



Department of Informatics
Technical University of Munich



TECHNICAL UNIVERSITY OF MUNICH

DEPARTMENT OF INFORMATICS

MASTER'S THESIS IN AUTOMOTIVE SOFTWARE ENGINEERING

**A Trustworthy Lifecycle Management System for Student
Research Projects**

Thomas Maurer

TECHNICAL UNIVERSITY OF MUNICH
DEPARTMENT OF INFORMATICS

Master's Thesis in Automotive Software Engineering

**A Trustworthy Lifecycle Management System
for Student Research Projects**
**Ein vertrauenswürdiges Management System für
den Lebenszyklus studentischer
Forschungsarbeiten**

Author:	Thomas Mauerer
Supervisor:	Prof. Dr.-Ing. Georg Carle
Advisor:	Dr. Holger Kinkelin Marcel von Maltitz, M.Sc.
Date:	June 15, 2018

I confirm that this Master's Thesis is my own work and I have documented all sources and material used.

Garching, June 15, 2018

Location, Date

Signature

ABSTRACT

For a chair, offering, performing, grading and finally filing student theses is associated with a considerable organizational effort. That is mainly due to the high number of different tasks and parties that are all involved into each single thesis and that need to work together in a coordinated way and synchronized in time. To support all involved stakeholders, it would be helpful to have a tool that automates as many tasks - related to the management of theses - as possible and assures that no tasks are left behind. Beside of the functional aspects that are covered by such tool, it is required to focus on non-functional requirements like privacy and security, since that tool has to deal with critical data like final grades and personal information.

In this Master's Thesis we describe the development of **Chairman** which is our solution for the demanded tool. We go through the entire process of software engineering. We start by analyzing the functional and non-functional requirements as accurately as possible, develop a suitable solution in theory and implement the software product afterwards. Finally, we evaluate Chairman and compare it with similar systems.

The first contribution of this Master's Thesis is a working version of the Chairman management system. The solution features a carefully designed software architecture which is essential to fulfill the requirements, especially the non-functional requirements. The architecture is based on *microservices* which let us physically separate critical from non-critical data. This is useful for creating a privacy-preserving system that provides data protection even in cases when an attacker has got access to the database. Apart from that, microservices help on structuring the entire application in a way that allows easy extensibility and maintainability.

Our second contribution is an accountability mechanism that shall prevent data misuse and show who is responsible for several actions. The mechanism extracts actions from the individual Chairman microservices, assigns these actions to responsible persons and generates a stream of events in a well-defined format. Currently, our accountability mechanism uses a *MongoDB* database for persisting these Chairman events. However, we show how increased requirements concerning non-modifiability and non-erasability of these accounting information can be reached by connecting our accounting mechanism to a blockchain-based backend.

ZUSAMMENFASSUNG

Für Lehrstühle an der Universität stellt es einen beträchtlichen organisatorischen Aufwand dar, Studienarbeiten auszuschreiben, durchzuführen, zu benoten und anschließend einzureichen. Hauptsächlich liegt dies daran, dass Studienarbeiten aus vielen einzelnen Aufgaben bestehen und mehrere unterschiedliche Personen beteiligt sind, welche alle aufeinander abgestimmt werden müssen. Um das Management zu vereinfachen, wäre es hilfreich, ein entsprechendes Tool zu haben, das erstens so viele Aufgaben wie möglich automatisiert und zweitens gewährleistet, dass keine Aufgaben vergessen werden. Neben den funktionalen Aspekten eines solchen Tools, muss ein spezieller Fokus auf nicht-funktionale Anforderungen wie Datenschutz und Angriffssicherheit gelegt werden, da das Tool mit vertrauensvollen Daten umgehen muss. Beispiele für vertrauensvolle Daten sind Abschlussnoten und persönliche Daten.

Diese Masterarbeit beschäftigt sich mit der Entwicklung eines solchen Managementsystems namens **Chairman**, indem einmal der komplette Zyklus des *Software Engineerings* durchlaufen wird. Als erstes werden konkrete Anforderungen hergeleitet. Danach wird das System in der Theorie entwickelt und entworfen und anschließend in der Praxis umgesetzt. Zum Schluss wird ausgewertet, ob bzw. wie gut Chairman die Anforderungen erfüllt. Zudem wird es mit ähnlichen, bestehenden Systemen verglichen.

Der erste Beitrag dieser Masterarbeit ist eine lauffähige Version von Chairman. Das entwickelte System hat eine intelligent entworfene Softwarearchitektur, welche die Erfüllung der meisten Anforderungen, insbesondere der nicht-funktionalen ermöglicht. Konkret basiert die Architektur auf *Microservices*. Dadurch lässt sich eine saubere Trennung zwischen kritischen und nicht-kritischen Daten erreichen. Dies ist notwendig, um Datenschutz sogar gewährleisten zu können, falls ein Angreifer Zugang zur Datenbank hat. Des Weiteren sind Microservices hilfreich zur Strukturierung des Gesamtsystems, wodurch Erweiterbarkeit und Wartbarkeit des Systems deutlich verbessert werden.

Der zweite Beitrag dieser Masterarbeit ist die Entwicklung eines Mechanismus zum Aufzeigen von Verantwortlichkeiten und zum Verhindern von Datenmissbrauch. Im Englischen wird hierfür der Begriff *Accountability* verwendet. Der Mechanismus zeichnet getätigte Nutzeraktionen auf, stellt einen konkreten Bezug zum entsprechenden Nutzer her und speichert diese Informationen in einem zuvor definierten Format. Aktuell wird eine *MongoDB* Datenbank zur Speicherung der Informationen verwendet. Allerdings wird auch gezeigt, dass man mit Hilfe einer Blockchain höheren Anforderungen bzgl. Unveränderbarkeit und Nicht-Löschbarkeit dieser Informationen gerecht werden kann.

CONTENTS

1	Introduction	1
1.1	Motivation	1
1.2	Approach & State of the Art	1
1.3	Goals & Research Questions	2
1.4	Methodology & Structure	4
2	Background	5
2.1	Privacy & Security	5
2.2	Technologies	8
2.2.1	OAuth 2.0	8
2.2.2	Hyperledger Fabric	9
2.3	Frameworks	11
2.3.1	Spring Boot	11
2.3.2	Angular	12
3	Requirements Engineering	13
3.1	Functional Requirements	13
3.1.1	Stakeholders in Chairman	13
3.1.2	Lifecycle of a Thesis	19
3.2	Non-Functional Requirements	22
4	Analysis	25
4.1	Analysis of the Software Architecture	25
4.1.1	Terminology	25
4.1.2	First Approach (Monolithic)	26
4.1.3	Refinement (Microservices)	29
4.2	Authorization with OAuth 2.0	32
4.3	Accountability Analysis	34
4.3.1	Definition of Chairman Events	34
4.3.2	Processing of Chairman Events	36

4.3.3	Database Solution	37
4.3.4	Blockchain-Based Solution	38
5	Design	41
5.1	Chairman Design	41
5.1.1	Microservices	41
5.1.2	Technologies	45
5.2	Graphical Design	46
6	Implementation	47
6.1	Overview	47
6.2	Implementation Details	49
6.2.1	Microservices	49
6.2.2	Angular SPA	60
6.3	Deployment	61
7	Evaluation	63
7.1	Implementation of the Functional Requirements	63
7.2	Observance of the Non-Functional Requirements	64
7.2.1	Privacy, Security & Accountability	64
7.2.2	Flexibility, Extensibility & Maintainability	66
7.2.3	Usability	66
7.2.4	Performance & Availability	67
8	Related Work	69
8.1	Architectural Relation: Netflix	69
8.2	Functional Relation	70
8.2.1	Alfresco	70
8.2.2	Campus Online	72
9	Conclusion	73
A	Abbreviations	75
B	Graphical Mockups	77
B.1	Student Mockups	77
B.2	Staff Member Mockups	80
B.3	Professor & Secretary Mockups	83
	Bibliography	87

LIST OF FIGURES

3.1	Lifecycle of a Student Thesis	20
4.1	Monolithic Architecture	27
4.2	Microservices Architecture	31
4.3	OAuth 2.0 in Chairman	33
4.4	Chairman Event	35
4.5	Monitoring Service in Chairman	36
4.6	Blockchain Infrastructure	38

LIST OF TABLES

2.1	Security Patterns	7
2.2	Transaction Flow in Hyperledger Fabric	10
3.1	Functional Requirements of Chairman	21
3.2	Relevance of the Functional Requirements	21
3.3	Non-Functional Requirements of Chairman	24
4.1	Authorization Code Grant Type Protocol	33
4.2	Actions in Chairman	35
5.1	Dependency Structure Matrix of the Microservices	44
5.2	All Microservices of Chairman	45
6.1	Spring Boot Applications Lookup Table	48

CHAPTER 1

INTRODUCTION

1.1 MOTIVATION

Every year, a high number of students write their final papers (i.e. Bachelor/Master theses) at universities. For instance, only at the chair of *Network Architectures and Services* the number is already about 100 student theses per year. Apart from the students themselves, there are many parties involved in a thesis and the organizational management is associated with a high effort. That is mainly due to the high number of different tasks that all need to be done in order to finish a thesis successfully. For example, staff members need to write and publish theses descriptions, the secretary needs to take care of registering a student, deadlines need to be observed, etc. Furthermore, students need to be able to download templates and publish their final results at the chair's website. For a smooth running, it is required that all involved parties work together in a coordinated way and synchronized in time.

1.2 APPROACH & STATE OF THE ART

In order to handle the complexity of managing student theses, it would be helpful to have some kind of tool that basically does three things: Firstly, it should automate as many tasks as possible because that would drastically reduce the workload of the individual parties. Secondly, it should centralize the management of theses. This would simplify the cooperation of all involved parties, so that they can work together in a synchronized way. Thirdly, it should be able to guarantee that no tasks are left behind.

Dr. Holger Kinkelin has already implemented a prototype - called **Chairman** - which is running for almost three years now. Chairman is written in *Python* and is based on a central subversion repository to which all involved parties have access. Due to privacy reasons, each student is assigned a separated folder. Checking out other paths than that specific folder is only possible for students if it has been explicitly allowed. Several jobs, like e.g. sending emails, are already automated in Chairman.

A positive side effect of subversion is its accountability mechanism. That is because subversion stores a version history which gives information about *what changes* were made by *which person* at *which time*. This is a useful feature and is for example required to guarantee transparency in the grading. If the final grade differs from the grading proposal, it shall be accountable where the difference has come from.

However, since it is only a prototype, the current version has also many drawbacks. The biggest problem is that Chairman is not able to guarantee that no tasks are left behind because it does not cover the entire lifecycle of a thesis. The result is that use cases and tasks respectively are missed. The complete registration process, which requires to fill out registration documents and contracts, is for instance not included in Chairman at all.

Secondly, there are some technical and general issues in the concrete implementation of Chairman. One problem is subversion itself. The documentation of subversion states that it is a "*common misuse to treat [subversion] as a generic distribution system*" [42] which is basically its main job in Chairman. Apart from that, most students are familiar with *git*, but not really with *svn* and many people have complained about the stability of subversion under Linux. Another problem is that Chairman does not utilize any database, but only stores flat files. This quickly becomes disorganized and also limits the possibilities in some cases. Probably because there is no graphically appealing user interface, some parts of Chairman are used only rarely by students and staff members. An example of that is tracking the steps that are required for finishing a thesis successfully, which currently requires to manually update a text file and commit the changes.

1.3 GOALS & RESEARCH QUESTIONS

We see that Chairman has been a good approach, but there are several improvements required in order to make it a real management system for student theses. However, instead of trying to enhance the current version, we start from scratch and focus on scientifically sound methodologies this time. The ultimate goal is to develop a stable

and easily usable application that covers the entire lifecycle of a thesis, helps the stakeholders in all situations and is completely platform independent. As a side effect of this, Chairman acts as a red thread that tells the stakeholders which tasks need to be done in order to finish a thesis successfully.

To achieve all these goals, it is not appropriate to start implementing directly, but rather go through the entire process of *Software Engineering* and have a scientific foundation. For this purpose, we want to explicitly answer the following scientific questions beside of developing the actual system.

1. **RQ1:** How to make Chairman secure and privacy preserving?

Security and privacy are the main focus of this Master's Thesis. However, since security and privacy are often contradicting to usability, we investigate this research question only up to a point that is practically feasible. One should not forget that the ultimate goal is to develop a system that is usable in production. At least, we want to achieve that it is impossible for unauthorized persons to gain access to private data or change it. This means that Chairman needs to clearly regulate authorizations of different stakeholders and guarantee that each stakeholder can only do what they are allowed to.

2. **RQ2:** How to address accountability in Chairman?

Since subversion is no longer used, we need to find another way to address accountability in Chairman. Independently on how this mechanism will look like, it must be guaranteed that all critical actions can be assigned to a concrete user and it is not possible to deny an action.

3. **RQ3:** How to make Chairman flexible, extensible and maintainable?

At the moment, there are only a few stakeholders known. As more stakeholders or at least use cases might be coming in the future, Chairman has to be implemented in a flexible way that allows easy extensibility. Doing maintenance work must be possible even for fellow students that are not familiar with all the internal details of Chairman.

These research questions are continued and extended with more details in connection with the non-functional requirements of Chairman in Section 3.2.

1.4 METHODOLOGY & STRUCTURE

According to the way Chairman is developed, the main part of this thesis follows the **Waterfall** Software Lifecycle Model. Even though this model is often described as outdated and superseded by agile methodologies, we still use it for the development of Chairman. This is because agile models usually require a team, whereas Chairman is basically a one-person project.

In the literature one can find many different variants of the Waterfall model. Some variants have five, whereas others have six phases. Winston W. Royce, who is often considered to be the founder of the Waterfall model [25], even had seven phases in the original draft [37]. However, basically they are all the same: all models start with defining the **requirements** of the software/system, followed by the **analysis** and **design** phase. Afterwards, the software is **implemented**, **tested** and **maintained** [37].

Requirements are described in Chapter 3. By analyzing the different stakeholders of Chairman and the lifecycle of a thesis, we can derive functional and non-functional requirements.

The analysis phase is part of Chapter 4. In this chapter we develop a solution that is able to fulfill the requirements. We analyze different possibilities and stay as abstract as possible in this phase.

Chapter 5 describes the design phase. Here, we become concrete and define exactly how the solution needs to look like.

Chapter 6 describes the practical part of this Master's Thesis and documents what has been implemented.

Testing and maintaining have not received their own chapters, but are part of the evaluation which is described in Chapter 7.

Before the main part starts, Chapter 2 has been inserted which describes necessary background information.

Finally, there is a chapter about related projects (Chapter 8) and the conclusion of this Master's Thesis (Chapter 9).

CHAPTER 2

BACKGROUND

This chapter provides background information that is required for following the discussions in the later parts of this Master's Thesis.

2.1 PRIVACY & SECURITY

Privacy and *security* are both terms with no clear definition. Article 7 of the European Charter of Fundamental Rights states that everyone "*has the right to respect for his or her private and family life, home and communications*" [19]. In our opinion, this article is a good starting point for defining privacy in general, but is rather useless when dealing with a data management system like Chairman. Article 8, which adds the "*right to the protection of personal data*" [19], is better suited. In addition to that, there is the General Data Protection Regulation (GDPR) which has come into effect in Europe on May 25, 2018 [22]. Article 5 of the GDPR also uses the term *personal data* and regulates principles related to processing of that data [22]. Which kind of data is meant by that term is, however, not defined accurately in both documents. *Personally identifiable information*, like name, age, address, etc. are often seen as data that requires protection [17]. However, this is not enough in our estimation: the final grade of a student, for instance, is something that does not identify a person, but still requires protection. For this reason, we rather talk about *privacy-relevant data* or *critical data* in this Master's Thesis.

Security is often named in the same context as privacy, but it is important to keep the two terms separated. Security concerns preventing attacks and is in fact more far-reaching than privacy: indeed, security mechanisms should be used to make a system privacy-preserving (see GDPR Art. 5, Par. 1 (f) [22]), but security mechanisms can also have the objective to prevent attacks not aimed at stealing or changing critical data.

Over the years, many different principles and standards have been developed that should be implemented in order to make a system secure and privacy-preserving. To target privacy, one should at least implement the so-called *Privacy Protection Goals* which are **unlinkability**, **transparency** and **intervenability** [36].

Unlinkability means that it must not be possible to link privacy-relevant data of one domain with data of a different domain [17]. First of all, this can be achieved by avoiding critical data at all. This means that critical data should only be collected if really necessary and should be deleted afterwards [17]. Legally, this is confirmed by GDPR Art. 5, Par. 1 (b,c) [22]. Secondly, by separation of contexts which could be a physical separation or at least the usage of different identifiers [17]. Last but not least, by anonymizing critical data [17] (see GDPR Art. 5, Par. 1 (e) [22]).

Transparency is a prerequisite for making a system verifiable and allow a user to understand *why* and *when* critical data is processed [36]. This is achieved with clear documentations and explanations [17]. Even the source code of an application could be made open source to allow verification [17]. Apart from that, logging and reporting can help to make a system transparent [17].

Intervenability guarantees that users can always intervene processing of critical data [17]. This is achieved by giving users the possibility to delete personal data of themselves, make corrective measures or stop data processing at all [17]. Changing specific settings must be easily possible and not be hidden somewhere in the application [17].

As the pendent to the *Privacy Protection Goals*, there are the well-known *Security Protection Goals* **confidentiality**, **integrity** and **availability** [36], which should be implemented in order to make a system secure. However, we think that these goals are too high-line and are not really helpful in practice. Yoder and Barcalow have come up with several security patterns which are way more tangible in our opinion [44]. The most important patterns (taken from [44]) are listed and explained in Table 2.1.

Making a system hundred percent secure and privacy-preserving is not achievable in practice. If one can show that at least the introduced protection goals and security patterns are implemented in a meaningful way, security and privacy are sufficiently covered.

Pattern Name	Explanation
Single Access Point	Entry to an application should be limited to a single point. This makes it easier to secure a system since all users and attackers respectively need to pass the single access point and have no other option to get into the system.
Check Point	Security policies may change from time to time. Hence, there should only be one part in an application that is responsible for checking the security policies. This makes it easier to react to changing policies. An example of such policy is which permission a normal user has in the application.
Roles	Securing a multi-user application is hard to achieve if all users have different permissions as to what they can do in the application. Hence, all users should be organized into groups. Security policies should then apply to an entire group, rather than to individual users.
Session	All parts of an application need to have a way to check the permissions of the currently active user. Hence, there should be a globally accessible object in which information about the user is stored. This object could be created by the check point and passed around to all parts when needed.
Limited View	Users should not see more than they have the permission to. The decision what a user can see should depend on the user's role which is stored in the session object.
Secure Access Layer	A system is only as secure as its weakest link. Hence, an application should be built around existing security mechanisms that are at best proven to be secure. When communicating over a network for example, one should use a TLS-secured connection.

TABLE 2.1: Security Patterns

2.2 TECHNOLOGIES

This section describes background information about technologies that are relevant for this Master’s Thesis. The first part is about an authorization framework called OAuth, whereas the second part is about a framework for private/permissioned blockchains.

2.2.1 OAUTH 2.0

OAuth 2.0 is a standardized authorization framework that mitigates a serious downside of a classical client-server authentication [32]. The problem of the classical approach is that a client needs to know the user credentials in order to access protected resources on the server [23]. While this is fine for clients that belong directly to the application, it is inadequate for third-party clients. That is because one should never share user credentials with a third-party since this means that the third-party is able to operate clients on their own without further control of the user. OAuth eliminates this disadvantage by introducing an access token that allows a client to access several resources on the server, while the user credentials are never shared with the client [23].

This is achieved by separating between the role of the human user and the client at the outset. A client can at most have the same authorizations as the human user, but never more. All in all, OAuth specifies the following roles [23]:

1. **Resource Owner:** The entity - usually a person - that grants access to protected resources. If the resource owner is a human, it is also called *end-user*.
2. **Resource Server:** The server that hosts the protected resources.
3. **Client:** The application that requests the protected resources with the resource owner’s authorization.
4. **Authorization Server:** The server that authenticates the resource owner, obtains authorization and issues access tokens to the client.

The way a client retrieves an access token from the authorization server can differ from use case to use case. The specification defines the following four *authorization grant* types [23]:

1. **Authorization Code:**
This is the most popular grant type. After authenticating themselves, a resource owner retrieves a *code* from the authorization server that is transferred to the client. The client then authenticates itself at the authorization server and uses the code in order to obtain an access token.

2. Implicit:

This is a simplified version of the *authorization code* type. Instead of retrieving a code from the authorization server, the resource owner directly issues an access token for the client. While this grant type is easier to achieve in many cases, it has the downside that the client does not have to authenticate itself at the authorization server.

3. Resource Owner Password Credentials:

In this grant type the client uses the resource owner's credentials directly in order to obtain an access token which means that the credentials have to be shared with the client. This grant type should only be used if the client belongs to the application and if there are no other options available.

4. Client Credentials:

In this grant type the client uses its own credentials in order to obtain an access token. This is used in situations where the client itself is the resource owner or when an arrangement with the authorization server and the resource owner has been previously made.

Examples of OAuth providers are *Facebook* and *Google*. Due to these providers, third party applications do not compellingly have to require a user to create an account. Instead, existing Facebook or Google accounts can be used directly. Another use case is to utilize OAuth in a distributed application where the individual components are stateless and do not share a session that can be used for storing the user information. With appropriate measures it is even possible to implement a Single Sign On mechanism in the distributed application.

2.2.2 HYPERLEDGER FABRIC

Hyperledger Fabric is an open source framework for operating *private* and *permissioned* blockchains. These kind of blockchains have the property that the participants are known and have a clear identity. Furthermore, only authorized entities are capable of writing data into the blockchain. The opposite are *public* blockchains which are for example known from Bitcoin and Ethereum. In this type of blockchains anybody can participate. [4]

A blockchain itself is a data structure that consists of blocks which are linked together to a chain through several hash values. It is not possible to manipulate or delete blocks without destroying the link to subsequent blocks. The blockchain technology requires a distributed network where the individual nodes store a copy of the blockchain. [12]

The way the nodes agree on a common order of blocks is called *consensus* algorithm [12]. Many public blockchains rely on the so-called **proof-of-work** algorithm which requires the nodes to solve a cryptographic puzzle [4]. In this point Hyperledger Fabric is advantageous since it uses a much more simple consensus algorithm. This basically works by having a special entity - called *orderer* - which specifies the order and broadcasts this information to all participants of the network [4].

Hyperledger Fabric defines three different roles for nodes in the network: clients, peers and orderer. Details about the roles and the transaction flow in general can be found in [4] or in the online documentation¹. Somewhat simplified, the transaction flow works as described in Table 2.2 [4]:

- (1) A client sends a *transaction proposal* to a subset of all peers, called *endorsers* or *endorsing peers*. The *endorsement policy* states which peers may act as endorsers and how many endorsements are required for the transaction to be accepted by the network.
- (2) The endorsing peers execute the transaction proposal, put the results² in a *proposal response* and send that proposal response back to the client. The results are called *endorsements* and are digitally signed by the peers with their private keys.
- (3) After the client has received enough endorsements, so that the endorsement policy is satisfied, it creates the *transaction*, which in turn includes the endorsements, and sends it to the orderer node.
- (4) The orderer node takes the received transactions of all clients and brings them into an arbitrary order. Multiple transactions are then packed together in a block which is broadcast to all peers in the network.
- (5) Finally, the peers validate the transactions inside the received block, and if valid, append that block to their local copy of the blockchain. In particular, these checks validate whether the endorsement policy is fulfilled.

TABLE 2.2: Transaction Flow in Hyperledger Fabric

¹<https://hyperledger-fabric.readthedocs.io/en/release-1.1/>

²readset and writeset

2.3 FRAMEWORKS

This section describes background information about frameworks which are used in the practical part of this Master's Thesis.

2.3.1 SPRING BOOT

Spring [40] in general is a Java framework which simplifies Java development in many situations and is quite popular for being used in web development. Spring Boot [39] in turn is a project within the Spring framework that enables faster results. This is for instance achieved by completely omitting *xml* configuration files. Instead, Spring Boot automatically configures projects based on annotations and import statements [39].

In contrast to classical Java web development, Spring Boot does not produce *war* files by default, but instead *fat jar* files which include all required dependencies. This is convenient because one can simply start applications by typing `java -jar FILE_NAME.jar` in the command line and does not have to tinker with downloading dependencies, setting up the execution environment and more. [39]

Spring Boot applications can be built with either *Maven* or *Gradle*. Apart from that, Spring Boot includes a command line tool which is helpful for prototyping or setting up new projects. [39]

The following listing (Listing 2.1) shows the famous *hello world* example implemented in Spring Boot. The `getIndex()` method is mapped to the path `/` due to the `@GetMapping` annotation. The returned string is directly rendered in the browser.

```

1 | import [...]
2 |
3 | @SpringBootApplication
4 | public class DemoApplication {
5 |     @RestController
6 |     public static class IndexController {
7 |         @GetMapping("/")
8 |         public String getIndex() {
9 |             return "Hello World";
10 |        }
11 |    }
12 |    public static void main(String[] args) {
13 |        SpringApplication.run(DemoApplication.class, args);
14 |    }
15 | }

```

LISTING 2.1: Hello World Example in Spring Boot

2.3.2 ANGULAR

Angular [5] is a web framework for developing frontends and client applications in *TypeScript* which is a programming language based on *JavaScript*. Angular is the successor of AngularJS and is developed by *Google Inc.* [7].

Angular enables a developer to build completely modular client applications by differentiating between *modules*, *components* and *services*. Due to *dependency injection*, one never has to instantiate services directly, but instead can simply let Angular inject the services whenever needed. [5]

Another useful tool is the Angular command line tool, or short *Angular CLI* [6]. It can be installed with *npm* and can be used in order to bootstrap Angular projects or generate Angular modules, components and services. Apart from that, it can build the application and run unit tests.

Angular is quite popular for **Single Page Applications** (SPA) which solve a problem of traditional web applications. That is the rather bad user experience of those traditional applications since nearly all user actions require a complete site to be loaded. In turn, this means that a user has to accept a high waiting period before the new site is rendered. SPAs try to overcome this problem by loading all resources directly at the start and only exchanging page components as needed. Usually, the initial request loads an *index.html* file together with some *JavaScript* and *CSS* bundles. The entire page is never reloaded. Only content, often in *json* format, is loaded afterwards. [27]

The advantage of this are short loading times which enable a fluid user experience. However, the initial request in turn lasts longer and search engines have problems with dynamically loaded contents. Hence, SPAs might have a worse page rank than traditional web applications. [27]

CHAPTER 3

REQUIREMENTS ENGINEERING

According to the Software Lifecycle Model, the first phase is about defining the requirements. The goal of this phase is the **Requirements Specification** which consists of both, functional and non-functional requirements. Initially, we have agreed on developing the new Chairman as a web application.

3.1 FUNCTIONAL REQUIREMENTS

We start this chapter with an analysis of the stakeholders and the thesis lifecycle in order to find out which functional requirements need to be addressed by the new Chairman. Non-functional requirements are analysed in Section 3.2.

3.1.1 STAKEHOLDERS IN CHAIRMAN

In this section we identify the stakeholders of Chairman and find out which tasks need to be done by the respective stakeholders. With this knowledge, we can then formulate the functional requirements (**FR**) of Chairman.

3.1.1.1 STUDENTS

The main stakeholders are the students who want to write their thesis at the chair. Basically, there are four tasks that need to be done by the students:

1. **Fill out contracts and registration documents**

Before a thesis can be written at all, a student needs to sign a contract with the chair. This contract regulates for instance that the chair is allowed to use and

utilize results of the thesis and that the final result can be published. Furthermore, a registration document needs to be handed in at the *Infopoint* in the case of a study program that is part of the department of Informatics.

2. Manage thesis

The main task is to work on the thesis and submit the final result before the deadline. There are several formalities that need to be observed when handing in a thesis. These are either formalities postulated by the *Technical University of Munich* itself, or by the chair. An example is the minimum length of a thesis or the way the title page needs to be formatted. For that reason, Latex templates should be used that fulfill the formal requirements.

3. Stay in contact with the advisors

It is required to have meetings with the advisors of the thesis on a regularly basis, e.g. once per week. In these meetings the student needs to show the progress that has been achieved since the last meeting. In return, advisors give feedback and help on finishing the thesis successfully.

4. Give talks and publish talk slides

This task differs from chair to chair. At least at the chair of *Network Architectures and Services*, it is required to give one talk in the midst of the thesis and one after the final submission. Apart from the professor, some students and members of the chair might be present at these talks. Furthermore, an introduction talk with the professor and the advisors must be held. For all of these talks, it is required to make an appointment first. Apart from that, students need to prepare presentation slides - using a mandatory template - and publish them afterwards.

According to the four tasks mentioned above, we can derive four functional requirements for Chairman:

- **FR 1: Provide downloads**

Chairman must enable a student to download several documents, like *How-To's*, *Guidelines* and Latex templates for talks and the final thesis. This way, it can be guaranteed that formal requirements are fulfilled and the student gets familiar with the additional formalities that need to be observed. Furthermore, it should be possible to download pre-filled contracts and registration documents, so that these documents do not have to be filled out by hand.

- **FR 2: Inspect progress, next steps and receive reminder mails**

Chairman must offer the possibility to inspect the current progress, the next steps and upcoming events, like talk dates. This ensures that there are no forgotten

tasks and a student is able to finish the thesis successfully. At minimum, this must be text-based, but a visual illustration should be preferred. Moreover, Chairman should send automatically generated emails reminding of deadlines and talk dates.

- **FR 3: Manage exchange between advisors and student**

It must be possible to capture decisions, like appointments which have been made between stakeholders, in order to avoid misunderstandings at a later time. Apart from that, advisors should be able to select theses that might be interesting for specific students and share the respective results with those students. Optionally, Chairman could further provide some kind of chat functionality that can be used for exchanging feedback and comments between students and advisors.

- **FR 4: Upload results**

Chairman must enable a student to upload and access his written results which are talk slides and the final result. As there might be additional research results that should be published as well, it should be further possible to upload a *zip* folder with arbitrary content.

3.1.1.2 STAFF MEMBERS

The second stakeholders of Chairman are staff members. The tasks that need to be done depend on whether a staff member is an advisor of a thesis or not. Task 1 and 2 in the following need to be done by all staff members, whereas tasks 3-5 are only relevant for theses advisors.

1. **Issue student theses**

Usually, a student thesis starts with a staff member issuing a new thesis description. The topic of the thesis has to be consulted with the professor of the chair. The description is then posted at the chair's blackboard and the website. Afterwards, students can apply for the thesis and get in contact with the respective staff member.

2. **Supply documents for students**

As already known, students need to observe several formalities. These formalities are partially described in documents like *How-To's* and *Guidelines*. Apart from that, there are Latex templates for talk slides and the thesis itself that fulfill the formal requirements. Therefore, staff members or maybe specially commissioned persons need to supply these documents and templates for the students.

3. Make and observe appointments

A task specifically relevant for advisors is to be the link between students and the professor. For instance, it is required to get in contact with the professor and make appointments, like talk dates. These dates must then be discussed and communicated with the student and observed.

4. Stay in contact with the student

This task is the counterpart to a task already mentioned in Section 3.1.1.1. Advisors have to meet with the student on a regular basis, give feedback on the current results and help on finishing the thesis successfully.

5. Grade the student's results

The final grade of a thesis is made by the professor. However, advisors are usually more involved into a thesis than the professor since they have regular meetings with the student and are informed about the progress of the thesis. Furthermore, they can tell how well the student has worked on the thesis, for example whether he has always been prepared in the meetings or whether he has been motivated at all. Hence, the last task of advisors is to write a grading proposal that is handed over to the professor.

According to the five tasks mentioned above, we can derive three further functional requirements that are not already part of the requirements mentioned in Section 3.1.1.1.

- **FR 5: Create/delete and update theses**

Chairman must offer a possibility to create, update and delete student theses. Formal thesis information that needs to be filled in is the title of the thesis, the type and the advisors. After a thesis is assigned to a student, the name of the student has to be captured as well. Apart from that, it must be possible to update the thesis information with talk, registration and submission dates. Automatically generating *pdf* postings for the blackboard/website is not required, but at least it should be possible to upload existing postings.

- **FR 6: Upload documents relevant for students**

This functional requirement is the counterpart to FR 2. Since documents like *How-To's*, *Guidelines* and Latex templates can vary over time, it is not reasonable to treat these documents as static files in Chairman, but instead offer a possibility to upload new versions and exchange the existing ones.

- **FR 7: Write/inspect grading proposal**

Chairman must enable advisors to generate, inspect and update grading proposals by filling out several forms in the application interface.

FR 2, FR 3 and FR 4 must also be fulfilled in Chairman from the perspective of staff members. In particular, advisors need to be able to upload results, as well. This is relevant when, for instance, student results are already available, but the student has left the university before finishing the thesis. Apart from that, advisors must have access to the uploaded results of students.

3.1.1.3 PROFESSOR & SECRETARY

The third stakeholders of Chairman are the professor and the secretary. Even though they have to deal with different business in general, these two stakeholders can be seen as one. That is because the secretary is the professor's assistant and should therefore have the same possibilities in order to take over some work. The tasks can be described as follows:

1. **Manage formalities like contracts and registration documents**

This task is mainly performed by the secretary. Contracts with students need to be prepared for being signed by the professor and registration documents must be handed in at the *Infopoint*. Furthermore, signed documents are usually scanned by the secretary for the reason of having a digital reference.

2. **Make and observe appointments**

This is the counterpart to a task already mentioned in Section 3.1.1.2. The professor needs to make appointments, like talk dates with the advisors. In these talks he needs to be present, give feedback and make improvement proposals for the thesis.

3. **Grade the student's results**

The final grade of a thesis is made by the professor. However, advisors are required to hand in a grading proposal that the professor can use as the foundation for submitting the final statement.

From the three tasks we can derive only one more functional requirement for Chairman that has not been pointed out before yet. However, we can also infer one additional requirement.

- **FR 8: Upload scanned documents**

Exclusively for the secretary Chairman must offer a possibility to upload scanned documents. These documents can for instance be signed contracts or grading statements which should be managed by Chairman.

- **FR 9: Finish thesis**

Chairman must offer a possibility to mark a thesis as finished. From that moment on, it must not be feasible to upload results or change thesis information anymore. This functional requirement cannot directly be derived from the tasks of the professor and the secretary. However, we think it is best mentioned in this context because a student thesis usually ends with the grading, which is the professor's task.

FR 2, FR 4, FR 5 and FR 7 must also be fulfilled from the perspective of professor and secretary. However, FR 4 and FR 5 are only partially required. This means that there is no need to upload any student results, but instead access the uploaded documents and it is only required to update talk dates, but nothing else.

3.1.1.4 SYSTEMS OF THE CHAIR

Systems or tools of the chair, like the website or the *Talk-Recording* tool can be seen as the last stakeholder of Chairman for the moment. Perhaps in the future, new stakeholders or at least new systems might be coming. The task of these systems can currently be combined:

- **Access or publish student results**

After finishing a student thesis, the results are published at the chair's website. At the moment, this is exclusively the case for the final thesis. However, in some situations it could make sense to publish talk slides, as well. The *Talk-Recording* tool on the other hand does not publish any results, but instead accesses them and makes them available at the presentation laptop. Furthermore, we can think of some terminal program in the future that enables the user to inspect the results directly from the command line.

From this task we can derive the last functional requirement of Chairman:

- **FR 10: Make student results available**

Chairman must provide a technical interface that gives other systems easy access to student results, as well as parts of the formal theses information. This applies to the name of the student, the name of the advisors and talk dates. All other information, especially critical data must not be leaked to other systems under all circumstances. The mechanism can either be push or pull.

As described in FR 10, systems need to have access to parts of the formal theses information. Therefore, FR 5 is partially relevant from this perspective, as well.

The entire functional requirements consisting of ten individual statements are summed up and listed in a more readable tabular form at the end of the following section.

3.1.2 LIFECYCLE OF A THESIS

In the last section we have analysed the different stakeholders of Chairman and have come up with ten individual functional requirements. One of the major goals of Chairman at all is to cover the entire lifecycle of a thesis. For that reason, we explicitly model the lifecycle in this section. With the model we can also prove whether the list of functional requirements is complete or not.

An illustration of the lifecycle is shown in Figure 3.1. As one can see, a thesis consists of eleven phases that are passed through one after another. In each of the phases there are several tasks that need to be carried out by the previously introduced stakeholders. These tasks are presented as ovals around the respective phases.

Each task that is not connected with a dashed line in the figure can directly be assigned to one of the ten functional requirements. *Scheduling talk dates* and *creating an LDAP account* are drawn with a dashed line. That is because those tasks are not part of any of the functional requirements yet. Fortunately, there is no need to cover these tasks in Chairman directly: Scheduling talk dates requires a communication between the professor, the advisors and the student which should be done verbally without the usage of Chairman. The only task of Chairman in this context is to capture the appointments. On the other hand, creating an LDAP account is a prerequisite of using Chairman at all because users are defined there. This needs to be done manually by the chair's system administrator.

As another abstraction we can summarize the eleven phases even further and end up with only three phases which is helpful in order to group and structure the tasks of Chairman. The phases are **Organisation & Registration**, **Working Time & Talks** and **Submission & Grading**.

One should not forget that the lifecycle, as presented here, follows the way theses are handled at the chair of *Network Architectures and Services*. For other chairs this might be similar, but not exactly the same.

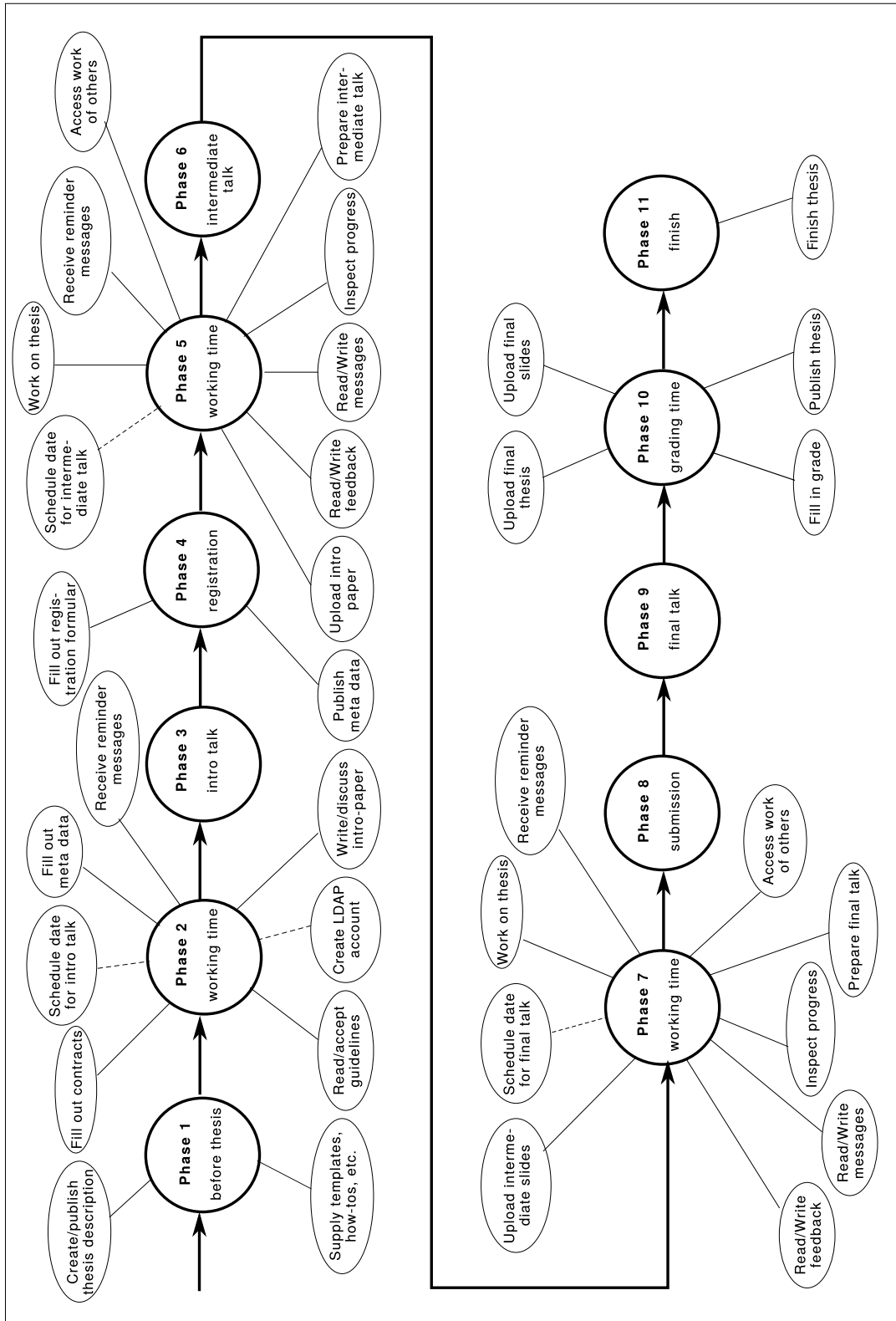


FIGURE 3.1: Lifecycle of a Student Thesis

3.1 FUNCTIONAL REQUIREMENTS

Summary: We have now analysed the functional requirements from two different perspectives: Firstly, from the perspective of the individual stakeholders. Secondly, from a temporal view. Since the same requirements can be derived from both perspectives, we can say that the list of functional requirements is complete.

In the following, two tables are provided that sum up the functional requirements of Chairman. For convenience, Table 3.1 lists the names again. The exact meaning of each requirement is explained in Section 3.1.1.

Table 3.2 assigns the requirements to the three phases and shows for which stakeholders the respective requirement is relevant.

- FR 1:** Provide downloads
- FR 2:** Inspect progress, next steps and receive reminder mails
- FR 3:** Manage exchange between advisors and student
- FR 4:** Upload results
- FR 5:** Create/delete and update theses
- FR 6:** Upload documents relevant for students
- FR 7:** Write/inspect grading proposal
- FR 8:** Upload scanned documents
- FR 9:** Finish thesis
- FR 10:** Make student results available

TABLE 3.1: Functional Requirements of Chairman

Phase	FR	Stakeholders			
		Students	Staff	Prof/Sec	Systems
Organisation & Registration	FR 5	✗	✓	✓	✓
	FR 6	✗	✓	✗	✗
	FR 8	✗	✗	✓	✗
	FR 10	✗	✗	✗	✓
Working Time & Talks	FR 1	✓	✗	✗	✗
	FR 2	✓	✓	✓	✗
	FR 3	✓	✓	✗	✗
Submission & Grading	FR 4	✓	✓	✓	✗
	FR 7	✗	✓	✓	✗
	FR 9	✗	✗	✓	✗

TABLE 3.2: Relevance of the Functional Requirements

3.2 NON-FUNCTIONAL REQUIREMENTS

The second aspect of the **Requirements Specification** are non-functional requirements (**NFR**) which are analysed in this section. The functional requirements have already been explained in Section 3.1. Basically, the non-functional requirements are part of the research questions of this Master's Thesis and hence already mentioned in Section 1.3. However, we go into more detail now and make more concrete statements.

First of all, it must be clarified once more that Chairman needs to deal with critical data. Examples of such data are final grades and personal information about students, like matriculation numbers, addresses and telephone numbers. Since contracts and registration documents have to be processed, this kind of data cannot be prevented from being collected. With this knowledge, we can formulate the first non-functional requirement as follows:

- **NFR 1: Privacy preservation**

Chairman must implement mechanisms that ensure the observance of the **Privacy Protection Goals** unlinkability, transparency and intervenability. How these goals can be achieved in general has already been described in Section 2.1. The main focus in this context should be put on a clear separation between critical and non-critical data. Apart from that, users should be informed when personal data is processed and should be able to react to or completely stop that processing.

As explained in Section 2.1, security is related with privacy, but is in general concerned about defending attacks. In this context, it makes sense to distinguish only between two types of attacks: Firstly, all kind of attacks that aim for stealing critical data or changing it in an unauthorized way. Secondly, all kind of attacks that aim for shutting down the system. The second type is, however, not really about application security, but instead requires mechanisms to be applied to the application server in production. This is not in our responsibility which is why we mainly concentrate on the first type and formulate the second requirement as follows:

- **NFR 2: Application security**

Chairman must implement mechanisms that prevent attackers from stealing private data or changing it in an unauthorized way. This should be achieved by implementing the security patterns, as defined in Table 2.1 in Section 2.1.

A subsequent step of making a system transparent (see NFR 1), is to make user actions accountable. This means that all important actions can be assigned to a concrete user and if possible, to a concrete thesis. This is required to show who is responsible for several actions, to reveal discrepancies and to prevent data misuse.

- **NFR 3: Accountability**

Chairman must implement a mechanism that assigns the main actions and the ones that deal with critical data to concrete users and theses. The assignment must not be disputable.

The fourth requirement is about flexibility, extensibility and maintainability of the system. Easy maintainability requires developers to understand what is happening in the code. We have already seen that understandability leads to transparency which in turn is required for accountability. Hence, we can demand easy maintainability in order to achieve other non-functional requirements. Currently, Chairman is tailored only to the use cases of the chair of *Network Architectures and Services*. Maybe in the future other chairs want to use Chairman, as well. Hence, it should be possible to extend Chairman with new functionalities without changing much of the existing code.

- **NFR 4: Flexibility, extensibility and maintainability**

Chairman must be implemented in a way that allows easy extensibility and maintainability. Especially for fellow students, it must be possible to do maintenance work or extend Chairman with new features without knowing all of the internals. Flexibility should be addressed by implementing Chairman in a way that allows to dynamically react to different load rise the system might experience.

Easy usability is a requirement that is implicitly demanded in any application, hence as well in Chairman. At minimum, *easy* means that all functionalities provided by Chairman have to be executable in a way that does not require any learning before. Apart from that, all user actions must do what one would instinctively expect. This refers for instance to mouse clicks: a left click results in selecting something, whereas a right click opens a context menu.

- **NFR 5: Easy usability**

Chairman must be usable in a way that does not require any learning before and user actions must behave as one would instinctively expect.

The last non-functional requirement of Chairman is about performance and availability, even though availability is actually part of the **Security Protection Goals**, as explained in Section 2.1. Performance aspects taken into account in this context are firstly, how long it takes to perform an average function of Chairman and secondly, which system requirements the machine needs to have on which Chairman is deployed. For availability, it makes sense to distinguish between primary functionalities and secondary functionalities which can still be carried out without Chairman, if necessary.

- **NFR 6: Good performance and high availability**

Performing a function in Chairman must not take longer than a few seconds and the overall user experience should be as fluid as possible. Apart from that, Chairman should be runnable on an average machine with a modern processor and around 8 GB of RAM. Primary functionalities should be available for the entire time when Chairman is running, whereas secondary functionalities might be unavailable sometimes.

Summary: We have now analysed the non-functional requirements of Chairman. Together with the functional requirements, the **Requirements Specification** is now complete and the first phase of the Software Lifecycle Model is finished.

To conclude this section, we provide Table 3.3 which sums up the non-functional requirements for later reference.

- NFR 1:** Privacy preservation
- NFR 2:** Application security
- NFR 3:** Accountability
- NFR 4:** Flexibility, extensibility and maintainability
- NFR 5:** Easy usability
- NFR 6:** Good performance and high availability

TABLE 3.3: Non-Functional Requirements of Chairman

CHAPTER 4

ANALYSIS

The second phase of the Software Lifecycle Model is called **Analysis**. While the requirements have been specified in Chapter 3, we provide a possible solution for a system that fulfills the requirements in this chapter. However, we mainly concentrate on the non-functional requirements first as they are more interesting from an academic point of view and have an even bigger influence on how the solution will look like. We take care of the functional details in the second part of this chapter and in Chapter 5.

4.1 ANALYSIS OF THE SOFTWARE ARCHITECTURE

The only initial constraint for Chairman has been to implement it as a web application since this is one of the easiest ways to allow many people using a software product. Apart from that, there are no technical instructions what Chairman should look like. Therefore, the next step is to define a software architecture which is suitable for the web and allows to fulfill the requirements. Beforehand, a few terms need to be introduced.

4.1.1 TERMINOLOGY

Software Architecture

Many different definitions for *software architecture* can be found in the literature. In 1994 a discussion group defined the software architecture as "*the structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time*" [20]. According to this definition, the architecture clarifies which parts or components shape a system and how the individual

components are related to each other. The said guidelines are often driven by the non-functional requirements which have a big influence on the entire structure of the system. An important aspect to remember is that the term *software architecture* denotes the architecture of a *concrete* system. Sometimes a much clearer picture is given by the term *architectural instance* [20].

Architectural Style & Architectural Pattern

Most of the time, these two terms are used synonymously in the literature which is also the case for this Master's Thesis. In contrast to an *architectural instance*, both terms describe a general solution that is not fine-tuned for a concrete system. Examples of such patterns are the *client-server*, the *publish-subscribe* or the *blackboard* pattern [11].

According to [11], patterns and styles are actually not the same since patterns are considered to be problem-solution pairs, whereas styles do not focus on a specific problem. To stay precise, the answer to *why* a specific problem can be solved with an architecture is hence only given in patterns, not in styles [11].

Architectural Family/View/Category

A group of architectural styles that share a common concept form a family. Examples of such concepts are *database-centered*, *layered* or *component-based*. The assignment of styles to a family is not always clear and most of the time, there is more than one possibility. In this Master's Thesis we use the term *architectural family*. Synonyms that can be found in the literature are for instance *architectural view* [11] or *category* [21].

4.1.2 FIRST APPROACH (MONOLITHIC)

Since Chairman should be implemented as a web application, one might intuitively think about dividing the system into two components: **Frontend** and **Backend**. The frontend is thereby the part of the system that the user sees and that interacts with him. On the other hand, the backend is where the business logic is implemented.

Dividing a web application into frontend and backend is in general a valid possibility. The architectural style that this division follows is typically referred to as **Client-Server** [11] in the literature, in which the frontend is part of the client and the backend is part of the server. The client-server pattern is characterized by an asymmetric communication between the two components [11]. This means that the client starts the communication by asking for a service that is provided by the server [11]. An illustration of this pattern can be found in Figure 4.1a.

However, treating the backend server as one logical block is in general too narrow-minded because technical aspects of the backend are not considered at all. At minimum, it is required to split the server into the three layers **Presentation**, **Business** and **Data**. The presentation layer is responsible for creating the UI components that are fetched and displayed by the client. This layer is often described simply as **UI** in the literature [43]. The business layer is responsible for performing the business logic which includes at least the functional requirements of the system [43]. Sadly, this layer is often called again **Backend** which makes a distinction with the classical client-server pattern more difficult. Lastly, the data layer is responsible for creating structures for the storage and persisting the data [43]. Fortunately, this layer is often simply called **Database**.

The resulting style can be described as **Layered Client-Server** [11] and is illustrated in Figure 4.1b. It can be assigned to the family of *database-centric* architectures. In contrast to the classical client-server style, the functional responsibilities of the server are now regulated more clearly.

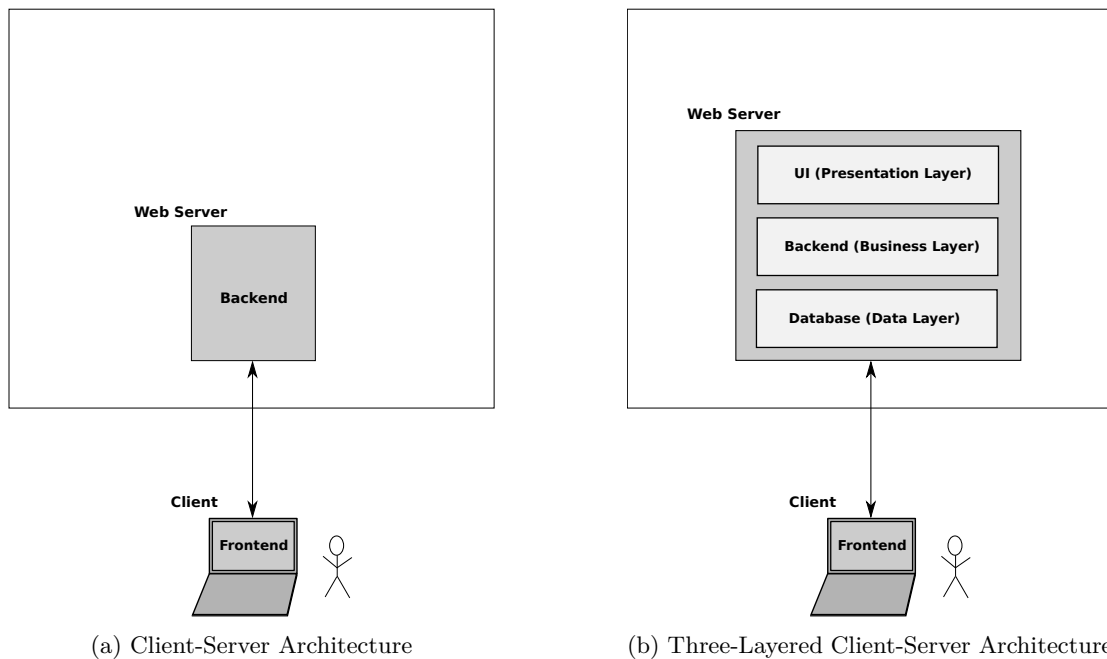


FIGURE 4.1: Monolithic Architecture

Even though we have not taken into account any of the functional requirements yet, we can recognize the following pros and cons when utilizing the layered client-server style for Chairman:

Pros:

- Application security (NFR 2):

A monolithic web server is a good starting point for enabling application secu-

rity. The main reason is that the complete server logic is physically at the same location and not distributed over a network. This makes it easy to implement security patterns [44] like **Single Access Point** and **Check Point**, as defined in Section 2.1. **Roles** with different authorizations can be realized and a classical authentication mechanism can be applied. This means that a user authenticates himself at the check point which in turn creates a **Session** object where the user information is stored and returns a cookie with the corresponding session Id.

- Good performance (NFR 6):

The performance of a system mainly depends on its concrete implementation. However, a monolithic server is again a good starting point because all of the communication between different parts can happen in place without a high latency. Having only one server may also result in less memory consumption than spanning up multiple server units.

Cons:

- Privacy preservation (NFR 1):

In a monolithic system it is not possible to implement all privacy protection goals in a meaningful way. The main problem is that unlinkability cannot be fulfilled as it is mostly achieved by *data avoidance* or *separation*. Since sensitive data like final grades should be managed by Chairman, we cannot prevent this kind of data from being collected. Rather should a separation mechanism be applied to Chairman. However, the three-layered approach does only provide one database in which all kind of data is stored, independently of the criticality. This means that final grades are stored in the same database as talk dates or templates. Separating critical data from the rest by using different tables is not enough. Instead, we should introduce a physical separation and apply extra security measures to the critical parts. This way, an attacker is not able to see all data if he manages to get access to the database. Apart from that, the physical separation is beneficial since non-critical parts of the application can then even be administered by students.

- Flexibility, extensibility, maintainability (NFR 4):

Extensibility and maintainability again depend a lot on the concrete implementation, but there is in general a relation recognizable between these two attributes and the size of the codebase. A smaller codebase allows to understand a system much faster and makes it therefore much easier to extend or maintain existing parts of it. However, a monolithic system will probably result in a big codebase which means that extensibility and maintainability are rather bad.

Flexibility has been defined in this context as a property that allows to dynamically react to different load rise that the system might experience. However, the only possibility to handle a high load rise is basically to deploy the entire system to a more powerful machine which will be a waste of resources most of the time.

Accountability (NFR 3) and usability (NFR 5) are neither listed in pros, nor in cons. Accountability requires to capture actions and usability mainly depends on the implementation of the frontend. Hence, we cannot make general statements without a concrete solution.

To conclude this section, we have to say that the layered client-server architecture is not suited for Chairman since the non-functional requirements, especially NFR 1 cannot be fulfilled.

4.1.3 REFINEMENT (MICROSERVICES)

Since the last section has shown that the three-layered client-server style is not suitable for Chairman, we make a refinement analysis for the software architecture. The ultimate goal is to get rid of the cons mentioned in Section 4.1.2 while preserving the pros of the first approach. However, we have to find a compromise solution.

In order to improve extensibility and maintainability, we should split the monolithic codebase into smaller pieces where the individual pieces have a high cohesion. This way, we can focus on specific parts of the application when implementing new features or doing maintenance work which is much easier. A *database-centric* architecture does, however, not offer many possibilities for dividing the business logic into pieces. Better suited might be an architectural style that belongs to the family of *component-based* architectures. In these styles the application is split into parts where each part fulfills a distinct purpose. Popular examples utilizing such style are the *Automotive Open System Architecture* [10] and the *Android Operating System* [3].

To achieve privacy - especially unlinkability - we have concluded that a separation mechanism must be applied. Hence, splitting the system into components seems to be a good approach from that perspective as well since sensitive data could then be separated from the rest. However, for a complete physical separation between sensitive components and normal ones, we need to make sure that these components can even be deployed independently. This aspect is apparently not a goal of *component-based* architectures since no literature could be found where this is mentioned. Instead, this is one of the main goals of *service-oriented* architectures [29].

A relatively new representative of *service-oriented* architectures is the concept of **micro-services** which are basically driven by the three-layered client server style [43]. The three layers are often considered to be the result of **Conway's Law** which has been formulated by Melvin Edward Conway [43]. This law states that a system designed by an organization will always have a structure that is a copy of the organization's communication structure [43]. Due to that mentioned *communication structure*, we can conclude that there are usually three teams where each team is responsible for one of the three layers.

However, the problem is that the implementation of a new feature usually requires changes in all three layers and is therefore quite hard to achieve since all three teams have to be coordinated accordingly. It would make more sense to not decompose the system only based on these technical aspects, but rather on functional aspects and utilize the three layers in each part. This is the basic idea of microservices [43]. An illustration can be found in Figure 4.2a. As one can see, there is no longer one monolithic server, but instead n different services which in turn are structured by the three layers. The services are the result of dividing the system by **functional** aspects. Hence, the services have a clear boundary which makes it possible to deploy them independently from each other. In addition to that, we explicitly use microservices to isolate critical from non-critical data which means that we do not divide the system by functional aspects only, but also by non-functional aspects like privacy. For this reason, microservices seem to fit perfectly for Chairman.

Flexibility is improved by microservices, as well because in the case of a high load rise, it is only required to start multiple instances of one service and utilize a load balancer.

However, in contrast to the first approach, microservices are worse for security since they provide a bigger attack surface. That is due to the distribution over the network. There is no longer a single access and check point. Furthermore, the classical authentication mechanism does not work anymore. The reason for this is that the microservices are explicitly meant for having a clear boundary and not holding a shared state in the form of a session object. Indeed, the individual services could manage a session, but that would require a user to authenticate himself at every microservice which would be a really bad user experience. Performance may also get worse because communication over the network results in a higher latency and memory consumption gets higher because of multiple server units. Lastly, there are technical issues distributing the UI layer and achieving a consistent look and feel is nearly impossible.

Therefore, we should make another refinement to the microservices approach by implementing the so-called **API-Gateway** pattern [29]. The crucial point of this pattern is the

interposed gateway between the client and the microservices. The client does no longer talk to the microservices directly, but instead communicates only with the gateway. The gateway in turn acts as a proxy and routes the requests to the respective services. This way, it is even possible to configure the microservices to be only accessible via the gateway and hence, the gateway becomes the new single access point. This is a big improvement for security. As another simplification, we implement the individual microservices completely stateless and let them communicate with the gateway and with each other only via REST. Because of that, there is no need to manage a session in the microservices. Lastly, we get rid of the UI layers in the individual microservices and instead develop a monolithic UI that is served directly by the gateway. This enables again a consistent look and feel. An illustration of the planned architecture is finally shown in Figure 4.2b.

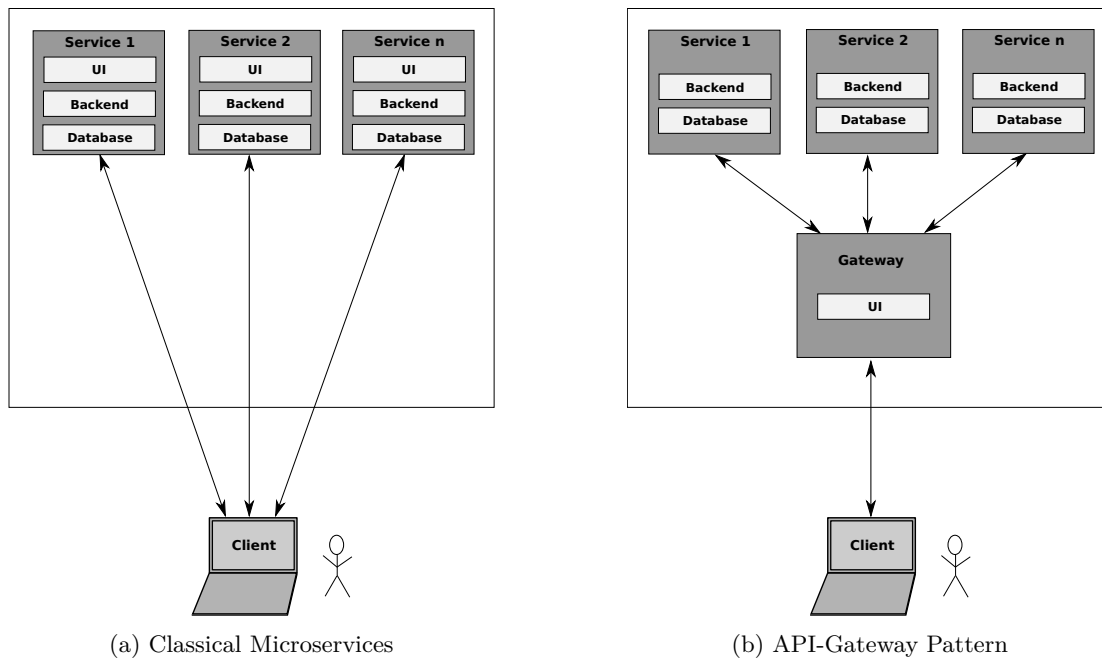


FIGURE 4.2: Microservices Architecture

Summary: We have now analysed different architectural styles that could be suitable for Chairman. We have seen that a monolithic approach (layered client-server architecture) is good for security and performance, but bad for privacy, flexibility, extensibility and maintainability. On the other hand, for a pure microservices architecture essentially the opposite is true. The compromise solution that fits best for Chairman is a slightly amended microservices approach that introduces a gateway between the client and the services and implements a monolithic UI layer.

4.2 AUTHORIZATION WITH OAUTH 2.0

One aspect that is still open from the last section is the authorization mechanism. We have seen that a classical approach based on session and cookies does not work in Chairman. That is due to the distributed software architecture where the microservices are stateless and do not share a session object in which the user information could be stored. Therefore, we should focus on a token-based mechanism that works stateless.

A popular token-based authorization framework is called **OAuth 2.0** [32]. Required background information has already been given in Section 2.2.1. OAuth is mainly known for providing third-party applications access to restricted resources. However, this is not the only use case of OAuth. With appropriate measures we can also use it for **Single Sign On** in a distributed application. Single Sign On means that a user is requested only once to authenticate himself against multiple services. In our case, these services are exactly the microservices of Chairman.

In order to integrate OAuth 2.0 into Chairman, we should start by defining a new microservice that is responsible for authentication and authorization. This microservice is the **Check Point** according to the security patterns, as defined in Section 2.1. The modified architecture and the assignment of OAuth roles to components of Chairman are shown in Figure 4.3. All of the already introduced microservices are **Resource Servers** whereas the new microservice is the **Authorization Server**. The human user is called **Resource Owner** and the **OAuth Client** is the already known client of Chairman.

The goal is now to let the client retrieve the authorization of the end-user to access the resources hosted at the resource servers. In this case, the resources are REST endpoints. As explained in Section 2.2.1, OAuth specifies four different *authorization grant* types. In this scenario, it would actually be fine to utilize the **Resource Owner Password Credentials** type since the client is no third-party client, but belongs directly to the application. Hence, there is no security problem when the resource owner's credentials are directly given to the client. Nevertheless, the specification prescribes to use this grant type only if there are no other options available [23]. For that reason, we implement the **Authorization Code** type. The protocol (taken from [23]) is described in Table 4.1 and is also illustrated in Figure 4.3¹.

¹The protocol steps 3 and 4 are actually proxied through the gateway which is not drawn in Figure 4.3 for simplicity

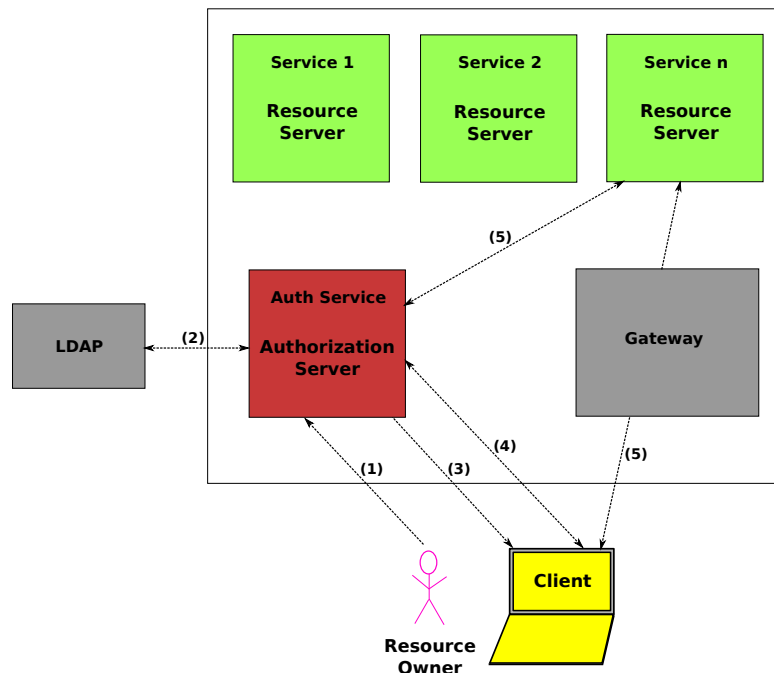


FIGURE 4.3: OAuth 2.0 in Chairman

- (1) The client directs the resource owner to the authorization server via the user agent.
- (2) The resource owner authenticates himself with his LDAP account. We assume that an LDAP account exists for every user of Chairman.
- (3) If the authentication in step 2 has been successful, the authorization server redirects the resource owner to the client with an attached authorization code.
- (4) The client authenticates itself at the authorization server and exchanges the authorization code for an access token. The authentication requires a client to be equipped with a client id and a secret which are pre-configured in the application.
- (5) The client sends the access token to the resource servers in a header field in every request. The resource servers in turn check whether the token has not expired yet and whether that token authorizes the client to perform that specific request. If yes, the request is handled.

TABLE 4.1: Authorization Code Grant Type Protocol

After logout, access should be denied by all resource servers. This is simply achieved by deleting the access token at the authorization server as all resource servers check the validity there. Since all access tokens have an expiry date, the tokens are deleted anyway after the expiration.

In 2014 a security vulnerability called **Covert Redirect** was found in the OAuth protocol by Wang Jing [33]. Even though, the vulnerability was very popular in the news, it was actually not a new finding since the method had already been described before as a threat called *Open Redirector* in an RFC about OAuth Security [26]. The problem is that after authentication, the resource owner gets redirected from the authorization server to a URL specified by the client. If an attacker manages to replace the redirect parameter with a malicious URL, they may get access to the authorization code and retrieve a valid access token in a subsequent step. In general, this vulnerability is a problem since the only way to prevent this attack is to register full redirect URLs for all clients [26]. This basically means that OAuth providers need to manage a whitelist of valid redirect URLs. This is really hard to achieve for big providers like Facebook or Google. However, in our case there is a straight forward countermeasure: Since there is only one client that uses the same redirect URL all the time, this URL can simply be pre-configured in the authorization server as the only valid URL. This means that **Covert Redirect** is not an issue in Chairman.

4.3 ACCOUNTABILITY ANALYSIS

Accountability has been demanded both, as a research question (RQ2) and as a non-functional requirement (NFR3). Accountability relies on transparency which in turn is achieved by logging and reporting, among others [17]. In order to make a system accountable, it is required to record important actions and assign them to concrete users. The assignment must not be disputable. The existing accountability mechanism, based on subversion, does no longer work in the new version of Chairman as we do not use subversion anymore.

4.3.1 DEFINITION OF CHAIRMAN EVENTS

The first step is to recognize important actions. Actions are derived from the functionality offered by Chairman which means that we can derive them from the functional requirements, as listed in Table 3.1. Not all possible actions a user can perform are worth capturing. Some actions are simply too trivial, for example viewing the list of upcoming events, or they do not result in requesting or changing any critical data. An example is downloading template files.

Table 4.2 lists all actions that should be made accountable in our opinion because they are either part of the main functionality of Chairman or they deal with critical data. The last column of the table shows from which functional requirement the action is derived.

	Action	Derived
A1	Thesis has been/is no longer shared with another student	FR3
A2	Result has been uploaded	FR4
A3	Thesis has been created/deleted	FR5
A4	A student has been added/removed to/from the thesis	FR5
A5	A talk date has been added/removed	FR5
A6	A talk date has been confirmed/unconfirmed	FR5
A7	A static document has been uploaded	FR6
A8	Grading proposal has been uploaded/deleted	FR7
A9	A document has been uploaded	FR8
A10	Thesis has been finished/reopened	FR9

TABLE 4.2: Actions in Chairman

The next step is to combine each action with additional attributes that clearly describe the assignment to a concrete user and possibly to a concrete thesis. Therefore, we introduce the term **Chairman Event**, or simply **Event**.

As one can see in Figure 4.4, a Chairman event consists of the action and six additional attributes. The *user name* is required for assigning the action to the user, whereas the *thesis Id* is used to assign the action to a thesis. Only action A7 cannot be assigned to one single thesis. In this case, the field should be left empty. *Time stamp*, *IP address* and the *user agent* can be used for plausibility checks later on. For the actions A2, A7, A8 and A9 it is required to guarantee which file exactly has been uploaded. Hence, we calculate and store a *hash value* (e.g. SHA-3) of the file. For all other actions this field should again be left empty.

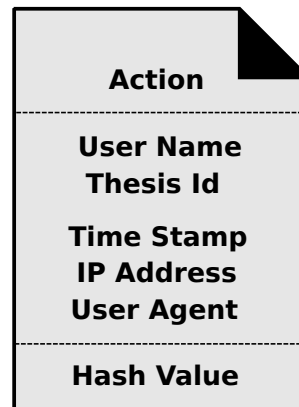


FIGURE 4.4: Chairman Event

4.3.2 PROCESSING OF CHAIRMAN EVENTS

In the last section we have shown what a Chairman event should look like. In this section we analyze how the actions listed in Table 4.2 can be reliably detected, bundled in an event and evaluated in order to prevent data misuse.

A valid question is how collecting and storing such critical information like IP addresses together with user names is in harmony with preserving the privacy of the users. From the *GDPR* (Art. 5, Par. 1 (b)) we know that collecting personal data is allowed if there are legitimate purposes [22]. This is, however, a matter of interpretation. If we did not store the collected information, we would not be able to guarantee accountability at all. As explained later in this section, the IP addresses can be used for making plausibility checks that help preventing data misuse. Hence, we argue that there are in fact legitimate purposes. However, we need to make sure that the stored data is sufficiently protected against unauthorized parties (see Par. 1 (f)).

We should start by defining a new microservice that is responsible for storing and evaluating Chairman events. This way, we can separate the accountability aspect from the actual functionality of Chairman. The new microservice is called **Monitoring Service** and has a special position which is illustrated in Figure 4.5. Interestingly, we can still view the other microservices as black boxes. How the storage technology has to look like is currently open. Possible solutions are a simple database or a blockchain-based backend.

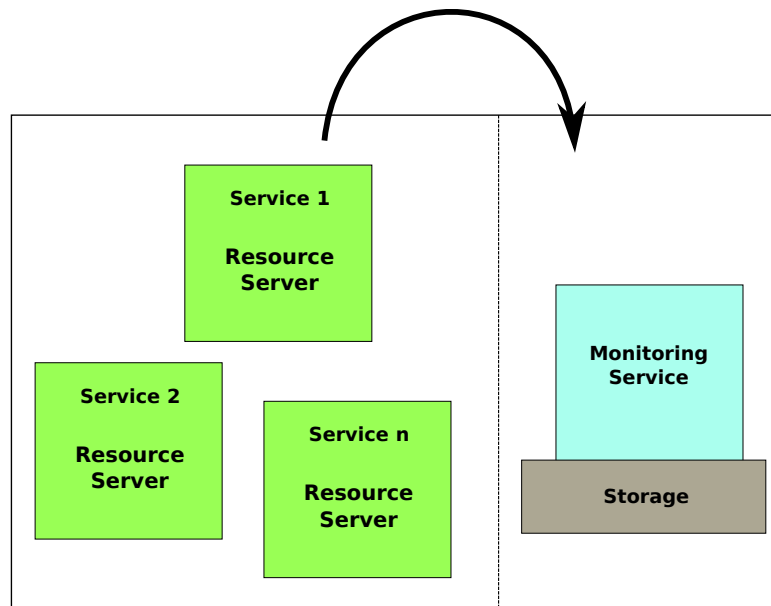


FIGURE 4.5: Monitoring Service in Chairman

Actions must be detected in the individual microservices that implement the respective functionality. Since we have decided to communicate exclusively via REST, it is rather easy to detect actions because this only requires to observe the REST endpoints which are triggered when a user performs an action.

Most of the attributes of a Chairman event can directly be taken from the HTTP request entering the REST endpoint. This applies to the IP address, the user agent and the user name which is taken from the OAuth token. The thesis Id is usually sent either as a path variable or as a request parameter. The time stamp should always be set to the current time and if a file has been sent, it is further required to calculate a SHA-3 hash of that file. The resulting event must then be transferred to the Monitoring Service, again via a REST call.

Storing Chairman events is required for making actions accountable. However, it is even more useful when these events are also evaluated. Since we have collected user agents and IP addresses, we are able to perform plausibility checks. For example, we can recognize if one user suddenly performs actions from a location completely different than usual and with a different device. This may be an indication for an unauthorized person who has got access to the user credentials. In order to prevent data misuse, we could then prohibit access to Chairman for that individual user and inform that user via email. Another meaningful feature would be to simply show the history of actions taken in the web interface.

4.3.3 DATABASE SOLUTION

After the Monitoring Service has received the Chairman events, it needs to store them, so that it is not possible to deny any actions.

The first idea that comes to mind is to utilize a database like MongoDB for the storage. MongoDB is a well-known storage technology and is easy to implement in practice. However, an implicit requirement is that it must neither be possible to delete a database entry, nor change one. It is only allowed to insert new entries. In practice, we can implement that by reacting only to POST, but not to PUT or DELETE requests. However, this approach introduces two problems: firstly, if an attacker manages to get access to the database, we can no longer guarantee accountability since the attacker could tamper with the stored data. Secondly, append-only is a quite uncommon use case for a database and does basically result in a chain of data blocks. Therefore, we should analyze whether it makes sense to use a **blockchain** for that job.

4.3.4 BLOCKCHAIN-BASED SOLUTION

As we have seen in the last section, a simple database is easy to implement in practice, but can not fully guarantee accountability.

Instead, a blockchain would be advantageous since it ensures with cryptographic means that valid blocks, which in turn contain Chairman events, can never be manipulated [12]. This is positive for the required accountability aspect. However, every blockchain has one property in common: it is a distributed and decentralized data storage managed by multiple nodes. This is something which is not directly given in Chairman because we have basically only one system that produces transaction data. Therefore, we should think a bit further: if not only the chair of *Network Architectures and Services* uses Chairman, but also other chairs of the university, the individual chairs could act as nodes for a blockchain. The required infrastructure is illustrated in Figure 4.6 where four different chairs are shown exemplarily. Already at this point we see the complexity of a blockchain due to that infrastructure.

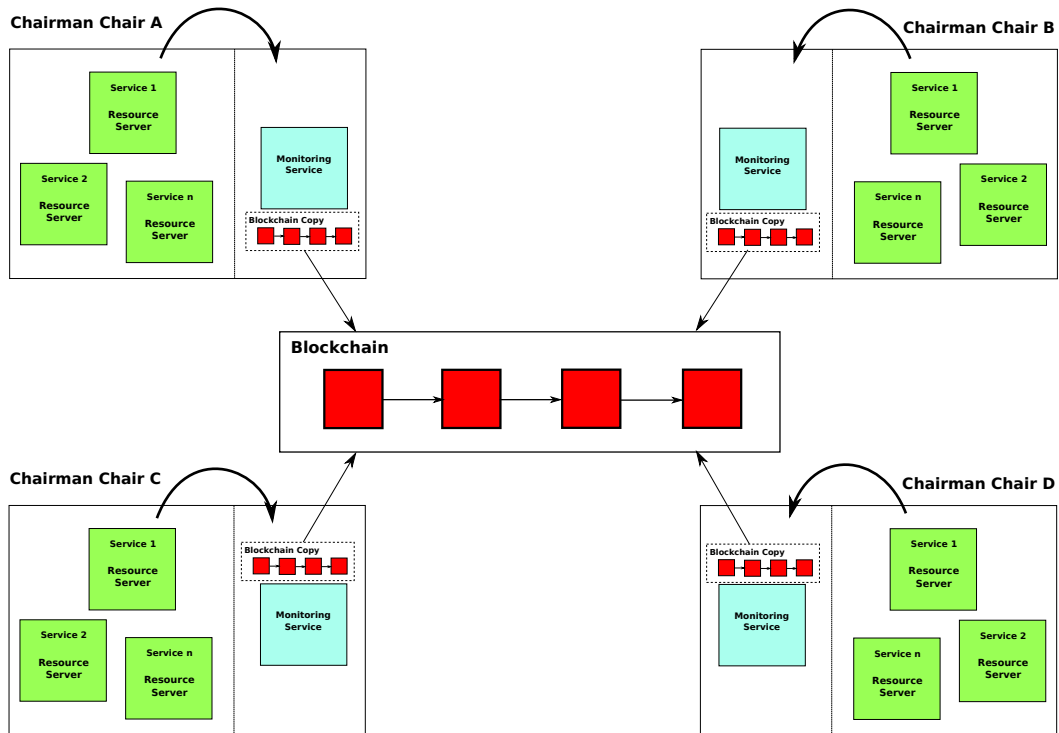


FIGURE 4.6: Blockchain Infrastructure

Nearly all famous public blockchains, as for example known from Bitcoin or Ethereum, use the so-called **proof-of-work** algorithm in order to gain consensus about the order of blocks [12]. The creation of a new valid block requires solving a cryptographic puzzle. This is associated with a high energy consumption which in turn is rewarded with a certain amount of the crypto currency [12]. However, this is not an option in Chairman.

Instead, it would be possible to use the *Hyperledger Fabric* framework because it uses a simple consensus algorithm that does not require any high computational effort. The fact that Hyperledger Fabric can only operate permissioned blockchains can even be viewed as positive. That is because not everybody should be able to participate in the blockchain that assures accountability in Chairman. Instead, only selected entities should be part of the network.

Necessary background information about Hyperledger Fabric has already been given in Section 2.2.2. In the terminology of Fabric, the **clients** are the respective Monitoring Services. Additionally, we need to introduce **peers** which process requests from the clients and manage a copy of the blockchain. Each chair should at least operate one peer. Lastly, we need an **orderer** node that is responsible for bringing blocks and transactions into a generally accepted order.

In addition to the complexity, a blockchain introduces another problem: we have already argued that there are legitimate purposes for collecting and storing critical data, like IP addresses together with user names. However, it is not appropriate to share this information with unauthorized parties which actually happens when using a distributed blockchain. As a consequence of that, we need to encrypt the data before processing it, but we also need to make sure that the endorsing peers are still able to make validation checks. These checks should guarantee that each chair can only create transactions in the blockchain which are related to a thesis of that specific chair. Hence, we need to define an appropriate crypto schema first in which some non-critical meta information are left unencrypted to allow these checks, and the actual transaction data is encrypted for privacy reasons.

As the effort for developing Chairman is already quite high, we leave this aspect open for future work and instead implement a simple database. Especially due to the Monitoring Service, we have, however, already created an accurate interface which enables a connection to a blockchain-based backend at a later time.

CHAPTER 5

DESIGN

The third phase of the Software Lifecycle Model is called **Design**. In the last chapter we have analyzed what a solution should look like to fulfill the requirements, as defined in Chapter 3. We have considered multiple approaches and have tried to stay as abstract as possible. In this chapter we continue the discussion about Chairman in a more concrete way and make tangible decisions.

5.1 CHAIRMAN DESIGN

In the last chapter the individual microservices have been viewed completely as black boxes. In the first part of this section we change that and analyze which concrete microservices are needed. The second part explains which technologies are used for implementing Chairman.

5.1.1 MICROSERVICES

A few microservices, relevant for technical reasons, are already known from the analysis phase. Firstly, the **Auth Service** which is the OAuth authorization server and therefore manages OAuth tokens. Secondly, the **Monitoring Service** which has been introduced to separate the accountability aspect from the actual functionality of Chairman. Lastly, the gateway which acts as a router and which serves the UI part of Chairman. We aptly call this service **UI-Gateway Service**.

A fourth service can also be derived from the current knowledge point: the service discovery. This is a special service that simplifies the communication between microser-

vices and the frontend. For this purpose, all microservices register themselves at the service discovery with a unique name, their URL and a port number. Afterwards, it is only necessary to know the names of the microservices in order to communicate with each other. The correct addresses are resolved automatically by the service discovery. As we will see in the next section, we reuse the service discovery provided by Netflix. Hence, the fourth service is called **Eureka Service**.

All other microservices which implement the actual functionality of Chairman are currently unknown. Therefore, we go through the entire list of functional requirements (see Table 3.1) and analyze which additional microservices are needed. Defining one microservice per functional requirement is not useful and can be considered as a typical antipattern, called **nanoservices** [34]. Instead, we should follow software design principles and best practices for decomposing a system into components in general, or particularly into microservices.

One major goal is to achieve *loose coupling* and *high cohesion* between the microservices [43]. This means that everything which belongs together should be placed into the same microservice and there should be as less dependencies on other services as possible [43]. This leads to more reliable and more maintainable products [24]. Changes in one service do not require other services to be changed accordingly if there are no dependencies. Other well-known principles related to loose coupling and high cohesion are the *Single Responsibility Principle* [35] and *Separation of Concerns* [18]. These principles are quite similar and short, they mean that each service should fulfill a distinct purpose.

To become more concrete, we focus on the following properties when defining the microservices:

- **Context:**
The first indication for placing functionalities into separate microservices is that they do not share the same context.
- **Critical Data:**
As already known from Chapter 4, we have to place functional features that deal with critical data into isolated microservices for privacy reasons.
- **External Dependencies:**
Functionalities with external dependencies should also be placed into distinct microservices because of easier maintenance.

By analyzing the list of functional requirements, one can actually recognize only three different contexts. Nearly all requirements deal with the context **Thesis**, whereas particularly FR1 introduces two more contexts: **Students** and **Registration**. That is because it should be possible to download pre-filled contracts and registration documents. For this purpose, we need to collect and manage personal information about students first and create these documents afterwards. Personal information has to be treated as critical data. Hence, we should place that feature into the fifth microservice, called **Meta-Data Service** and the functional details for creating these documents into the sixth microservice, called **Registration Service**.

Apart from personal information, the final grade has to be considered as critical data. This is covered by FR7. Even though the final grade belongs to the context *Thesis*, we should separate that aspect from the rest. This leads us to the seventh microservice, called **Grading Service**.

FR2 includes sending of emails which has an external dependency on an SMTP server. For that reason, we define one microservice that is responsible for messaging (FR3) and emails (FR2). The eighth microservice is hence called **Message Service**.

FR8 is covered by the Meta-Data Service and the Grading Service. All other functional requirements not named yet, do neither introduce any new external dependencies, nor deal with critical data. Furthermore, they all have the same context. For that reason we place them all into the last microservice which is called **Thesis Service**. At the end of this section, all defined microservices are again listed in a tabular format.

A well-known best practice in software engineering is called *Don't Repeat Yourself* (DRY) which means that there should be as few redundant code as possible. This is for example useful for bug fixes since a bug fix does not have to be performed multiple times then. However, in a microservices architecture it is often a good idea to violate this principle, at least across the service borders [43]. The reason is that microservices can stay independent from each other and can therefore even be deployed individually [43]. This has been one of the main reasons why we have looked into microservices at all. On the other hand, one should certainly try to adhere to the DRY principle in a microservice itself.

As explained, we should aim at low coupling when defining the microservices and all microservices should have a clear boundary. For this reason, we create a **Dependency Structure Matrix** (DSM) for the five newly introduced microservices in order to find out whether we have met the requirements. The DSM is illustrated in Table 5.1. An

entry in the matrix means that the item in the column has a dependency on the item in the row. The ultimate goal is to have as little dependencies as possible.

	Grading S.	Message S.	Meta-Data S.	Reg. S.	Thesis S.
Grading S.	-				
Message S.	X	-			X
Meta-Data S.			-	X	
Reg. S.				-	
Thesis S.	X			X	-

TABLE 5.1: Dependency Structure Matrix of the Microservices

First of all, we can recognize some positive aspects in the DSM: most of the matrix entries are empty which means that there are not many dependencies at all. Secondly, we do not have mutual dependencies which might become a problem in practice. Lastly, there is no service from which all others are dependent.

The Grading Service and the Thesis Service both depend on the Message Service. This is because there are functionalities implemented in these two services that trigger the sending of an email. However, we think it is better to have this dependency instead of implementing the email functionality twice, especially because the Message Service has an external dependency on an SMTP server.

The Grading Service also depends on the Thesis Service. This is because of a check whether a user is authorized to update or write a grading proposal which applies to theses advisors and the professor. The dependency is however necessary and cannot be avoided since grading-related data has been viewed as critical data and should therefore be separated from the general thesis-related data. Hence, we cannot combine the Grading Service and the Thesis Service.

The Registration Service depends on the Meta-Data Service and the Thesis Service. This is because registration documents require both, information about the students as well as information about the thesis. Separating student-related data, which can again be considered as critical data, from thesis-related data is useful. However, we could argue that there is no need for the Registration Service. The functionality could also be implemented directly inside the Meta-Data Service because there are no other dependencies that would cause a conflict. However, since the functionality of the Meta-Data Service and the Registration Service are completely different, we keep them separated and instead accept the dependency.

Summary: To conclude this section, we provide Table 5.2 which lists all required microservices in alphabetic order in a tabular format for later reference. The last column indicates which functional requirements are covered by the respective microservice.

Service	Task	Covers
Auth Service	Handles OAuth token management	-
Eureka Service	Service discovery	-
Grading Service	Handles grading proposals	FR7, FR8
Message Service	Handles messages and emails	FR2, FR3
Meta-Data Service	Handles storing and querying of student information	FR1, FR8
Monitoring Service	Stores and evaluates Chairman events	-
Registration Service	Handles creation of registration documents and contracts	FR1
Thesis Service	Handles everything directly related to a student thesis	FR1-FR6, FR9, FR10
UI-Gateway Service	Gateway and Angular SPA	-

TABLE 5.2: All Microservices of Chairman

5.1.2 TECHNOLOGIES

For the implementation of Chairman we make use of quite many different technologies. For the individual microservices we use Spring Boot [39] because it simplifies web development in many situations and is perfectly suited for microservices. That is for instance because it creates executable `jar` files instead of `war` files. Hence, there is no need to deploy many different files to an application server like *Tomcat* [9], but instead only start multiple Java executables that can even run on different machines.

Netflix is famous for using microservices and fortunately, they have made many components open source¹. From their stack, we reuse *Eureka* for the service discovery, *Zuul* as the gateway and *Feign* as a REST client.

The UI part of Chairman is implemented as a *Single Page Application* (SPA) because this enables a fluid user experience. Hence, we use Angular [5] for the frontend.

Lastly, for all of the databases we use MongoDB [28]. In fact, it does not make much difference whether we use a document-based or a relational database in this case. However, we have decided to only use one type of database for the entire application.

¹see Github repository: <https://github.com/netflix>

5.2 GRAPHICAL DESIGN

A non-functional requirement of Chairman is *easy usability* (NFR5). Usability directly depends on the graphical design of an application since this is what the user sees and how the user interacts with a system. To meet the requirement, we should not implement any spectacular design features, but instead develop a standard, solid web frontend for Chairman.

As already known from Section 3.1.1, there are different types of users with different authorizations. The frontend must reflect that and provide only views to functions that the currently logged in user is allowed to execute. This is the implementation of the **Limited View** pattern, as introduced in Section 2.1.

We have agreed on a consistent design that has a navigation bar at the top of the page and the content directly below it. The bar itself should contain different tabs and a logout button always on the right side. Since Chairman is modelled around a student thesis, all important functionalities related directly to the thesis, should be accessible under the first tab. Secondary features, like e.g. downloading/uploading templates should be outsourced to other tabs.

For the three stakeholders *students*, *staff* and *professor*, we have created several mockups which illustrate how the final frontend should look like. These mockups are printed in the appendix of this Master's Thesis (Chapter B).

CHAPTER 6

IMPLEMENTATION

The fourth phase of the Software Lifecycle Model is called **Implementation**. In the previous chapters we have analyzed and designed Chairman in theory. What follows is the practical part which is described in this chapter.

6.1 OVERVIEW

In this section we give a general introduction to the practical part without going into specific details of individual components.

Each of the nine microservices, as listed in Table 5.2, is developed as a distinct Spring Boot application. In addition to that, we implement the frontend as a Single Page Application in Angular. The *git* repository of the project reflects that since each Spring Boot application and the frontend, respectively have received their own directories. The rest of this section only explains general information about the Spring Boot applications. The Angular part is exclusively described in Section 6.2.2.

In order to build and run the Spring Boot applications, one needs to have a working Java JDK installed on the development machine with a minimum Java version of 8. Apart from that, a MongoDB server is required.

The directories of the individual microservices follow the standard structure of Maven. Source code is always in *src/main/java/[PACKAGE_NAME]*, whereas unit tests can be found in *src/test/java/[PACKAGE_NAME]*. Spring Boot applications can be configured by either using *.properties* or *.yml* files. We have decided to exclusively use

YAML because it is more readable in our opinion. The configuration file is always called *application.yml* and is placed in *src/main/resources*.

We use Gradle as the build system. The Gradle dependencies are always placed in the top level *build.gradle* file and one should use the Gradle Wrapper for downloading dependencies and building the application. The final *.jar* file can be found in *build/libs* and started from the command line by typing *java -jar [FILENAME].jar*.

The source code directories are structured similarly most of the time, but can differ slightly in some cases. Each Spring Boot application has an *Application.java* file at the top which contains the *main* method that starts the application. Security configurations are always placed in */config*. REST controllers are either in */controllers* or */web/controllers*. The same applies to services which are either in */services* or in */web/services*. Repositories can be found in *repositories* and usually there are additional folders, like */utils*, */tools* or */models*.

The next section explains details of the individual Spring Boot applications and each part is structured equally. First, a short **Overview** is given which summaries the tasks of that specific application. What follows is a description of the **Characteristics** and **Configuration** details which describe how the respective application is currently configured. When running in production, some attributes might have to be adjusted. At the end of each part we provide a table that describes the **Interface**, hence the REST endpoints with the specific security constraints. For convenience, we also provide Table 6.1 as a lookup table in alphabetic order. The table gives information about where to find which microservice and lists a few specific characteristics. As already mentioned, the Angular part is described in Section 6.2.2.

Service Name	Section	Characteristics
Auth Service	Sec. 6.2.1.1	OAuth Auth., Eureka C., LDAP, MongoDB
Eureka Service	Sec. 6.2.1.2	Eureka Server
Grading Service	Sec. 6.2.1.3	OAuth Res., Eureka C., Feign, MongoDB, PdfBox
Message Service	Sec. 6.2.1.4	OAuth Res., Eureka C., Mail, Feign
Meta-Data Service	Sec. 6.2.1.5	OAuth Res., Eureka C., MongoDB
Monitoring Service	Sec. 6.2.1.6	OAuth Res., Eureka C., MongoDB
Registration Service	Sec. 6.2.1.7	OAuth Res., Eureka C., Feign, PdfBox
Thesis Service	Sec. 6.2.1.8	OAuth Res., Eureka C., MongoDB, Feign
UI-Gateway Service	Sec. 6.2.1.9	Zuul Proxy, OAuth Single Sign On, Eureka

TABLE 6.1: Spring Boot Applications Lookup Table

6.2 IMPLEMENTATION DETAILS

In this section we finally describe implementation details. The first part is about the individual Spring Boot applications, whereas the second part is about the Angular Single Page Application.

6.2.1 MICROSERVICES

6.2.1.1 AUTH SERVICE

Overview

The Auth Service is the OAuth authorization server which authenticates the users and creates access tokens. When running in production, the accounts should be taken from the chair's local LDAP server.

Characteristics

The Auth Service is configured as an Eureka client and as an OAuth authorization server. Furthermore, it is also configured as an OAuth resource server since it provides REST endpoints that should be secured. Currently, we have introduced a client with the name *acme* and the password *secret*. When running in production, these values should be changed to something that is not easily guessable. The created tokens are stored directly in memory.

The *spring-boot-starter-data-ldap* dependency is already added to the *build.gradle* file. However, the connection to the LDAP server is not implemented yet. Currently, we use a local MongoDB database in which several dummy accounts with different roles are stored.

In order to let external resource servers - which are nearly all microservices - retrieve information about the logged in user, we provide a special */user* REST endpoint. This approach is virtually standard and is for example also used by Facebook.

Configuration

```
spring.application.name:      auth-service
server.port:                  8092
server.contextPath:          /auth
eureka.client.serviceUrl.defaultZone: http://localhost:8081/eureka
```

The *contextPath* is required in this case in order to avoid cookie collisions when the UI-Gateway Service is also running on localhost.

Interface

note: all endpoints need to be prefixed with the *contextPath*

Method	Endpoint	Description	Roles
GET	/user	Returns information about the logged in user	all
GET	/details	Returns basic information about specific users	all
POST	/authLogout	Destroys the token	all

6.2.1.2 EUREKA SERVICE

Overview

The Eureka Service is a special microservice that acts as the service discovery for all other microservices. This simplifies the communication between the microservices and the Angular application since only the name needs to be known in order to communicate with microservices. To use the service discovery, all microservices need to be configured with a *name* and a *port number*. Furthermore, the *defaultZone* of the Eureka server has to be specified and the main class has to be annotated with *@EnableEurekaClient*.

Characteristics

The Eureka Service only consists of one Java file which is annotated with *@EnableEurekaServer*. The service requires the *spring-cloud-starter-eureka-server* dependency to be added to the *build.gradle* file. Anything else is automatically managed by Spring Boot.

Configuration

```
spring.application.name:      eureka-service
server.port:                  8081
eureka.client.serviceUrl.defaultZone: http://localhost:8081/eureka
```

The *defaultZone* is the URL which must be configured in all Eureka clients. If the clients are running on a different machine than the Eureka server, then the hostname has to be adjusted.

Interface

no REST endpoints

6.2.1.3 GRADING SERVICE

Overview

The Grading Service manages grading proposals in Chairman. Since final grades are considered to be critical data, we can not include this feature in the Thesis Service. In addition to that, the Grading Service should actually be deployed on a distinct machine.

Characteristics

The Grading Service is configured as an Eureka client and as an OAuth resource server. Hence, all REST endpoints are secured by default and only accessible with a valid token. The individual endpoints are further configured to be only accessible by the roles *staff* and *professor*. We have defined a *Grading* class which has all necessary attributes, like motivation, statement, grade and more. The objects of that class are persisted in a MongoDB database.

With the help of the *PDFBox* library - provided by Apache [8] - we can automatically create *pdf* documents for specific grading proposals. However, we have only implemented the basics for creating *pdf* files so far and leave improvements open for future work.

Adding or updating grading proposals requires that action to be made accountable. Therefore, we have implemented a Feign client that is able to interact with the Monitoring Service. The queries are created directly in the respective controller methods. All necessary Java files for the accountability feature can be found in the *monitoring* folder.

Configuration

spring.application.name:	grading-service
server.port:	8094
spring.data.mongodb.uri:	mongodb://localhost:27017/gradings
eureka.client.serviceUrl.defaultZone:	http://localhost:8081/eureka
security.oauth2.resource.user-info-uri:	http://localhost:8092/auth/user

Interface

note: all endpoints need to be prefixed with */api/grading*

Method	Endpoint	Description	Roles
GET	/ {id}	Gets a single grading from the database	staff ¹ , professor
GET	/file/ {id}	Downloads the grading in pdf format	staff ¹ , professor
POST	/	Creates a new grading in the database	staff, professor
PUT	/ {id}	Updates a grading in the database	staff ¹ , professor
DELETE	/ {id}	Deletes a grading in the database	staff ¹ , professor

6.2.1.4 MESSAGE SERVICE

Overview

The Message Service provides methods for sending automatically generated notification or reminder emails to students.

Characteristics

The Message Service is configured as an Eureka client and as an OAuth resource server. Hence, all REST endpoints are secured by default and only accessible with a valid token. As one can see below, the Message Service has actually only one REST endpoint which is further configured to be only accessible by the role *staff*. For sending emails it is required to add the *spring-boot-starter-mail* dependency to the *build.gradle* file and configure the *SMTP* server.

When a staff member adds a student to a thesis, the REST endpoint is automatically triggered which results in a notification mail for the respective student. In addition to that, the Message Server has a scheduled method which is called every day at 7.0 a.m. The method uses a Feign client to check for upcoming talk dates and submission dates in one week at the Thesis Service. If there are any, a reminder mail is sent.

¹ only possible if staff member is an advisor of that thesis

Configuration

```

spring.application.name:      message-service
server.port:                 8095
spring.mail.host:           smtp.net.in.tum.de
spring.mail.port:           10025
eureka.client.serviceUrl.defaultZone: http://localhost:8081/eureka
security.oauth2.resource.user-info-uri: http://localhost:8092/auth/user

```

Interface

Method	Endpoint	Description	Roles
POST	/api/mail	Sends a notification mail to the student	staff

6.2.1.5 META-DATA SERVICE

Overview

The Meta-Data Service manages personal information about students which are mainly required for filling out several documents, like contracts or registration documents. Since personal information is considered to be critical data, this microservice should actually be deployed on a distinct machine.

Characteristics

The Meta-Data Service is configured as an Eureka client and as an OAuth resource server. Hence, all REST endpoints are secured by default and only accessible with a valid token. Most of the endpoints are further configured to be only accessible with the role *student*. We have defined a class called *Student* which has all attributes that are required for registration documents and contracts, like first name, last name, address, etc. As always, we use MongoDB for the database. The Meta-Data Service is further able to output all valid courses of study for which registration documents can be created.

Configuration

```

spring.application.name:      meta-data-service
server.port:                 8091
spring.data.mongodb.uri:     mongodb://localhost:27017/metadata
eureka.client.serviceUrl.defaultZone: http://localhost:8081/eureka
security.oauth2.resource.user-info-uri: http://localhost:8092/auth/user

```

Interface

Method	Endpoint	Description	Roles
GET	/api/students/{id}	Gets a single student from the database	students ²
GET	/courses	Gets a map of all courses	all
POST	/api/students	Adds a new student to the database	students
PUT	/api/students/{id}	Updates a student in the database	students ²
DELETE	/api/students/{id}	Deletes a student in the database	students ²

6.2.1.6 MONITORING SERVICE

Overview

The Monitoring Service implements the demanded accountability feature of Chairman. Chairman events are actually created in other microservices, but the Monitoring Service is responsible for receiving and persisting these events.

Characteristics

The Monitoring Service is configured as an Eureka client and as an OAuth resource server. Hence, all REST endpoints are secured by default and only accessible with a valid token. MongoDB is again used for the database. We have defined a *LogEntry* class which has all the attributes, as defined in Figure 4.4. Each entry is further assigned a unique id that is created automatically. In order to guarantee that incoming POST requests do never overwrite existing entries, we let Spring Boot generate a new id each time. This way, one can only append entries to the database instead of updating existing ones.

Configuration

```

spring.application.name:           monitoring-service
server.port:                       8096
spring.data.mongodb.uri:          mongodb://localhost:27017/logs
eureka.client.serviceUrl.defaultZone: http://localhost:8081/eureka
security.oauth2.resource.user-info-uri: http://localhost:8092/auth/user

```

²only possible for own entry

Interface

Method	Endpoint	Description	Roles
GET	/api/logs	Gets all entries in the database	all
GET	/api/logs/{thesisId}	Gets all entries related to the specified thesis	all
POST	/api/logs	Appends a new entry to the database	all

6.2.1.7 REGISTRATION SERVICE

Overview

The Registration Service automatically fills out the official registration documents of the *Technical University of Munich* and the forms of the chair of *Network Architectures and Services*. In order to use that service, one first needs to provide information about the thesis (see Thesis Service), as well as personal information about the student (see Meta-Data Service). This information is then automatically fetched from the respective microservices. The filled out documents can be downloaded in *pdf* format.

Characteristics

The Registration Service is configured as an Eureka client and as an OAuth resource server. Hence, all REST endpoints are secured by default and only accessible with a valid token. The individual endpoints are further configured to be only accessible with the role *student* since student information is required for creating valid documents. We use a Feign client in order to fetch the information from other microservices. If no argument is specified at the REST endpoint, the service will process the registration document for the given course of study. For all other documents it is required to provide an argument as a path variable.

The document templates are placed in the *src/main/resources* folder and are filled out with the help of the *PDFBox* library provided by Apache [8]. We have introduced an abstract class (*AbstractForm.java*) that implements all fundamental functionalities. In order to process specific documents, one should inherit from that class. This design decision makes it easy to extend the Registration Service in the future with further documents. Currently, the following documents can be created:

Bachelor	Informatics Games Engineering
Master	Informatics Economic Computer Science Applied Computer Science Biomedical Computing Games Engineering Automotive Software Engineering
Forms	Reception Form Contract

Configuration

```

spring.application.name:      registration-service
server.port:                 8090
eureka.client.serviceUrl.defaultZone: http://localhost:8081/eureka
security.oauth2.resource.user-info-uri: http://localhost:8092/auth/user

```

Interface

Method	Endpoint	Description	Roles
GET	/forms	Creates the registration document	students
GET	/forms/{document}	Creates the specified document	students

6.2.1.8 THESIS SERVICE

Overview

The Thesis Service is the main service of Chairman which implements most of the functional requirements. Hence, it is also the most extensive microservice in terms of code size and complexity. All non-critical thesis-related information are managed here. Furthermore, the service administers file uploads and provides static files, like Latex templates or the guidelines document.

Characteristics

The Thesis Service is configured as an Eureka client and as an OAuth resource server. Hence, all REST endpoints are secured by default and only accessible with a valid token. Many of the individual endpoints are further configured to be only accessible with a certain role which is shown in the interface description below. As always, we use MongoDB for persisting thesis-related information and use Feign as a REST client in order to communicate with other microservices. The code is structured in a way that

follows the *Model-View-Controller* pattern as best as possible. This makes it easier to extend or maintain the code since different aspects are clearly separated.

The database persists objects of the *Thesis* class which has all necessary attributes like title, type, advisors, student, talk dates, registration dates, submission dates and more. Each thesis is further assigned a unique id that is formed by the title, the name of the advisor and the year. The id does not contain the student name since a student is added after the entry in the database has been created.

Uploaded files are stored as flat files on the file system in a directory which is named as the respective thesis id. Additionally, the files are referenced and extended with meta information in the database in order to enable efficient queries. All uploaded files are thoroughly checked before they are accepted. The checks include whether the mime types and the file extensions are correct, whether the file sizes are less than the maximum allowed and whether the files do not pose any security risk. Static files are also stored on the file system in plain format, but are not referenced any further.

Many of the REST endpoints require the action to be made accountable. Therefore, we have implemented a Feign client that is able to interact with the Monitoring Service. Most of the queries are created directly in the respective controller methods. All necessary Java files for the accountability feature can be found in the *monitoring* folder.

Configuration

```
spring.application.name:      thesis-service
server.port:                  8093
spring.data.mongodb.uri:     mongodb://localhost:27017/theses
eureka.client.serviceUrl.defaultZone: http://localhost:8081/eureka
security.oauth2.resource.user-info-uri: http://localhost:8092/auth/user
http.multipart.max-file-size: 5MB
```

The *max-file-size* specifies the maximum file size for uploaded files.

*Interface**Theses*

note: all endpoints need to be prefixed with */api/theses*

Method	Endpoint	Description	Roles
GET	/	Gets all theses from the database	all
GET	/id	Gets a single thesis	all
GET	/search/{advisorName}	Gets all theses of the specified advisor	all
GET	/search/status/{status}	Gets all theses with the specified status	all
GET	/search/student/{name}	Gets all theses of a specific student	all
GET	/search/shared/{name}	Gets all shared theses of a student	all
GET	/search/date	Gets all upcoming talk dates	all
POST	/	Creates a new thesis in the database	staff
PUT	/id	Updates a thesis in the database	staff, professor
PUT	/finish/id	Sets a thesis to read-only	professor
PUT	/unfinish/id	Makes a theses writable again	professor
DELETE	/id	Deletes a thesis in the database	staff

Files

Method	Endpoint	Description	Roles
GET	api/file/id	Downloads the specified file	all
POST	api/file/id	Uploads the specified file	students, staff
DELETE	api/file/id	Deletes the specified file	students, staff

Static files

note: all endpoints need to be prefixed with */api/static*

Method	Endpoint	Description	Roles
GET	/guidelines	Downloads the guidelines file	all
POST	/guidelines	Updates the guidelines file	staff
GET	/howto	Downloads the how-to file	all
POST	/howto	Updates the how-to file	staff
GET	/slides	Downloads the slides template (Powerpoint)	all
POST	/slides	Updates the slides template (Powerpoint)	staff
GET	/beamer	Downloads the slides template (Latex)	all
POST	/beamer	Updates the slides template (Latex)	staff
GET	/thesis	Downloads the thesis template (Latex)	all
POST	/thesis	Updates the thesis template (Latex)	staff

6.2.1.9 UI-GATEWAY SERVICE

Overview

The UI-Gateway Service is a special service which uses Zuul in order to act as the gateway between the Angular SPA and the other microservices. All requests from the SPA are filtered and proxied to the respective microservices. The exact addresses are resolved by the service discovery (see Eureka Service).

Characteristics

The UI-Gateway Service is not configured as an Eureka client. However, it still needs the *spring-cloud-starter-eureka-server* dependency to be added to the *build.gradle* file in order to resolve the addresses of the microservices for which it proxies incoming requests. The proxy functionality itself does not require any coding since we reuse *Zuul* for that job. The only task that needs to be done is to annotate the main class with *@EnableZuulProxy* and configure the Zuul routes. The routes are simply a mapping from the service name to a path value that is always named the same as the respective microservice.

The Angular SPA is served by the UI-Gateway Service, as well. To achieve that, we need to place the generated Angular files inside the *src/main/resources/static* folder.

The UI-Gateway Service also plays a special role for OAuth because it enables Single Sign On in Chairman. This requires the main class to be annotated with *@EnableOAuth2Sso* and the client credentials to be configured in the *application.yml* file. When a user now logs in to the Angular application, he first gets redirected to the Auth Service according to the OAuth protocol. After the protocol is finished, it is however not the Angular application that retrieves the token, but instead the UI-Gateway Service itself. In addition to that, the UI-Gateway Service generates a session for the user and returns a cookie that contains the session id. Since cookies are automatically sent by browsers, all incoming requests will contain that cookie. Zuul then only needs to filter the requests by exchanging the cookie with the correct token before forwarding the requests to the microservices. In order to log out from Chairman, it is now however not enough to destroy the token at the Auth Service. The session at the UI-Gateway Service has to be destroyed, as well.

Configuration

```

spring.application.name:          ui-gateway-service
server.port:                     8080
eureka.client.serviceUrl.defaultZone: http://localhost:8081/eureka
security.oauth2.client.accessTokenUri: http://localhost:8092/auth/oauth/token
security.oauth2.client.userAuthorizationUri: http://localhost:8092/auth/oauth/authorize
security.oauth2.client.clientId:   acme
security.oauth2.client.clientSecret: secret
security.oauth2.resource.user-info-uri: http://localhost:8092/auth/user

```

The `clientId` and `clientSecret` must match the specified values of the Auth Service. The specific Zuul routes are emitted here because the path is always named exactly as the service name.

Interface

no REST endpoints

6.2.2 ANGULAR SPA

The frontend files can be found in the *chairman-frontend* directory. In order to work with the Angular application, one needs to have *nodejs*, *npm* and *Angular CLI* installed on the development machine. An embedded testing server can be started by typing *ng serve* in the command line, whereas *ng build* builds the application. To use the UI-Gateway Service instead of the testing server, one needs to copy the generated files to the *static* folder of the UI-Gateway Service. Further information about how to use the Angular CLI can be found in the included *README* file.

The frontend directory has the standard structure of an Angular project and has been generated by the Angular CLI. Source code is placed in */src/app*. For each of the roles *students*, *staff* and *professor*, we have introduced a distinct Angular module. As modules are only loaded when needed, this design decision reduces the initial loading time drastically. Everything that is required in all three modules is placed in a separated *shared* module which is always loaded. The folders of the modules consist again of several subfolders for Angular components, services, pipes and more.

The overall goal when developing the frontend is to reconstruct the mockups, as printed in the appendix (Chapter B). We have introduced a separated Angular component for each visible tab. A component itself consists of three parts: an *HTML* part for defining the structure of the visible content, a *CSS* part for styling and a *Typescript* part for

asynchronously exchanging data with the microservices. For the CSS styling we make use of a styling framework, called *Semantic UI* [38].

As already implied in Section 6.2.1.9, a special part of the Angular application which requires an explanation is the login mechanism. Even before the user interface is loaded, the application tries to fetch the user information from the OAuth *user* endpoint¹. This request however fails when the user is not logged in yet. In this case the user gets redirected to the Auth Service according to the OAuth protocol. After the protocol is finished, the user returns to the Angular application and the *user* endpoint is again called. This time, the request contains a cookie which is replaced with a valid token by the UI-Gateway Service. This means that the Auth Service now answers successfully with the current user information. With that knowledge the application can load the correct Angular module.

6.3 DEPLOYMENT

In order to deploy Chairman, one needs to check out the git repository first and build all parts. The build process requires two steps: firstly, all microservices need to be built with the *Gradle Wrapper*. Secondly, the Angular app needs to be built with the *Angular CLI* and the generated files need to be copied to the *static* folder of the UI-Gateway Service. We have provided a bash script² which automates the necessary steps.

The individual microservices can either run all on the same machine or on multiple machines. The only requirements are a working *Java Runtime Environment* (JRE) with a minimum Java version of 8 and one or more MongoDB servers. The MongoDB servers do not necessarily have to run on the same machine than the microservices. For the start process, we have also provided a bash script³ which automates the job of starting all Java executables. However, the script starts all microservices on the same machine which might be inappropriate in production.

All Java executables have already an embedded configuration file, but the configuration can easily be overwritten by putting an *application.yml* file in the same directory as the respective *.jar* files.

¹proxied through the UI-Gateway Service

²build-all.sh

³run-all.sh

CHAPTER 7

EVALUATION

In this chapter we close the circle of Chairman and analyze whether we have reached the functional and non-functional requirements, as defined in Chapter 3.

7.1 IMPLEMENTATION OF THE FUNCTIONAL REQUIREMENTS

The large part of the functional requirements (see Table 3.1) is implemented in Chairman and is in a working state. This applies to FR1, FR2, FR4, FR5, FR6, FR7 and FR9.

FR3 is only partially implemented. Advisors are able to share certain theses with other students, but the comment function is currently missing. Since this feature is rather inessential, it is left open for future work. For the moment, advisors and students have to communicate with each other by using external technologies, like for example email or Slack.

FR8 is currently not implemented because it is unclear whether this functional requirement is actually necessary in practice. It does not seem so at the moment which is why it is left open for future work, too. However, this feature can easily be implemented at any time since it is basically the same as FR4 and FR6, only with a different storage location.

We are currently not able to implement FR10 because there is no appropriate infrastructure or connection to other systems available. From a technical point of view, it should be rather easy to implement this functional requirement. A *cronjob* that runs on a daily basis and uses *rsync* for synchronizing the storage directory of the Thesis Service with

a predefined directory should be sufficient. The storage directory of the Thesis Service is where all the uploaded results can be found.

As already explained in Section 6.2.1.1, the connection to the LDAP server is not implemented yet. Even though this is not directly listed as a functional requirement, it is still required when running in production. The MongoDB server instead should be emitted.

7.2 OBSERVANCE OF THE NON-FUNCTIONAL REQUIREMENTS

7.2.1 PRIVACY, SECURITY & ACCOUNTABILITY

To achieve privacy we have demanded to implement unlinkability, transparency and intervenability in a meaningful way (see NFR1).

Unlinkability is mainly achieved by the software architecture of Chairman. Critical data is outsourced to own microservices with distinct databases. This refers to the Meta-Data Service and the Grading Service. If a non-authorized person manages to get access to Chairman, he is still not able to collect *all* data due to that separation which is a big advantage for privacy. Chairman collects only personal information that is really necessary for filling out the registration documents and contracts which means that GDPR Art. 5, Par. 1 (c) [22] is implemented correctly. Unfortunately, registration documents require quite a lot of data. This is however not up to our discretion, but instead predetermined by the Technical University of Munich. After a thesis is finished, we should actually delete all personal information in order to fulfill GDPR Art. 5, Par. 1 (e) [22], which is currently not the case.

Transparency is implicitly given by this Master's Thesis which acts as the documentation of Chairman. Even though the source code is currently not publicly available, every user of Chairman can at least request *read* access to the source code. To improve transparency we should add more notifications to the frontend which enlighten the users about *when* and *why* critical data is processed. Additionally, we need to make sure to add an *imprint* and a *data protection statement* before running in production. This is required due to the *Telemediengesetz* [41] in Germany.

Intervenability is partly achieved. Every student can change or completely delete personal information about themselves. However, this is associated with a loss of functionality since a student then has to fill out registration documents and contracts manually. On the other hand, a student currently cannot prevent processing of the final grade.

To achieve security we have demanded to implement the security patterns, as defined in Table 2.1 (see NFR2).

The Single Access Point to the application is the UI-Gateway Service which routes all incoming requests to the respective microservices. In order to fully have a single point, it is required to configure all other microservices to be only accessible via the gateway. However, this can only be done when running in production.

The Check Point is the Auth Service that authenticates users, checks their authorizations and creates OAuth tokens, depending on the user's role. Currently, there are three different roles implemented: students, staff and professor.

The Session pattern is not implemented since the microservices are completely stateless and do not create or share a session. However, because Chairman uses a token-based approach for authentication, a session is in fact not necessary at all.

The Limited View Pattern is completely implemented in Chairman. There are three different frontends, one for students, one for staff members and one for the professor. Which frontend is loaded depends on the role of the currently logged in user.

Unfortunately, the Secure Access Layer Pattern is currently not implemented, but definitely required in production. In order to communicate via a TLS-secured connection, all microservices need to be equipped with a certificate. This is in fact one of the downsides of a distributed architecture like microservices because multiple certificates are required.

The accountability feature of Chairman ensures that important actions can be assigned to concrete users and theses. By collecting information like IP addresses and user agents, it is further possible to perform plausibility checks. Chairman events are stored in a MongoDB database which only allows to add new entries, but not modify or delete existing entries. However, this is only guaranteed because there is no functionality implemented that would modify or delete entries. If an attacker manages to get access to the database, he could tamper with the stored data. A blockchain would certainly be better in this case since it uses cryptographic means to ensure that entries in the blockchain can not be altered. Unfortunately, a blockchain is associated with a very high organizational and technical effort, which is why we have decided to use MongoDB in practice.

7.2.2 FLEXIBILITY, EXTENSIBILITY & MAINTAINABILITY

By using microservices we achieve a high flexibility, especially because we are reusing parts of the Netflix stack. Since we use *Eureka* for the service discovery, every microservice is configured as a Eureka client. As a consequence of that, Spring Boot automatically adds *Ribbon* - which is the load balancer of Netflix - to the classpath. This means that we only need to start multiple instances of one microservice in order to make use of the load balancer and handle big amounts of incoming requests. Whether this feature is actually required depends on the amount of incoming requests in production and cannot be answered during development.

Chairman is definitely implemented in a way that simplifies extending and maintaining the source code. The only microservice that is a bit more extensive is the Thesis Service. All others have a very small code base which makes it easy for developers to understand how the services are working. The only disadvantage of Chairman is that it makes use of quite many different tools and technologies a developer should know before doing maintenance work. The non-exhaustive list includes Spring (Boot), JPA, Jackson, Eureka, Feign, Zuul, Ribbon, MongoDB, OAuth, Angular and many more.

7.2.3 USABILITY

Just as it has been difficult to define *good* usability, it is difficult to evaluate it. In general, we think that Chairman is easy to use since the web interface has a clear structure due to the navigation bar consisting of several tabs. There are no hidden menus, all main functions are directly visible under the first tab and secondary features are outsourced to the following tabs. In addition to that, each user can only see what he is allowed to do according to his role.

However, Chairman has also potential for improvement since some error messages of the microservices are not sent to the SPA at the moment. This can lead to silent failures. If for instance a file upload fails due to the wrong file extension or because the thesis is already marked as finished, there is currently no way for a user to notice that. Another example is the creation of registration documents: if the user has not filled out personal information yet, the creation of the document fails silently. Hence, appropriate notifications would improve the usability of Chairman.

7.2.4 PERFORMANCE & AVAILABILITY

In order to evaluate performance and availability, we have built the Angular SPA with *production mode* enabled and started all microservices on a *Dell XPS 13* with an *Intel Core i5* processor and *8 GBs* of RAM. Beside of the usual background tasks of a computer, a MongoDB server and an instance of Firefox were running. The entire RAM usage was *6.6 GBs* directly after the start and *7.2 GBs* during operation. Presumably, we could waste less memory by deploying multiple microservices to the same Tomcat server, instead of directly using the embedded Tomcat servers of the Java executables. However, this would also mean that we lose some of the flexibility because it would no longer be possible to simply start a Java file on an arbitrary machine.

The start of the nine microservices took *2.30* minutes and we had to wait around *one* minute before all microservices have resolved the names of all others from the service discovery. Hence, the initial waiting time is a bit high. However, as this process is usually required only once, it does not really worsen the entire performance.

Due to the SPA, all user actions happen nearly instantly without any latency since only a few kilobytes have to be exchanged with the microservices. As the SPA has been built with *production mode* enabled, even the start of the application is really fast. Since these processes are required a lot, we can conclude that the overall performance of Chairman is pretty good.

One observable problem are forwarding errors which happen sometimes when a user action is triggered for the first time. Unfortunately, we do not know *when* and *why* exactly this happens. This problem should be fixed before Chairman is used in production to guarantee best user experience.

CHAPTER 8

RELATED WORK

In this chapter we analyze and compare existing software systems with Chairman that are similar, either from an architectural perspective or from a functional point of view.

8.1 ARCHITECTURAL RELATION: NETFLIX

In this thesis we have shown that a software architecture based on microservices fits well to the use case Chairman. Therefore, we take a deeper look at Netflix in this section because Netflix is famous for using microservices in production.

Netflix is a *video on demand* service that lets users stream movies and series to multiple devices, like smart TVs, game consoles, smartphones, tablets and more. The business model of Netflix consists of three different subscription models where a user needs to pay on a monthly basis. The most expensive model offers the best picture quality and allows to simultaneously stream to multiple devices. The first month is free in all three models. [30]

We see that Netflix is completely different from Chairman from a functional perspective. However, the software architecture is quite similar. Netflix has successfully moved from a traditional monolithic application to a microservices architecture, where the individual microservices are deployed to *Amazon Web Services* [15]. Typically, there are *tens of thousands* instances of microservices and *thousands* of Cassandra database nodes running simultaneously in the cloud [14]. A very positive aspect of Netflix is its favour to open source. Hence, a lot of their code and documentation can be found on

Github¹. Apart from that, they run a blog on the internet² and publish many of their presentation slides³. For that reason, we are able to reuse multiple parts of the Netflix stack in Chairman. This applies to Zuul, Eureka and Feign.

The objectives for the architecture of Netflix have been *scalability, availability, agility* and *efficiency*. On the other hand, the principles include *sharing* and *separation of concerns* [14]. Hence, the requirements are comparable to Chairman. The resulting architecture consists of several **Edge Services** (UI-Gateway Service in our case) that are the entry point to the individual microservices in the cloud [16]. The Edge Services, which obviously use Zuul, are used to filter incoming requests, perform authentication, make security checks and more [16]. This way, the microservices can completely implement functional details without any high overhead of technical aspects. This has also been a goal when designing the microservices in Chairman and is again comparable.

In 2016 Netflix has made up 35.2% of the entire downstream traffic in North America which shows that microservices can successfully be run in production [31]. Hence, we can draw the conclusion that Chairman is able to run successfully, too - even though, it is completely different from Netflix from a functional point of view.

8.2 FUNCTIONAL RELATION

In this section we take a look at two software systems that are similar to Chairman from a functional perspective.

8.2.1 ALFRESCO

Alfresco is a web-based document management system written in Java. Its business model consists of three modules which are the *Alfresco Content Services*, the *Alfresco Process Services* and the *Alfresco Governance Services* [1]. The modules can either be purchased separately or as a complete solution. A user can buy one out of three editions: the Starter, the Business or the Enterprise edition which differ in functionality, support and price. Apart from that, there is the free Community Edition with only slimmed functionality and no support. [1]

¹<https://github.com/netflix>

²<https://medium.com/netflix-techblog>

³<https://www.slideshare.net/>

The Alfresco Content Services module is the main module which offers the actual functionality that is required for a document management system. Users can for example share documents, work together on documents and access the documents from everywhere with a multitude of supported end devices. Alfresco does not only store the actual documents, but also uses a database for storing meta information. This way, it is possible to efficiently search for documents or query information about them. Restoring older versions of stored documents is possible, too. [1]

The Alfresco Process Services module extends the functionality with business processing capabilities. Business processes can be defined in the BPMN language and run by the open source workflow engine **Activiti**. The third module adds the functionality for records management. [1]

From a functional point of view, the Alfresco Content Services module is in fact comparable to Chairman because Chairman also manages documents. This applies for example to uploaded results, templates and registration documents. Like in Alfresco, we do not only store the actual documents, but also utilize a database for meta information, like file types or upload times. Sharing documents with others is possible, too. However, Chairman does not offer the possibility to work on shared documents together, but instead only grants read access. A functionality completely missing in Chairman is the restoration of older versions. This is in fact a useful feature that could be implemented in the future. Apart from functional details, Alfresco is also similar to Chairman because it uses the same tools and technologies, like Java, Spring and Angular [1].

By looking at the documentation of the Alfresco Content Services module, we can see that it has a *component-based* software architecture which is different in Chairman. The main components are the UI component, the Index component and the Database component. Apart from that, there are optional components which are added depending on the purchased edition. [2]

Since we have seen that Alfresco is comparable to Chairman from a functional perspective, we should analyze why we can not simply use Alfresco, but instead develop our own system. In fact, there are multiple reasons: Firstly, because Alfresco is either really expensive, or free but limited in functionality and support. Secondly, because Alfresco is not tailored to our use case. One should not forget that Chairman is modelled around a student thesis which includes managing documents, but is not limited to. On the other hand, Alfresco offers functionalities that are definitely not required in our use case and hence, unnecessarily inflate the software.

8.2.2 CAMPUS ONLINE

Campus Online - often written as *CAMPUSonline* - is a web-based management system which covers nearly all business processes of a university. It has been developed by the *TU Graz* in 1998 and is currently used by 36 different universities, including the *Technical University of Munich*. [13]

Campus Online provides a web interface and a database in which all kind of data is stored that is required for the business processes. Some example processes are application processes, examination management, rooms administration and many more. [13]

Campus Online is comparable to Chairman since it is also concerned with business processes of a university: in fact, *theses management* can be seen as only one process. Apart from that, it manages similar roles (students, employee, alumni) and is comparable from a technical point of view: it provides Single Sign On for multiple services and is completely web-based. [13]

Since it is comparable, we should analyze whether we cannot use Campus Online instead of Chairman. The main reason is that it is not tailored to our use case. A service that fits our use case could be integrated into Campus Online with high probability, but nevertheless, Campus Online offers many features which are definitely not required. Another problem is that it provides only one database for all processes. This is not enough for our privacy requirement since we have shown that we need a physically separated database in order to guarantee privacy even in cases when an attacker has access to the database.

Summary: To conclude this chapter, we can say that it is useful to analyze comparable existing systems and get ideas from them. However, most of the time these existing systems are not perfectly tailored to the required use case. Therefore, it often cannot be avoided to develop and implement a system by oneself.

CHAPTER 9

CONCLUSION

In this Master's Thesis we have dealt with a management system for student research projects. We have seen that the processes which are required for finishing a thesis successfully have become quite extensive and complex. Hence, we can no longer imagine doing a thesis completely without any supporting tool like Chairman.

Even though the first version of Chairman was a good idea, the implementation has been problematic in many ways. Therefore, we have decided to start from scratch and focus on scientifically sound methodologies for the second version of Chairman.

We have started with a requirements analysis during which the processes and the stakeholders have become visible. A special focus has been laid on privacy and security since Chairman has to deal with critical data like final grades and personal information. After the system had been analysed and designed in theory, we have implemented Chairman as a web based application, evaluated it and compared it with similar systems.

The crucial point which helps to fulfill most of the non-functional requirements like privacy and security is a wisely chosen software architecture based on microservices. The microservices approach gives us a perfect possibility to separate critical from non-critical data and even deploy the critical parts on isolated machines. This way, it is possible to apply extra security measures to the critical microservices and guarantee that even an attacker who has got access to the database is not able to see *all* data, but only parts of it. Furthermore, microservices help to structure the entire application in a meaningful way which is positive for maintainability, extensibility and flexibility.

We have demonstrated that accountability is required to show who is responsible for several actions and to prevent data misuse. Our implemented accountability mechanism is able to record important actions and assign them to concrete persons and theses. For persisting these information, we use a MongoDB database which is easy to implement in practice. However, it is actually required that all actions are not disputable. Currently, we cannot guarantee that property if an attacker manages to get access to the database as he could tamper with the stored data in that case. Instead, a blockchain has the demanded property. We have shown how to connect a blockchain-based backend with Chairman. The essential point is that we have created an accurate interface by defining the Monitoring Service which separates the accountability feature from the actual functionality and makes it hence easy to improve that aspect in the future.

The Chairman implementation delivered in this thesis is in a working state at the moment and most of the planned functionalities are already implemented. However, before Chairman can be run in production, several additional steps are required. Most important is to equip all microservices with a TLS certificate in order to establish encrypted connections between the microservices. It is actually rather useless to secure all REST endpoints with a token if the tokens are sent in plain format. To use the already existing accounts of the chair, it is further necessary to connect the LDAP server with Chairman. Rather secondary, Chairman can be improved in many places to fulfill the functional and non-functional requirements even better.

In addition to that, the accountability mechanism should be improved in the future by connecting a blockchain-based backend with Chairman. To do this, one should start by defining a suitable crypto schema in which the actual data is encrypted, but it is still possible to perform validation checks on non-critical meta information. Probably, there are also better alternatives than *public-key* cryptography where the keys are hardcoded in the respective Monitoring Services. This has to be investigated in the future.

CHAPTER A

ABBREVIATIONS

BPMN:	Business Process Model and Notation
CLI:	Command Line Interface
CSS:	Cascading Style Sheets
DRY:	Don't Repeat Yourself
DSM:	Dependency Structure Matrix
FR:	Functional Requirement
GDPR:	General Data Protection Regulation
HTML:	Hypertext Markup Language
JSON:	JavaScript Object Notation
LDAP:	Lightweight Directory Access Protocol
NFR:	Non-Functional Requirement
PDF:	Portable Document Format
REST:	Representational State Transfer
RQ:	Research Question
SHA-3:	Secure Hash Algorithm 3
SMTP:	Simple Mail Transfer Protocol
SPA:	Single Page Application
TLS:	Transport Layer Security
UI:	User Interface
URL:	Uniform Resource Locator
XML:	Extensible Markup Language

CHAPTER B

GRAPHICAL MOCKUPS

B.1 STUDENT MOCKUPS

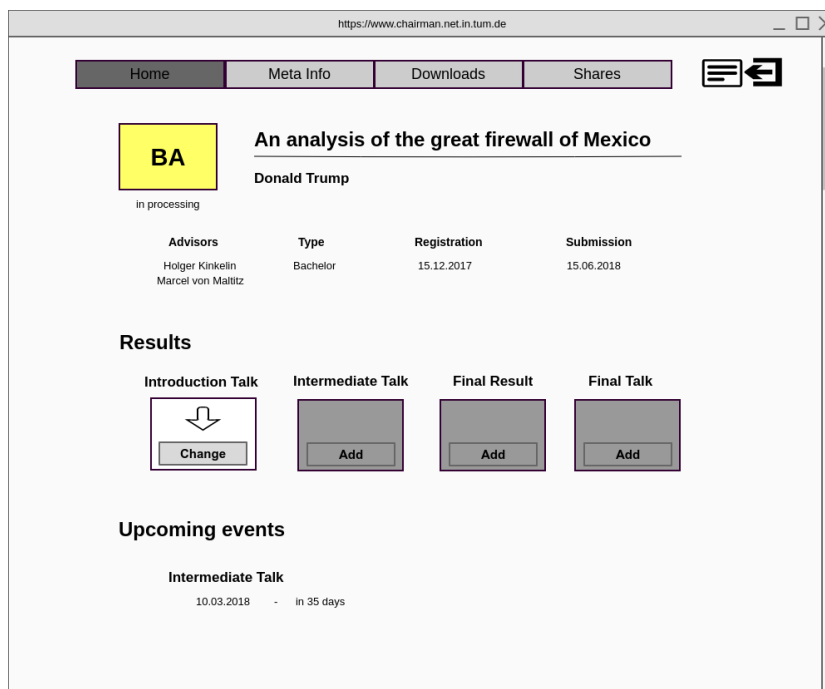


FIGURE B.1: Students - Home

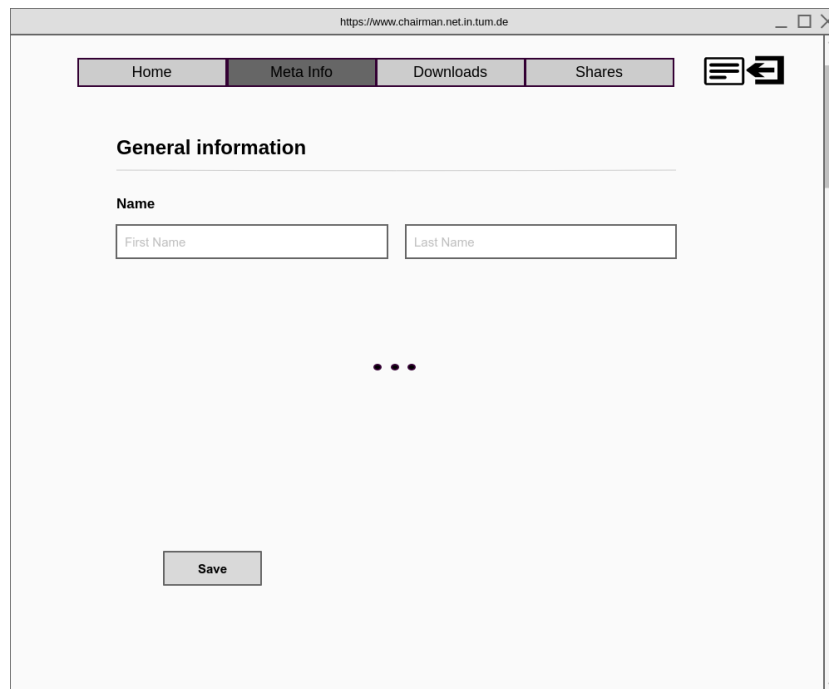


FIGURE B.2: Students - Meta Data

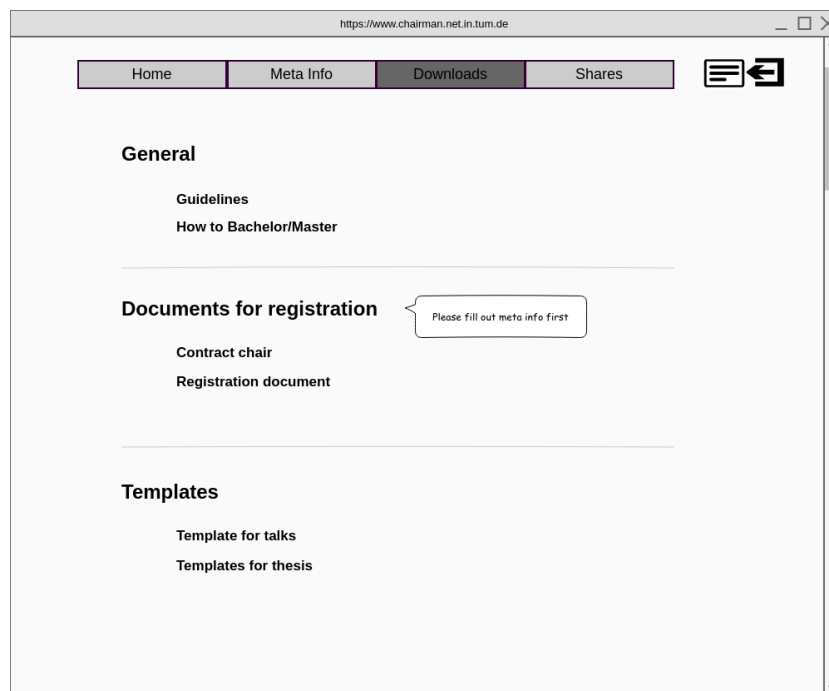


FIGURE B.3: Students - Downloads

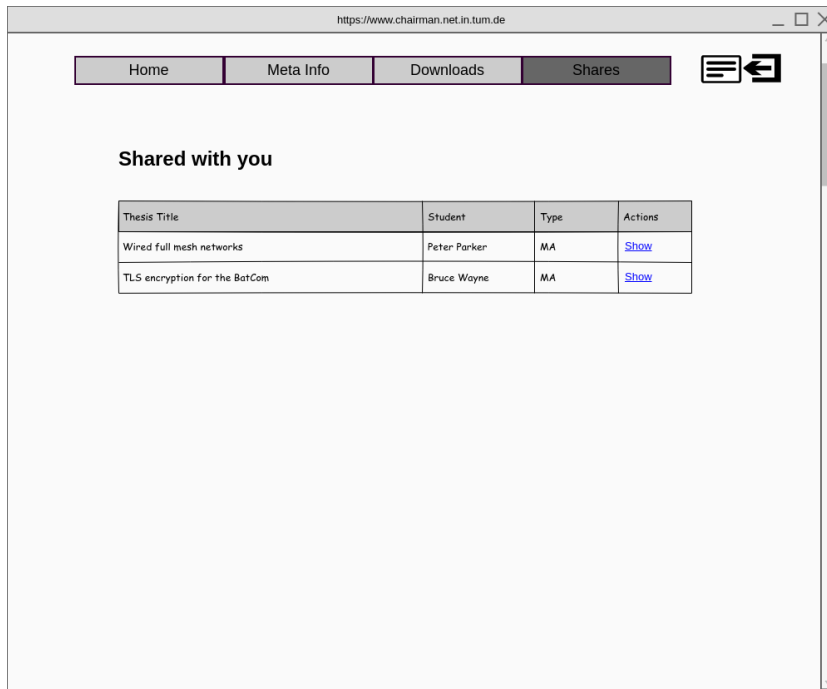


FIGURE B.4: Students - Shared Theses

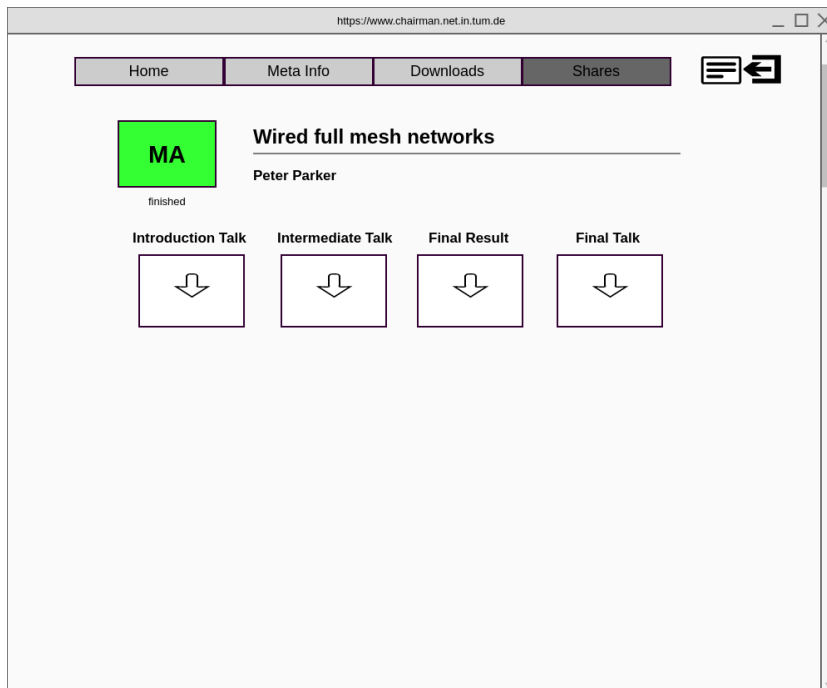


FIGURE B.5: Students - Shared Thesis Details

CHAPTER B: GRAPHICAL MOCKUPS

B.2 STAFF MEMBER MOCKUPS

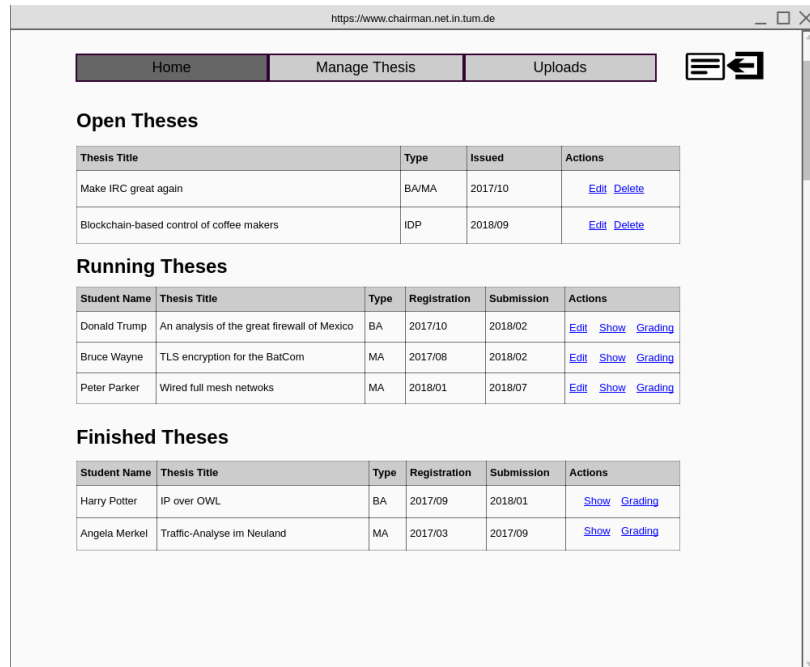


FIGURE B.6: Staff Members - Home

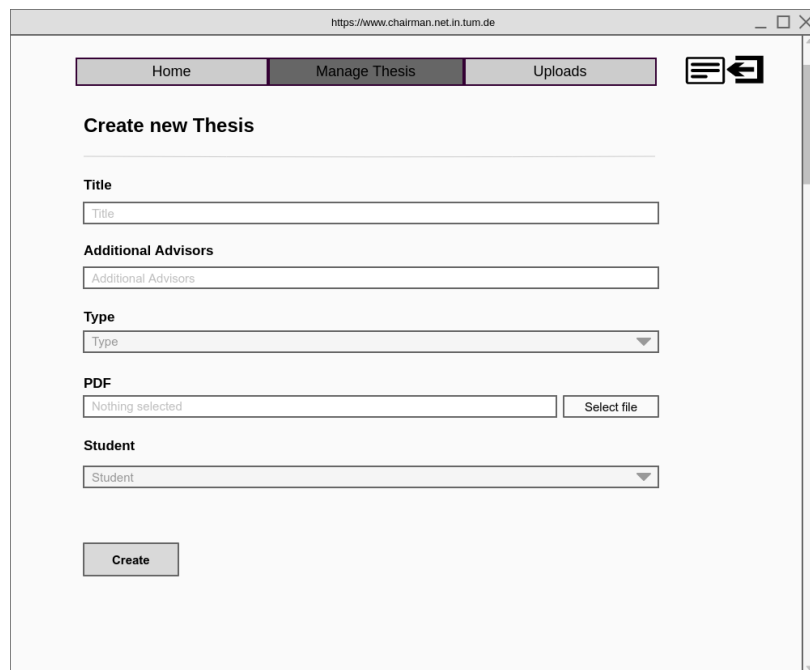


FIGURE B.7: Staff Members - Create new Thesis

B.2 STAFF MEMBER MOCKUPS

https://www.chairman.net.in.tum.de

Home Manage Thesis Uploads

Edit Thesis

Title
Make IRC great again

Additional Advisors
Marcel von Maltitz

...

Talks
2017/09/10 2017/12 .Final Talk

Registration
2017/10 2018/02

Share with other students
Student

Save

FIGURE B.8: Staff Members - Edit Thesis

https://www.chairman.net.in.tum.de

Home Manage Thesis Uploads

MA
finished

Wired full mesh networks

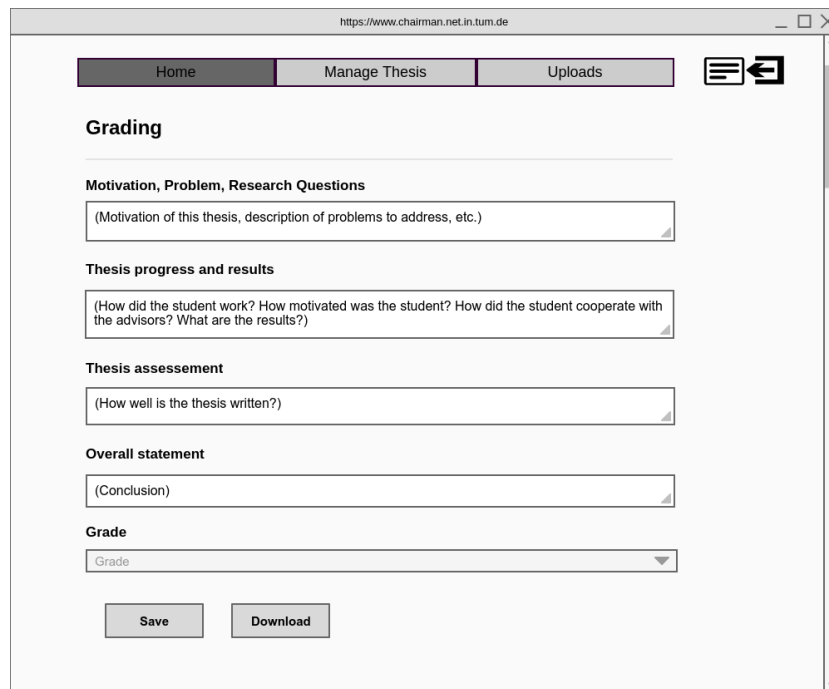
Peter Parker

Introduction Talk Intermediate Talk Final Result Final Talk

↓ ↓ ↓ ↓

FIGURE B.9: Staff Members - Show Thesis Details

CHAPTER B: GRAPHICAL MOCKUPS

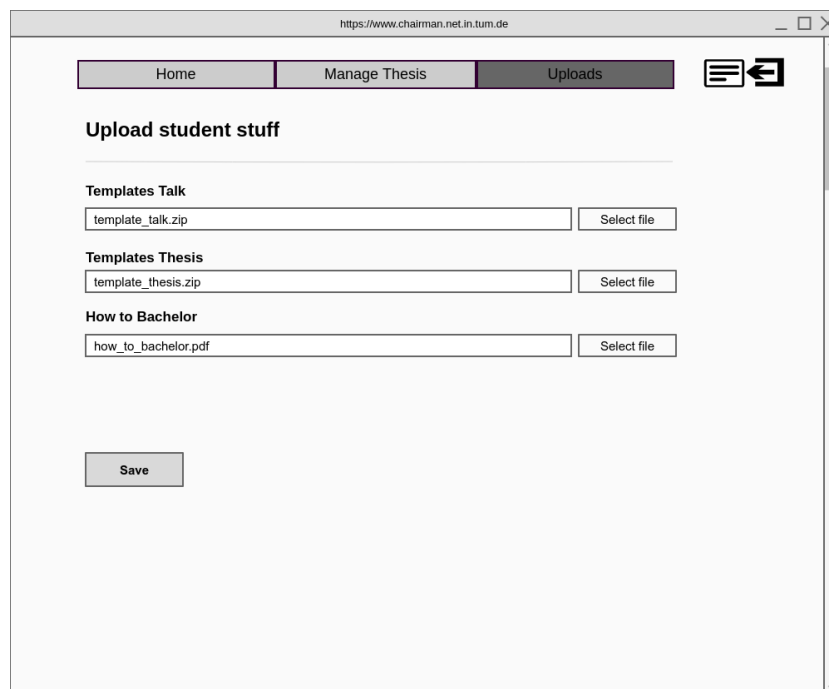


The screenshot shows a web browser window with the URL `https://www.chairman.net.in.tum.de`. The navigation bar contains three tabs: "Home", "Manage Thesis", and "Uploads". The "Grading" section is active. It features several text input fields for providing feedback and a grade dropdown menu. The fields are labeled as follows:

- Motivation, Problem, Research Questions**: (Motivation of this thesis, description of problems to address, etc.)
- Thesis progress and results**: (How did the student work? How motivated was the student? How did the student cooperate with the advisors? What are the results?)
- Thesis assesement**: (How well is the thesis written?)
- Overall statement**: (Conclusion)
- Grade**: A dropdown menu currently showing "Grade".

At the bottom of the form, there are two buttons: "Save" and "Download".

FIGURE B.10: Staff Members - Grading



The screenshot shows a web browser window with the URL `https://www.chairman.net.in.tum.de`. The navigation bar contains three tabs: "Home", "Manage Thesis", and "Uploads". The "Upload student stuff" section is active. It features three file upload sections, each with a text input field and a "Select file" button:

- Templates Talk**: Input field contains "template_talk.zip".
- Templates Thesis**: Input field contains "template_thesis.zip".
- How to Bachelor**: Input field contains "how_to_bachelor.pdf".

At the bottom of the form, there is a "Save" button.

FIGURE B.11: Staff Members - Uploads

B.3 PROFESSOR & SECRETARY MOCKUPS

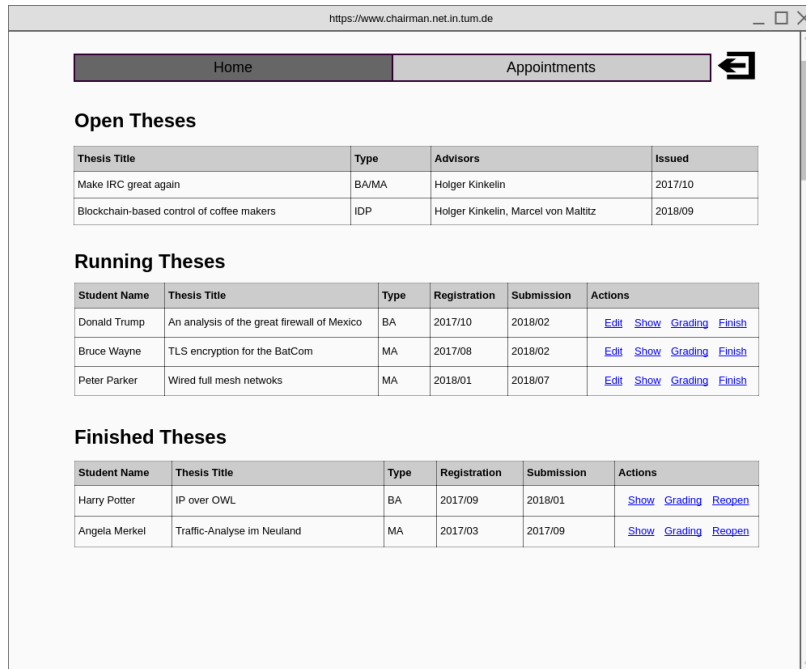


FIGURE B.12: Professor & Secretary - Home

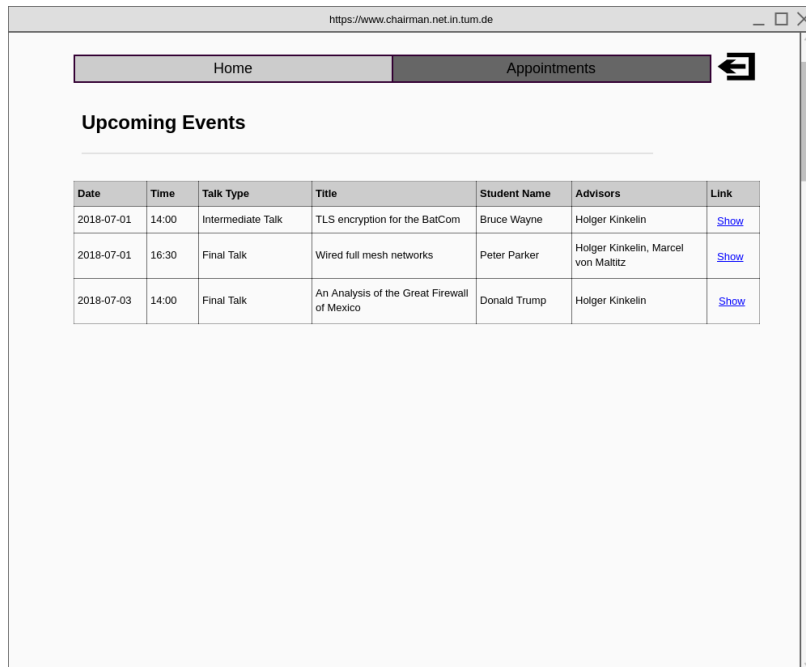


FIGURE B.13: Professor & Secretary - Appointments

CHAPTER B: GRAPHICAL MOCKUPS

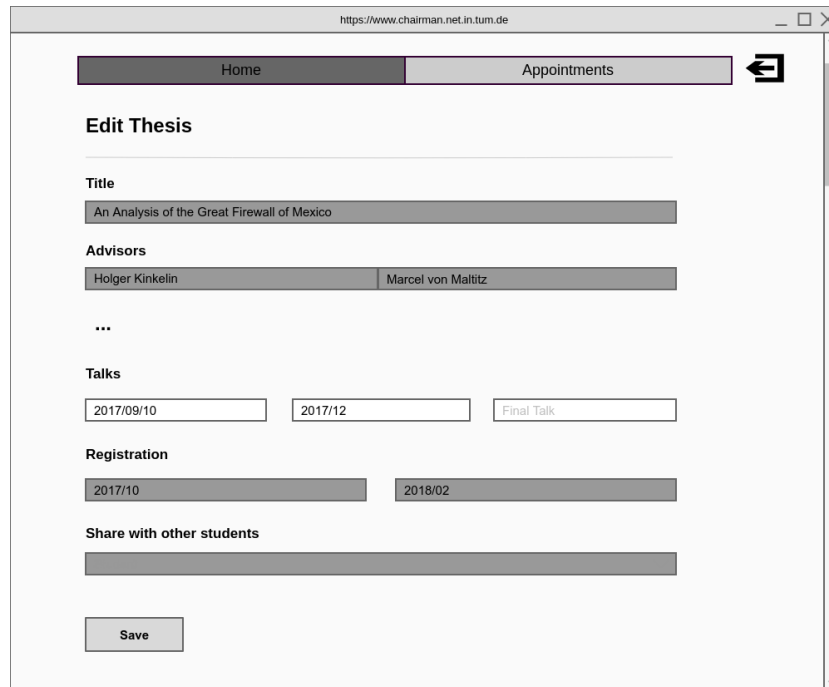


FIGURE B.14: Professor & Secretary - Edit Thesis

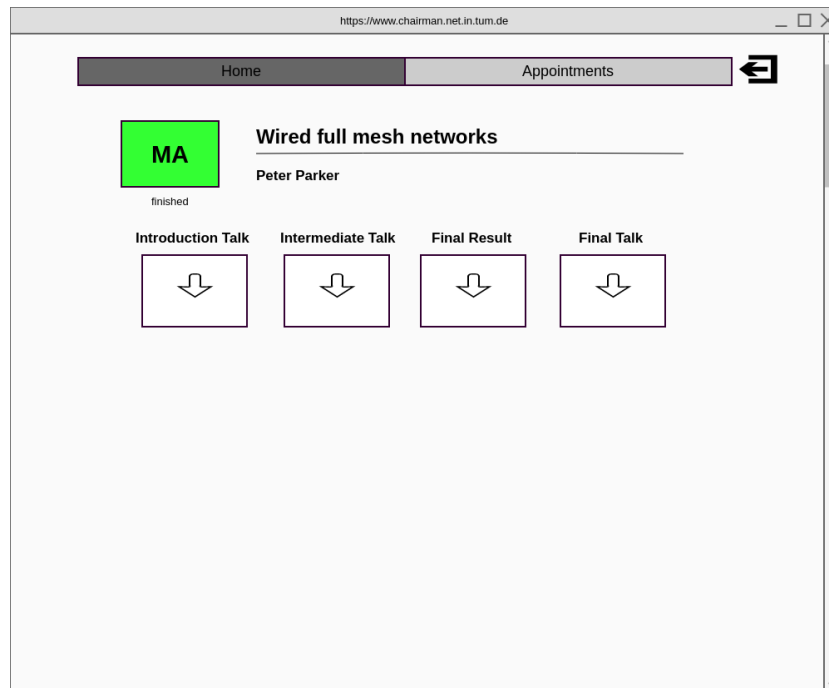


FIGURE B.15: Professor & Secretary - Show Thesis Details

B.3 PROFESSOR & SECRETARY MOCKUPS

https://www.chairman.net.in.tum.de

Home Appointments

Grading

Motivation, Problem, Research Questions

(Motivation of this thesis, description of problems to address, etc.)

Thesis progress and results

(How did the student work? How motivated was the student? How did the student cooperate with the advisors? What are the results?)

Thesis assesement

(How well is the thesis written?)

Overall statement

(Conclusion)

Grade

Grade

Save Download

FIGURE B.16: Professor & Secretary - Grading

BIBLIOGRAPHY

BIBLIOGRAPHY

- [1] *Alfresco*. Website. <https://www.alfresco.com/de/>, Accessed: 2018-06-12.
- [2] *Alfresco Content Services 5.2 On Premises - Reference Architecture*. Technical White Paper. https://www.alfresco.com/sites/www.alfresco.com/files/alfresco_content_services_5.2_reference_architecture.pdf, Accessed: 2018-06-12.
- [3] *Android*. Website. <https://www.android.com/>, Accessed: 2018-06-12.
- [4] Elli Androulaki et al. “Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains”. In: *ArXiv e-prints*. Jan. 2018.
- [5] *Angular*. Website. <https://angular.io>, Accessed: 2018-06-12.
- [6] *Angular CLI*. Website. <https://cli.angular.io>, Accessed: 2018-06-12.
- [7] *AngularJS*. Website. <https://angularjs.org>, Accessed: 2018-06-12.
- [8] *Apache PDFBox - A Java PDF Library*. Website. <https://pdfbox.apache.org/>, Accessed: 2018-06-12.
- [9] *Apache Tomcat*. Website. <https://tomcat.apache.org/>, Accessed: 2018-06-12.
- [10] *AUTOSAR - The standardized software framework for intelligent mobility*. Website. <https://www.autosar.org/>, Accessed: 2018-06-12.
- [11] Paris Avgeriou and Uwe Zdun. “Architectural Patterns Revisited - A Pattern Language”. In: *Proc. 10th European Conference on Pattern Languages of Programs (EuroPLoP 2005)*. UVK Konstanz, 2005.
- [12] Aleksander Berentsen and Fabian Schär. *Bitcoin, Blockchain und Kryptoassets*. 1st ed. BoD - Books on Demand, Norderstedt, Jan. 2017.
- [13] *Campus Online*. Website. <https://campusonline.tugraz.at/>, Accessed: 2018-06-12.
- [14] Adrian Cockcroft. *Patterns for Continuous Delivery, High Availability, DevOps & Cloud*. Presentation Slides. <https://www.slideshare.net/adrianco>, Accessed: 2018-06-12.
- [15] Adrian Cockcroft. *The Global Netflix Platform*. Presentation Slides. <https://www.slideshare.net/adrianco/global-netflix-platform>, Accessed: 2018-06-12.

BIBLIOGRAPHY

- [16] Mikey Cohen. *Netflix's Global Cloud Edge Architecture*. Presentation Slides. <https://www.slideshare.net/MikeyCohen1/edge-architecture-ieee-international-conference-on-cloud-engineering-32240146>, Accessed: 2018-06-12.
- [17] George Danezis et al. "Privacy and Data Protection by Design - from policy to engineering". In: *Technical Report (ENISA)*. 2015.
- [18] Erik Ernst. "Separation of Concerns". In: *Proceedings of the AOSD 2003 Workshop on Software-Engineering Properties of Languages for Aspect Technologies (SPLAT)*. Boston, MA, Mar. 2003.
- [19] *EU Charter of Fundamental Rights*. Website. https://ec.europa.eu/info/aid-development-cooperation-fundamental-rights/your-rights-eu/eu-charter-fundamental-rights_en, Accessed: 2018-06-12, PDF accessible at <http://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:12012P/TXT&from=EN>.
- [20] David Garlan and Dewayne Perry. "Introduction to the Special Issue on Software Architecture". In: *IEEE Transactions on Software Engineering*. Apr. 1995.
- [21] David Garlan and Mary Shaw. "An Introduction to Software Architecture". In: *V. Ambriola and G. Tortora (eds), Advances in Software Engineering and Knowledge Engineering, vol. 2*. World Scientific Publishing Company, 1993.
- [22] *General Data Protection Regulation*. Website. <https://gdpr-info.eu/>, Accessed: 2018-06-12.
- [23] D. Hardt. *The OAuth 2.0 Authorization Framework*. RFC 6749 (Proposed Standard). Internet Engineering Task Force, Oct. 2012. URL: <http://www.ietf.org/rfc/rfc6749.txt>.
- [24] Martin Hitz and Behzad Montazeri. "Measuring Coupling and Cohesion in Object-Oriented Systems". In: *Proc. Int'l Symp. Applied Corporate Computing*. Monterrey, Mexico, Oct. 1995.
- [25] Craig Larman and Victor R. Basili. "Iterative and Incremental Development: A Brief History". In: *IEEE Computer*. June 2003.
- [26] T. Lodderstedt, M. McGloin, and P. Hunt. *OAuth 2.0 Threat Model and Security Considerations*. RFC 6819 (Informational). Internet Engineering Task Force, Jan. 2013. URL: <http://www.ietf.org/rfc/rfc6819.txt>.
- [27] Michael S Mikowski and Josh C Powell. "Single page web applications". In: *B and W* (2013).
- [28] *MongoDB*. Website. <https://www.mongodb.com/>, Accessed: 2018-06-12.
- [29] Fabrizio Montesi and Janine Weber. "Circuit Breakers, Discovery, and API Gateways in Microservices". In: *arXiv preprint arXiv:1609.05830*. 2016.
- [30] *Netflix FAQs*. Website. <https://help.netflix.com/de/node/412>, Accessed: 2018-06-12.

- [31] *Netflix online traffic volume in North America*. Website. <https://www.statista.com/statistics/245986/netflix-share-of-peak-period-downstream-traffic/>, Accessed: 2018-06-12.
- [32] *OAuth 2.0*. Website. <https://oauth.net/2/>, Accessed: 2018-06-12.
- [33] *OAuth Security Advisory: 2014.1 "Covert Redirect"*. Website. <https://oauth.net/advisories/2014-1-covert-redirect/>, Accessed: 2018-06-12.
- [34] Ali Ouni et al. "Web Service Antipatterns Detection Using Genetic Programming". In: *Proceedings of the 2015 on Genetic and Evolutionary Computation Conference, ser. GECCO'15*. 2015.
- [35] Martin C. Robert and Martin Micah. *Agile Principles, Patterns, and Practice in C#*. 1st ed. Prentice Hall, July 2006.
- [36] Martin Rost and Andreas Pfitzmann. "Datenschutz-Schutzziele - revisited". In: *Datenschutz und Datensicherheit (DuD)*. 2009.
- [37] W.W. Royce. "Managing the Development of Large Software Systems: Concepts and Techniques". In: *Proc. WESCON*. Aug. 1970.
- [38] *Semantic UI*. Website. <https://semantic-ui.com/>, Accessed: 2018-06-12.
- [39] *Spring Boot Reference Guide*. Website. <https://docs.spring.io/spring-boot/docs/2.0.2.RELEASE/reference/htmlsingle/>, Accessed: 2018-06-12.
- [40] *Spring: the source for modern java*. Website. <https://spring.io/>, Accessed: 2018-06-12.
- [41] *Telemediengesetz*. Website. <https://www.gesetze-im-internet.de/tmg/>, Accessed: 2018-06-12.
- [42] *What is Subversion?* Website. <http://svnbook.red-bean.com/en/1.7/svn.intro.whatis.html>, Accessed: 2018-06-12.
- [43] Eberhard Wolff. *Microservices: Flexible Software Architecture*. 1st ed. Addison-Wesley Professional, Oct. 2016.
- [44] Joseph Yoder and Jeffrey Barcalow. "Architectural Patterns for Enabling Application Security". In: *Proc. 4th Conference on Pattern Languages of Programs (PLoP 1997)*. Monticello, IL, USA, 1997.