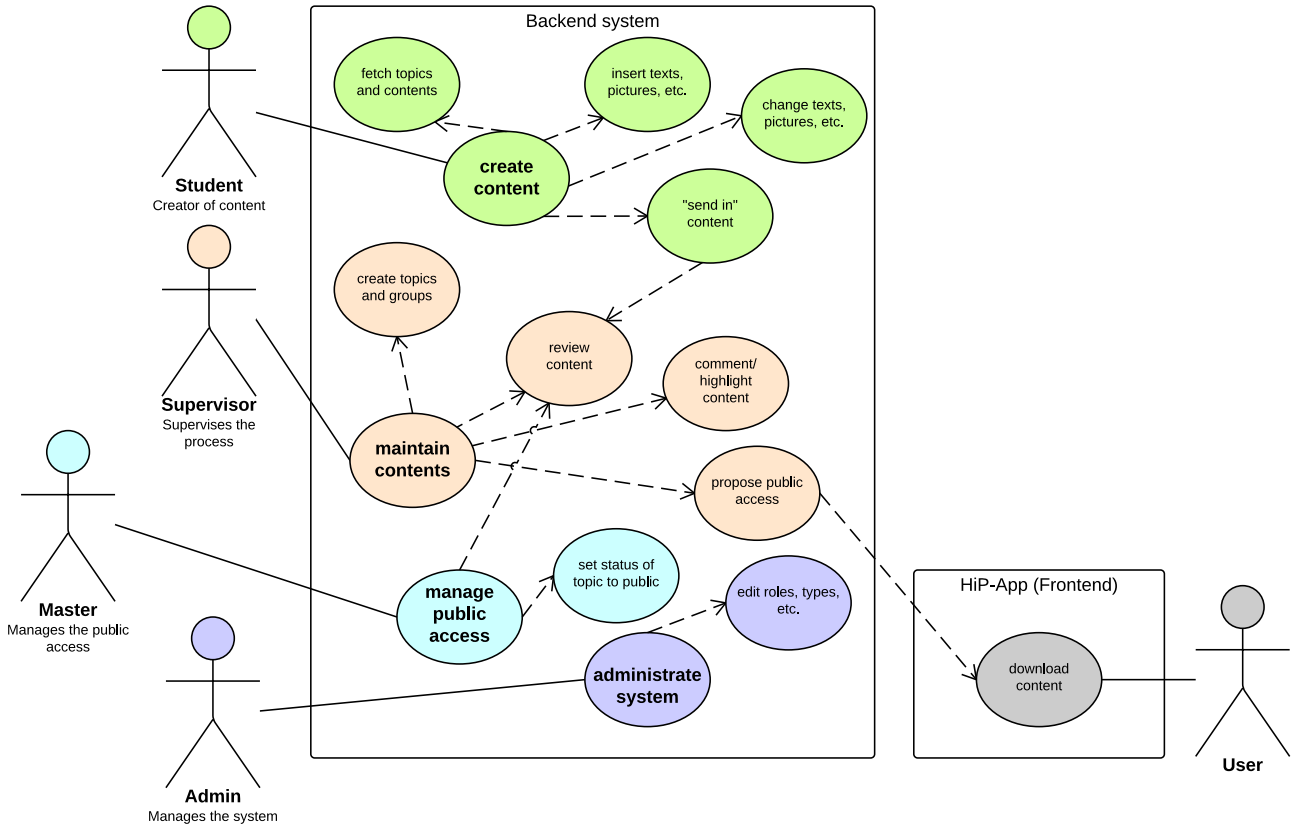

Figure 22: An UML2 use case diagram showing the different users of the HiP system.

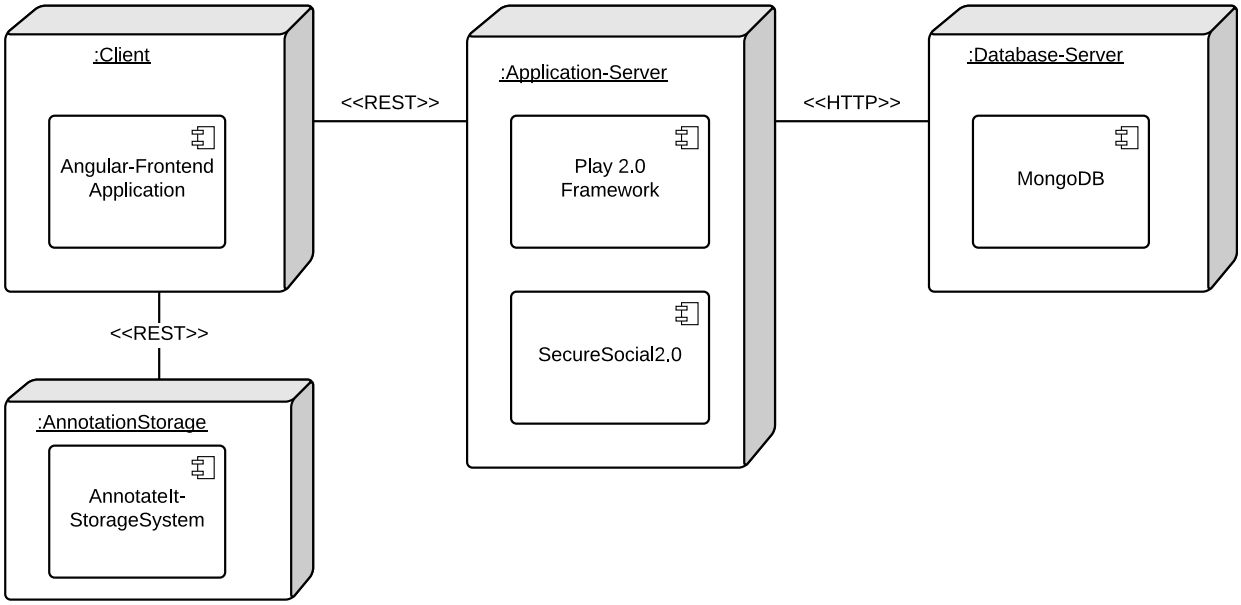

Figure 23: An UML deployment diagram showing the different parts of the HiP application.

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

Table 10: The found usability problems within the walktrough.

ID	Description	Solved within last version	Priority
UW01	Use better icons (e.g., flaticons.com)	Yes	1
UW02	Make tabs better recognizable	Yes	1
UW03	Use <a> with href to force other mouse cursor	Yes	1
UW04	Create a new way to show topics/subtopics	No	1
UW05	Be able to include group members to existing groups	Yes	1
UW06	Include imprint	Yes	1
UW07	Startpage: Use a better position to the login button	Yes	1
UW08	Startpage: Describe system better	Yes	1
UW09	Groups: Exchange *Welcome to the group*	Yes	1
UW10	Groups: Request confirmation on delete	Yes	1
UW11	Groups: Use light signs to show status	Yes	1
UW12	Groups: Show content as tree-structures	No	1
UW13	Groups: Add user-feedback to the dialogue	Yes	1
UW14	Chat: Improve the whole chat system (i.e., make it global)	No	1
UW15	Topic creation: Add free-text search	Yes	1
UW16	Topic creation: Show subtopics in tree-structures	No	1
UW17	Topic creation: Add status *hidden*	Yes	2
UW18	Topic modification: Use groups as first filter	No	1
UW19	Group creation: Show subtopics in tree-structures	No	1

Continued on next page

Table 10 – continued from previous page

ID	Description	Solved within last version	Priority
UW20	Group creation: Use simpler date-picker	No	2
UW21	Group-box: Remove refresh button	Yes	2
UW22	Template-box: Do not close menu by clicking on the <*div*>	Yes	2
UW23	Template-box: Exchange the way the pulldown-button works	Yes	2
UW24	Template-box: Use a scroll-view for the preview mode	No	3
UW25	Browse: Remove button	Yes	3
UW26	Notification-box: Exchange icon for filter	Yes	2
UW26	Notification-box: Show data more tabular	No	2
UW27	Navigation: Remove Home-button	Yes	2
UW28	Navigation: Show the currently logged-in user	Yes	2
UW29	Mail-system: Show user-feedback for sent-messages	Yes	2
UW30	Mail-system: Add outbox	Yes	2
UW31	Mail-system: Show usernames and not mail-addresses	Yes	2
UW32	Mail-system: Add e-mail notification for in-system messages	No	2
UW33	Contact form: Add contact form	No	3
UW34	Propose topics: More help-text	Yes	2
UW35	Propose topics: Wrong label on button	Yes	2
UW36	Topic edit: Exchange full-screen logo	Yes	3
UW37	Topic edit: Edit-frame is buggy	No	3
UW38	Topic edit: Automatically show new pictures	Yes	2

Continued on next page

Table 10 – continued from previous page

ID	Description	Solved within last version	Priority
UW39	Topic edit: Exchange editor-plugin	No	3
UW40	Topic edit: Do not switch view on map by click on the map	Yes	2
UW41	Topic edit: Make Button labels better understandable	No	2
UW42	Topic edit: Show what is located at a given GPS position	No	2
UW43	Topic edit: Add autocomplete for address field (GPS)	No	2
UW44	Topic edit: Change position of footnote-box with respect to locality	Yes	2
UW45	Topic edit: Do not use red-colors on buttons	Yes	2
UW46	Topic edit: Add delete-mode for tags to prevent deleting by accident	Yes	2
UW47	Topic edit: Order of history is buggy	No	2

Listing A.1: Complete upload Action of the FileController

```
def upload(topicID: String) = Action(parse.multipartFormData) {
    request =>
    request.body.file("file") match {
      case Some(photo) =>
        val TARGET_W = 64; // width of the thumbnail
        val TARGET_H = 64; // height of the thumbnail

        val filename = photo.filename
        val contentType = photo.contentType

        val newFile = new File("/tmp/picture/uploaded")

        if (newFile.exists())
          newFile.delete()

        photo.ref.moveTo(newFile)

        val gridFS = new GridFS(db, "media")
        val fileToSave = DefaultFileToSave(filename,
            contentType)

        /* create thumbnail */
        val fileToSaveThumb = DefaultFileToSave("thumb_"+
            filename, contentType)

        val os = new ByteArrayOutputStream()

        /* load image for scaling operation (needed to derive
            thumbnail) */
        var before = ImageIO.read(newFile)

        /* create scale operation */
        val wScale  = TARGET_W / before.getWidth().asInstanceOf
            [Double]
        val hScale  = TARGET_H / before.getHeight().
            asInstanceOf[Double]

        var at = new AffineTransform()
        at.scale(wScale, hScale)
        var scaleOp = new AffineTransformOp(at,
            AffineTransformOp.TYPE_BILINEAR)

        /* create object that will contain the scaled image */
        val w = (before.getWidth() * wScale).asInstanceOf[Int]
        val h = (before.getHeight() * hScale).asInstanceOf[Int]
        var after = new BufferedImage(w, h, BufferedImage.
            TYPE_INT_ARGB)

```

```
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. (2013). User stories. Website, http://guide.agilealliance.org/guide/user-stories.html, last checked 07.03.2015. (Cited on page 10.)

Alliance, A. (2015). Guide to agile practices. Website, http://guide.agilealliance.org/guide/tdd.html, 14.02.2015. (Cited on pages 22 and 23.)

AngularJS (2015a). Comprehensive directive api. Website, https://docs.angularjs.org/api/ng/service/$compile#directive-definition-object, last checked 18.02.2015. (Cited on page 60.)

AngularJS (2015b). karma-runner/karma. Website, https://github.com/karma-runner/karma, 14.02.2015. (Cited on page 23.)

Apple (2014). Getting started with ibeacon. PDF, https://developer.apple.com/ibeacon/Getting-Started-with-iBeacon.pdf, last checked 25.03.2015. (Cited on page 47.)

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. (Cited on page 2.)

Beck, K. (1999). Embracing change with extreme programming. *Computer*, 32(10):70–77. (Cited on page 23.)

Beck, K. (2003). *Test-driven development: by example*. Addison-Wesley Professional. (Cited on page 77.)

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. (Cited on pages 6 and 8.)

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. (Cited on page 6.)

Bevan, N. (1997). Quality and usability: a new framework. *Achieving software product quality, van Veenendaal, E, and McMullan, J (eds)*. (Cited on page 72.)

Bondo, J., Barnard, D., and Burcow, D. (2010). *iPhone User Interface Design Projects*. Apress. (Cited on page 2.)

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. (Cited on page 8.)

Cohn, M. (2004). *User stories applied*. Addison-Wesley Professional. (Cited on page 10.)

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. (Cited on pages 11 and 78.)

Danube (2014). Scrum effort estimation and story points. Website, http://scrummethodology.com/scrum-effort-estimation-and-story-points/, 12.02.2015. (Cited on page 10.)

Deissenboeck, F., Juergens, E., Lochmann, K., and Wagner, S. (2009). Software quality models: Purposes, usage scenarios and requirements. In *Software Quality, 2009. WOSQ09. ICSE Workshop*, pages 9–14. IEEE. (Cited on page 69.)

Dev.metaio.com (2015). Overview | metaio developer portal. (Cited on pages ix and 20.)

Duvall, P. (2012). Breaking down barriers and reducing cycle times with devops and continuous delivery. (Cited on page 11.)

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. (Cited on page 33.)

Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, USA. AAI9980887. (Cited on page 15.)

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. (Cited on page 22.)

Google (2014). Google maps javascript api version 3. Website, https://developers.google.com/maps/documentation/javascript/directions, 14.02.2015. (Cited on page 45.)

Google (2015). Polyline. Website, https://developer.android.com/reference/com/google/android/gms/maps/model/Polyline.html, 14.02.2015. (Cited on page 46.)

Hampel, T. (2001). *Virtuelle Wissensräume*. PhD thesis, Universität Paderborn. (Cited on page 2.)

Herman, L. (1996). Towards effective usability evaluation in asia: Cross-cultural differences. In *Computer-Human Interaction, 1996. Proceedings., Sixth Australian Conference on*, pages 135–136. IEEE. (Cited on page 73.)

IEEE (1990). Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84. (Cited on page 27.)

ISO/IEC (2001). *ISO/IEC 9126. Software engineering – Product quality*. ISO/IEC. (Cited on page 70.)

iText Software Corp (2015). itext core functionality. Website, http://www.itextpdf.com/functionality, 14.02.2015. (Cited on page 49.)

Janzen, D. S. and Saiedian, H. (2005). Test-driven development: Concepts, taxonomy, and future direction. *Computer Science and Software Engineering*, page 33. (Cited on page 22.)

Jones, C. (2003). Why flawed software projects are not cancelled in time. *Cutter IT Journal*, 16(12):12–17. (Cited on page 8.)

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. (Cited on page 20.)

Jung, H.-W., Kim, S.-G., and Chung, C.-S. (2004). Measuring software product quality: A survey of iso/iec 9126. *IEEE software*, 21(5):88–92. (Cited on page 70.)

Kaufmann, T. (2014). Bluetooth low energy ibeacon ist mehr als ein leuchtfeuer. *Golem*. (Cited on page 47.)

Keaveney, S. Conboy, K. (2011). *Cost Estimation in Agile Development Projects*. National University of Ireland. (Cited on page 7.)

Lidwell, W., Holden, K., and Butler, J. (2010). *Universal Principles of Design, Revised and Updated: 125 Ways to Enhance Usability, Influence Perception, Increase Appeal, Make Better Design Decisions,*. Rockport Publishers. (Cited on page 19.)

Lin, H. X., Choong, Y.-Y., and Salvendy, G. (1997). A proposed index of usability: a method for comparing the relative usability of different software systems. *Behaviour & information technology*, 16(4-5):267–277. (Cited on page 72.)

Mast, R. (2013). Architektur in agilen teams: Was softwarearchitekten von jazzmusikern lernen können. *OBJEKTspektrum*. (Cited on pages 33 and 34.)

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. (Cited on pages 7 and 22.)

Maxxor (2015). Software development process | maxxor. Website, https://www.maxxor.com/software-development-process, last checked 14.02.2015. (Cited on pages ix and 8.)

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. (Cited on page 17.)

Microsoft (2009). *Microsoft Application Architecture Guide, 2nd Edition*. Microsoft Press book. (Cited on page 33.)

MongoDB-Inc. (2015). Gridfs. Website, http://docs.mongodb.org/manual/core/gridfs/, last checked 16.02.2015. (Cited on page 57.)

Mueller, E., Wickett, J., and Gaekwad, K. (2011). What is devops? Website, http://theagileadmin.com/what-is-devops/, last checked 21.02.2015. (Cited on page 11.)

Nerur, S., Mahapatra, R., and Mangalaraj, G. (2005). Challenges of migrating to agile methodologies. *Communications of the ACM*, 48(5):72–78. (Cited on page 22.)

North, D. (2012). Bdd is like tdd if... Website, http://dannorth.net/2012/05/31/bdd-is-like-tdd-if/, 14.02.2015. (Cited on page 23.)

OpenSource (2012). Osmand. Website, https://code.google.com/p/osmand/, 14.02.2015. (Cited on page 47.)

Paulk, M. C. (2002). Agile methodologies and process discipline. *Institute for Software Research*, page 3. (Cited on page 7.)

Peterson, K. (2015). Angularjs: One-time vs two-way data binding. Website, http, last checked 16.04.2015. (Cited on page 18.)

Pichler, R. (2010). 10 tips for writing good user stories. Website,http://www.romanpichler.com/blog/10-tips-writing-good-user-stories/ ,last checked 17.03.2015. (Cited on page 10.)

RadiusNetworks (2015a). Android beacon library. Website, http://altbeacon.github.io/android-beacon-library/, last checked 25.03.2015. (Cited on page 47.)

RadiusNetworks (2015b). Reference application. Website, http://altbeacon.github.io/android-beacon-library/samples.html, last checked 25.03.2015. (Cited on page 48.)

Rodriguez, A. (2008). Restful web services: The basics. *Online article in IBM DeveloperWorks Technical Library*, 36. (Cited on page 15.)

Schwaber, K. and Beedle, M. (2002). *Agile Software Development with Scrum*. Pearson. (Cited on page 7.)

Shamsuddin, N. A., Syed-Mohamad, S. M., and Sulaiman, S. (2014). Capturing users' actions in a web application to support learnability. In *Software Engineering Conference (MySEC), 2014 8th Malaysian*, pages 142–147. IEEE. (Cited on page 73.)

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. (Cited on page 1.)

Sutherland, J. Jakobsen, C. (2009). Scrum and cmmi – going from good to great. *Agile Conference*. (Cited on page 7.)

Trelle, T. (2014). Mongodb. *JavaMagazine*, 5:56–62. (Cited on page 15.)

Vogel, L. (2010). Creating pdf with java and itext. Website, http://www.vogella.com/tutorials/JavaPDF/article.html, 14.02.2015. (Cited on page 49.)

Waghmode, M. and Jamsandekar, P. (2013). Software quality models: A comparative study. *Information and Software Technology*. (Cited on page 69.)

Wattson, M. (2013). Continuous delivery for winners – with a feedback loop. Website, http://stackify.com/wp-content/uploads/2013/08/continuous-delivery.png, last checked 16.02.2015. (Cited on pages ix and 12.)

Wikimedia-Foundation (2014). About us. Website, https://www.wikimedia.de/wiki/%C3%9Cber_uns, last checked 12.09.2014. (Cited on page 1.)

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. (Cited on page 22.)

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, April 17, 2015

Jörg Amelunxen