Czech
Technical
University
in Prague

System for management of company training

Tsimafei Raro

Supervisor: doc. Ing. Ivan Jelínek, CSc.
May 2020

Acknowledgements

I would like to thank ČVUT for being a fantastic *alma mater*.

Declaration

I declare that I am the sole author of the following project.

Abstract

This project attempts to simplify the management of educating employees of small-scale engineering companies. The problem it attempts to solve is simplifying and streamlining the process of managing lectures held within a company. Four popular Learning Management Systems were analyzed, from which it was concluded that there is a lack of small-scale e-learning applications focused on data management that also can integrate with other systems commonly found within software companies, such as LDAP and Git.

A custom solution was designed and built on the basis of this research, which consists of a web client written in Angular, and a server written in Java, which is also connected to a relational database. The solutions does its best to adhere to current standards of good architecture and good security. A suite of automated tests was created to make sure the application remains in working condition during development. Manual testing was also conducted and showed several problems in its initial design, which were subsequently successfully resolved. The resulting application is in a working state, however there are more improvements planned for the future.

Keywords: web application, lms, e-learning, java, spring, angular

Supervisor: doc. Ing. Ivan Jelínek, CSc.

Abstrakt

Cílem projektu je zjednodušení managementu učení zaměstnanců malých software engineering firem. Konkrétněji, problém, který zkouší být vyřešen, je zrychlení a zjednodušení procesu archivace lekcí čtených uvnitř firmy. Čtyři populárních Learning Management Systému byly zanalýzované, a jako důsledek bylo usouzené, že současně existuje nedostatek malých e-learning aplikaci s fokusem na správu dat a integraci s dalšími populárními nástroje, které jsou často použité uvnitř firem (např. LDAP a Git).

Vlastní aplikace byla navržena a implementována na základe provedené rešerší, která se skládá z web klientu v Angular, serveru v Java, a relační databáze. Vytvořena aplikace se stará odpovídat současným standardům dobré softwarové architektury a zabezpečení. Sada automatizovaných testů byla vytvořena pro zajištění funkčností aplikace během její vývoje. Také bylo provedeno manuální testování, které ukázalo několik problému v původním návrhy aplikace, které následovně byly vyřešeny. Výslední aplikace je ve funkčním stavu, ale současně existují i plány na možné zlepšení její funkcionality v budoucnu.

Klíčová slova: webová aplikace, lms, e-learning, java, spring, angular

Contents

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Raro Tsimafei**	Personal ID number:	**474678**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Computer Graphics and Interaction**		
Study program:	**Open Informatics**		
Branch of study:	**Computer Games and Graphics**		

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

System for management of company training

Bachelor’s thesis title in Czech:

Systém pro správu podnikových školení

Guidelines:

Design and implement an interactive web application for managing business training:
Research existing e-learning tools to manage business training and justify the need for a new solution.
Define the technical specification of the solution and design the application architecture with different access modes.
Create an application framework: connect the application to the enterprise database, formulate basic server logic, define a functional user interface.
The solution must include security according to company standards.
Implement the whole system, formulate tests.
Implement the application, test it in practice and evaluate the result.

Bibliography / sources:

Nicholas S. Williams (2014) Professional Java for Web Applications, John Wiley and Sons
Craig Walls (2011) Spring in Action, Manning, Edition 3
Martin, R.C. (2017) Clean Architecture: A Craftsman's Guide to Software Structure and Design, Pearson Education
Gierke, Darimont, Strobl, Paluch, Bryant (2019) Spring Data JPA Documentation, Online, Available at:
https://docs.spring.io/spring-data/jpa/docs/2.2.3.RELEASE/reference/html/referen
ce
Alex et al. (2019) Spring Security Documentation, Online, Available at:
https://docs.spring.io/spring-security/site/docs/5.2.1.RELEASE/reference/htmlsin
gle

Name and workplace of bachelor’s thesis supervisor:

doc. Ing. Ivan Jelínek, CSc., Center for Software Training, FEE

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **31.01.2020** Deadline for bachelor thesis submission: ___________

Assignment valid until: **30.09.2021**

doc. Ing. Ivan Jelínek, CSc.	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature		Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student’s signature

Chapter 1

Introduction

1.1 Preface

The inspiration for the subject of the project has been taken from the internal workings of the software engineering company called **Quanti s.r.o.** I, Tsimafei Raro, am the sole author of the project. Nevertheless, it must be stated that I am an employee of the aforementioned company.

1.2 The problem

During the last year, the aforementioned company has been trying out a system which allows employees to hold unsupervised lectures about professional subjects within the company. These lectures are held for other employees in the company, which basically makes it an internal education system. Outside of sharing profession experience, the core idea of this system is to strengthen interpersonal relationships between the employees, and to allow them to practice and develop their presentation skills.

The employees of the company are divided into groups based on their occupation. These groups are called *guilds*. There is, for example, a "Java guild", which houses all of the programmers working with Java or JVM-based programming languages. There is a "Tester guild", which is composed of most of the company's testers. Originally, guilds contained all of the similarly-occupied employees of the company. However, as time went on and the system evolved, the company has switched guilds to be more "opt-in". Now, for example, the Java guild does not hold all of the company's Java programmers, but rather only those interested in sharing their knowledge.

Within the guilds, *lectures* are held on a regular basis. Before a lecture, guild's members hold a vote on a subject for it. A subject is usually relevant to the guild member's daily job: it could be something someone has been working with in the last couple of weeks, and wants to share the acquired know-how. After choosing a subject and agreeing upon a time and date, the member of the guild who was most interested in lecturing about the subject is tasked with researching and presenting it to the rest of the guild. During the following lecture, this employee presents the subject, gives an explanation of why it might be useful, gives examples of their own experience with the subject, and then answers questions at the end of the presentation.

Despite the fact that different employees take the role of a lecturer for each lecture, organization of lectures is usually done by a single person that does not change (referred to as the *moderator*). Moderator's task is to keep their guild up and running. This includes keeping a list of potential topics for a lecture up to date, holding votes on next lectures, sending reminders to guild's other members about upcoming lectures, and making sure past lectures are archived so the knowledge is preserved and easily accessible. All of this is done by hand using an array of different tools and systems (for example, one of the guilds uses Google Sheets to keep a list of potential topics and to vote on them, an internal wiki system for archiving past lectures, and Slack and email messages for notifications).

This kind of management tends to become tedious and can eat up a lot of moderator's time. This leads us to the main problem this particular project will attempt to solve: to make the work moderators do a little bit easier. The way this is intended to be done is by creating a unified system, which would remove the necessity of using many different tools to organize lectures, and to automate discovering and archiving lectures, providing lecturers with feedback, and notifying guild members about upcoming lectures they could attend.

Due to the fact that the problem is tangentially linked to E-learning as a whole, research of existing E-learning solutions will be done first. This way, we will verify that currently no solutions exist which fit the problem in quite the way we want. Due to that, we will end up designing and implementing a custom application.

To give a quick summary, the following are the overarching steps that the development of our custom solution, and of this whole projects, could be divided into:

1. Researching existing E-learning solutions.
2. Designing the application.
3. Implementing the foundation of the application.
4. Implementing the application itself.
5. Testing and deployment of the application.

Before we take a look at the first of those steps, it should also be noted that the application which we have developed during this project is not considered to be "finished". There is room for future improvement, which will be discussed in more detail in chapter 9.

Chapter 2

Analysis of existing Learning Management Systems

This part of the research has been done as a part of the semestral project[6]. Therefore, this whole chapter will contain only a short summary of the work done during that phase of the project.

The first step of the project was evaluating whether the problem at hand was not solved by someone else already. The problem is related to the field of *e-learning*, of which a short description is necessary. For the sake of our project, E-learning will be understood as the particular type of learning that relies on using computers and software applications. The kind of software that is specifically aimed at managing e-learning is called *Learning Management Systems* (LMS for short). The method chosen for deciding whether our problem was already solved or not was to take a look at several distinct LMS, which are still being kept up-to-date, and evaluate, based on a certain set of criteria, whether they fit our problem or no.

Before taking a look at the actual LMS evaluations, let us define the set of criteria which will be used. To be a "good enough" fit for us, a potential LMS must meet the following requirements:

- **Be regularly updated** — working with an outdated LMS would very likely become a waste of time, as it would not integrate properly with other modern software.
- **Have a web GUI** — Software developers are frequently working on

various operating systems, which makes portability an important concern.

- **Provide the ability to store lectures with additional data** — Lectures usually produce additional data in the form of presentations, lecturer's notes, and videos.
- **Provide different authorities to different user groups** — Users are separated into application's administrators, guild moderators, and regular employees. Having fine-grained control over what they should be able to do is paramount.
- **Provide a modern web API** — There should be a way to expose information stored within the application to other applications.
- **Support LDAP authentication** — LDAP is a popular solution for centralizing user authentication, which is frequently used within software engineering companies.
- **Allow users to leave feedback on lectures** — Lecturers are regular employees, so it would be beneficial if they could give each-other feedback.

In the beginning, there was an additional requirement for the ability to send customizable e-mail notifications. However, as the project evolved, it turned out that this was not as important as initially though. Therefore, it is no longer considered in the evaluation of the LMS.

By taking the "be regularly updated" and "have a web GUI" requirements together (which are the broadest of them all), four LMS were chosen for evaluation. These are *Moodle*, *OpenOLAT*, *Canvas*, and *Sakai*. What follows is a short summary of the analysis of each of them.

Moodle is one of the most popular learning management systems. Its particular strength is the strikingly large ecosystem of plugins, which means that there is a lot of potential for customizing it.

Moodle fits most of our requirements: it has a robust system of storing content, provides a fairly granular REST API, allows users to be authenticated via LDAP, and allows dividing users into several groups with different levels of authorization. Nevertheless, Moodle does not integrate well with JIRA or Git, as there were no fitting plugins found. It also does not allow users a simple way to leave feedback on individual lectures. This is because it is oriented more towards the traditional studying system akin to how learning is managed in a university: long-running courses, tests, and clear separation of teachers and students.

Because of those shortcomings, Moodle was deemed not suitable enough for our problem.

OpenOLAT was initially developed in University of Zurich, and is currently provided as an open-source LMS. It provides much of the same functionality as Moodle: its content storing system is good enough for our requirement of archiving lectures, it provides LDAP authentication, and it exposes a REST API. However, it suffers from the same drawbacks as Moodle: at its core, OpenOLAT is oriented towards traditional learning, which means that it lacks the frictionless two-way communication that we would like to have between the teachers and the students. Alongside that, OpenOLAT has a smaller community than Moodle, which means that it's probably going to be more difficult to work with, as both the support and the documentation are likely to not be very robust.

Canvas is the third LMS on our list of potential solutions. Same as OpenOLAT, it is available as an open-source solution at GitHub. it's built on Ruby on Rails, and provides LDAP authentication. However its integration capabilities are fairly lacking, so it has been deemed not fit for our problem.

Sakai is the last LMS that has been evaluated before deciding that enough research has been done to make a decision about developing a custom solution. Its main strength is solid performance when serving huge numbers of concurrent users. But due to the fact that we are looking for solutions for small to medium-sized companies, this is not a particularly important point. Sakai also does not provide a good way for teachers and students to communicate with each-other.

2.1 Research conclusion

To give a short summary, none of the chosen LMS were deemed a good enough fit for our particular problem, despite successfully covering most of our requirements. All of them had similar problems, which originated in the fact that most of the existing LMS are geared towards traditional long-term learning rather than simpler, short-term management that we are interested in. They frequently provide complicated systems such as management of tests, exams, and whole forums for communication, which are all unnecessary for us.

One more problem of the existing solutions is that making modifications to them is cumbersome or impossible, because they are either proprietary soltuions, or the open-source codebase is expansive and would require significant effort to get accustomed to. Seeing how the concept of guilds has changed even during the last several months, the ability to make quick modifications to our system would be welcome, which makes working with an existing solution undesirable. Given all of that, a custom learning management application is a better choice than trying to adapt an existing LMS to our needs.

Chapter 3

Technical specification of the project

As has been shown in chapter 2, some of the more popular LMS solutions which already exist do not fit our requirements. It is possible that some smaller, less well-known solution would fit better. However, further research has been deemed not valuable enough to continue. Therefore, building a custom implementation becomes our next most reasonable choice. This chapter will go over the very beginning of this process – the design of the application.

Our own custom application must obey the requirements for the existing LMS solutions which were set in chapter 2. The only exception from this is the necessity of the LMS to not be outdated. This is not of a concern in our case, as we can choose any up-to-date technologies for the application and support it for as long as is necessary. Given that, we are left with the following set of problems our application must solve:

- Provide graphical web-based interface for the end user.
- Allow lecture archiving.
- Allow users to be given different roles which govern access to information.
- Provide a way for users to suggest future lectures.
- Provide a way for users to give feedback on past lectures.
- Integrate with the existing software used within the company.

3.1 Technology stack

With the slightly modified list of requirements being made clear, the next logical step in the designing process is to choose the technologies which will be used to implement our custom solution.

Because of the necessity of a web-based user interface and the potential benefits of exposing a custom web API, it would make sense to separate the web-based client from the server and use the web API for communication between them. This particular idea will be discussed in more details in section 3.3, where we will discuss the architecture of the application.

The core technologies for the server and the client were chosen in the previous phase of the project [6]. To give a short summary, the application uses the following list of technologies:

- **Java 11** is the core language for the server. It has been chosen due to its robustness and maturity in the context of developing web and enterprise applications.
- **Apache Maven** is used to manage server's dependencies on external libraries and to manage compilation, building, and packaging the application.
- Several **Spring** frameworks have been heavily used on top of Java to streamline the development of a web application. Most notably, **Spring Boot** is used to handle most of the Spring configuration, **Spring Data JPA** is used to handle the Data Access layer of the application, and **Spring Security** is used to handle user authentication and authorization.
- **MariaDB** was chosen as the underlying database for the server due to its maturity and personal previous experience with using it.
- **Angular 8** is the main driving technology for the client.
- The **Bootstrap** library provides the foundation of the visual styles for the UI.
- Lastly, **Git** is used as the version control system to handle the development process of both the server and the client. Moreover, the server integrates with Git's internal tag system to provide a good solution for application versioning. More on this will be said in section 4.4.2.

3.2 Domain of the problem

3.2.1 The domain

The application we are trying to develop is intended to be used to manage real-life ideas: lectures. One of the first and obvious realizations is that we will need a way to represent these ideas within the application. To do this, we need to define a set of *entities*, each having a set of properties and a set of relationships to other entities. Specifying the ideas we will be working on in these terms will allow us to easily map them to tables in a relational database (which, as will be discussed below, was chosen in favor of a non-relational database). And other than that, this also aligns nicely with the use of Java as the primary programming language: we will be able to leverage Java's objects-oriented nature.

The most important entities are **Lectures**, **Guilds**, and **Users**. Lectures are the core concept of the application – a single Lecture is defined by a title, date when it was held, description, and status. It's related to a single Guild where it was held, has a single User lecturer and a single Meeting Room. Alongside that, it has many (i.e. indeterminate amount which can be more than one) User attendants, many Attachments, and many Comments. Guilds model the professionally-linked group of users, and are defined by their name and two flags: confidential and archived. A Guild has many User members and a single User moderator. User models the user of the application, who is simultaneously an employee of the company. User is defined by a username, a name, and an email address. User can be a member of many Guilds, and can simultaneously moderate many Guilds and attend many lectures.

The secondary entities, which are necessary for the workings of the application, but do not hold so much meaning within the domain of the problem itself, are **Attachments**, **Meeting Rooms**, and **Comments** – all three relating to Lectures – as well as **Roles**, which are related to Users. Attachments are external files which can be linked to a Lecture. Meetings Rooms model the physical rooms where meetings and lectures are generally held. A Lecture may have a relation to a Meeting Room. Comments model simple text-based feedback which Users may leave many comments on Lectures. Roles hold very little meaning in the actual domain, and are rather a convenient entity which simplifies persisting information about User's individual permission levels.

3.3 Architecture

3.3.1 Three-tier architecture

The problem of maintainability is ever-present in the process of developing any software. It arises in the early stages of designing the architectural backbone of an application to the late stages of deployment and support.

Our application lends itself nicely to being separated into three large pieces: the client, the server, and the database. This division is commonly known as *three-tier architecture*: the client, also referred to as the *Presentation layer*, concerns itself with presenting data to the end user. The server (the *Application layer*) handles complex data processing which is relevant to the domain of the problem. Lastly, the database (the *Data layer*) handles data persistence and long-term storage.

The reasoning for separating the server and the database should be fairly obvious – the database is already running as a separate application which we will not be developing. Therefore, we will now focus on explaining the reasoning for separating the client and the server.

As already stated, the server will be running on Java 11 with the addition of Spring. The web user interface could have been written using, for example, JavaServer Pages without any separation from the server. This would have been simpler than creating a separate client application in so far as not requiring creation of a separate application and setting up communication between the client and the server. So why has this choice been made? The reasoning for opting for a completely separate client is that this made the final project more modular. It opens the possibility for creating, for example, a mobile application, which would simply connect to the server as the existing client, and there would be absolutely no need to implement any changes on the server.

The consequence of this decision is that there must now exists an interface through which the client and the server will be communicating. This was achieved by making the server expose a web API. In particular, a REST-like API was implemented. There are several web API standards gaining in popularity today, (for example, *JSON:API* or *GraphQL*), but those have not been considered due to the relatively small scale of the application. The particulars of the implementation of this web API will be discussed in more details in section 4.2.

3.3.2 Layered approach to server architecture

The high-level division of the application into the client, the server, and the database cleanly separates presentation logic from business logic and data storage. However, we have now created two separate applications (plus the database, which is basically a third application). We are not solving the database's architecture, as this was done for us by its developers, but we are solving the architecture of the client and the server, which now both have to be reasonably well structured so they are maintainable in the future.

The server must solve several problems:

- Communicating with the database to persist and fetch domain-related data.
- Creating ways to modify data in a way that makes sense within the domain and strictly enforcing that all data modifications are valid.
- Providing access to both the domain data itself and to ways to modify it to external users (i.e. the clients).

These tasks are not unique to our application – indeed, they are generic enough that virtually every web application must solved them. It therefore should come as no surprise that there are well-established ways to deal with them, which have been tested by time. These conventions can be given a wrapper name: *layered architecture*. Just as we have separated our formerly ill-defined "application" into "client", "server", and "database", we can further divide the server into *layers*, which are all mostly isolated from each-other and solve their own unique sets of problems:

- **Controller layer** – this layer's main responsibility is exposing data to other applications. In the context of a web application's server, this usually means providing a set of special endpoints, listening for incoming requests on all of them, deserailizing them, routing them deeper into the application, and lastly handling serialization and sending of the responses when they are ready.

- **Service layer** – the layer responsible for processing of the actual data. In a certain way, this layer is the "heart", as this is where most of the actual business logic unique to the application's particular domain is held. After the controllers get a hold of a request, the request is sent down into the service layer, where it is further processed. If necessary, this layer further communicates with the Data access layer to fetch necessary resources or persist freshly modified ones.
- **Data access layer** – this deepest layer concerns itself with mapping the data structures and domain entities produced by the Service layer to the underlying data storage of the application. It also provides an interface for the Service layer to access the data stored in the database. In our case, the Data Access layer handles ORM (object-relational mapping – in our case, mapping of in-memory Java objects to database rows and SQL queries) and connection to the relational database.

There is also a separate concept of *domain entities*, which are the internal representations of the data which the application operates with (a concrete representation of what a "lecture" or a "guild" is, which we can work with). It doesn't map to any single of the aforementioned layers, but we could make an argument that the domain entities are most closely related to the Service layer.

This layer-based division is quite effective in keeping small and medium-sized applications maintainable, as it enforces clean and strict structure to the codebase.

3.3.3 Client – component-based architecture

The client handles the presentation of the data to the end user. The problem of presenting data is not quite as well-divisible into layers as the problems that the server was handling. In the end, the division of the client is similar to the MVC (Model-View-Controller) pattern, which is a fairly popular way of managing user interfaces. This pattern divides the application into three isolated parts:

- The **View**, which is the interface which the user sees.
- The **Model**, which is the data and the business logic which exist behind the scene.

- The **Controller**, which is the mediator between the two – reacting to the user's interactions with the View and transforming them into requests for the Model.

The architecture of the client is similar to MVC, but not exactly that, as the Model in this case is the server itself, leaving only the View and the Controller on the client. The View, in this case, becomes the collection of styled HTML templates, and the Controllers are the collection of TypeScript classes linked to those templates. Angular handles these by aggregating them into something called *components* – an HTML template and a TypeScript code-behind bundled together. This approach is not exactly perfect, as the View and the Controller become tightly coupled with each-other, however it works well enough within our application to warrant not solving that particular architectural problem.

The client is built from these components, which model individual pages the user can visit. Some components model not whole pages, but rather smaller reusable pieces of the interface – like the pagination module, which is used across most of the tables on the server. Those smaller components are then used within other, larger ones.

3.3.4 Data storage

The last big piece of the application is the data storage. We cannot rely on keeping data purely in server's internal memory – that way, every restart of the server would completely wipe it all out. That's reason enough to justify the necessity of a database.

The question then becomes whether to use a relational (such as PostgreSQL or MariaDB) or a non-relational database (such as MongoDB). Non-relational databases are typically better suited for storing dynamically structured data, which would otherwise be troublesome to fit into a strict definition of tables in a relational databases. But as should be already evident from the specification of the domain entities, our application's data is pretty well-structured. Thus, the choice to use a relational database was made, because not only does it fit our data definition, but we would also be able to enforce strict constraints on all of our entities. To be more specific, *MariaDB* was chosen due to its good balance between performance and ease of use, overall maturity as a relational database solution, and my personal professional experience in working with it.

Chapter 4

Server implementation

4.1 Implementing layered architecture

The foundation of the server was implemented in the semestral project [6]. Each of the main domain entities were divided into a group of core classes, which belong to different architectural levels of the application:

- *Domain entity* itself, which is an internal representation of a single real-world idea (like a *Lecture* or a *Guild*).
- **Controller**, whose purpose is to handle HTTP requests relevant to the given entity.
- **Service interface and implementation**, which handles any in-between business logic like filtering collections of entities, validating the results or preparing entities to be persisted.
- **Data access object** (also simply called DAO), which handled getting data from the database and mapping them to domain objects within the application. This is almost entirely handled by Spring's Data JPA library [3].

Additionally, as the application grew in complexity, most entities have also been enriched by Data transfer objects and entity mappers.

Data transfer objects (DTOs for short) are classes whose purpose is to be a temporary container for data. These could be used to store data in the middle of a long processing chain or to define the data structure for HTTP responses. The latter use case is the one most prominent in our application. But why are DTOs useful for handling HTTP requests and responses? It would certainly be simpler to directly serialize the internal domain objects and send the result to the client. However, there is a problem with doing that: the internal domain objects frequently do not contain the data which is required by a client. Moreover, some of that data can change depending on who is accessing. For example, lectures have a boolean field called `attending`, which signifies whether the currently accessing user is attending the given lecture. This is a field which does not exist in the domain entity itself (where the same information is persisted as a special relationship between the Lecture and the User entities).

If DTOs, in the context of processing HTTP requests, are how we specify how the received and sent data should look, then **entity mappers** are responsible for transforming data between DTOs and internal domain objects. In our applications, an external library called *Mapstruct* was used to facilitate implementation of entity mappers.

It should also be noted that not all parts of the server's logic are concerned with handling domain entities. Security, for example, necessitated the creation of several configuration classes and web filters, which for the sake of keeping the code clean were isolated to a separate package.

4.2 REST-like API

The server exposes a web API, which can be consumed by a "good" client. What does "good" mean? In the case of this application, this means that it is properly authenticated and is capable of consuming JSON-based data.

Given the fact that web APIs are a popular way of letting applications communicate with each-other, many conventions arose about how they should be structured and implemented. One of them is REST, at which we will be taking a closer look.

Representational state transfer (REST for short) was first defined by Roy Fielding in his PhD dissertation *Architectural Styles and the Design of Network-based Software Architectures*. Modern web APIs can not adhere to

REST at all, adhere to it partly (then they are commonly called *REST-like* APIs), or adhere to it completely (in which case they are referred to as *RESTful* APIs). In our particular case, our web API belongs to the second group – it is REST-like.

Why is that? To answer this question, let's quickly take a look at the definition of REST. Per R. Fielding, REST is a set of architectural constraints for a web service [5]. These constraints include things like *statelessness* and *cacheability*. However, we will be interested only in the constraint called *uniform interface*. An interface, which is what the client uses to consume information from the server, is defined as *uniform* if it satisfies the following properties:

1. **Resources are identified in requests** – clients should be able to identify a resource from a URI contained within the request.
2. **Resources are manipulated through their representations** – given the representation of a resource (what the client retrieves from the server), clients should be able to modify the resource itself (the data on the server).
3. **Messages are self-descriptive** – all representations of resources which are returned to the client should also contain information on how to process them.
4. **Hypermedia as the Engine of Application State (HATEOAS)** – client should be able to discover actions which it could take on a representation of a resource given only that representation. To put it simply, this means that the server should provide clients not only with data, but also with explicit instructions about the actions which can be taken on that data.

Interface in the context of our application is the web API. It is straightforward enough to prove that our web API obeys the first three properties. All resources can be accessed via `GET` requests to endpoints of the `/api/<resource-name>/<resource-id>` format. Because of this, the URI is enough to unambiguously identify any resource, which satisfies the first property. After obtaining a representation of a resource, client can modify it by sending `POST`, `PUT`, or `DELETE` request on the same endpoints, which means that the second property is satisfied. All responses from the server contain a `Content-Type` header, which allows the client to recognize how the response should be parsed. Therefore, the third property is respected.

However, our web API does not adhere to the last property. To explain why, a quick summary of what HATEOAS is is necessary. *Hypermedia* *as the Engine of Application State* is a convention, also first defined by R. Fielding, about how to ensure the discoverability of actions a client can take when working with an API. To put it simply, an API adhering to HATEOAS principle would, on request, provide not only a representation of the requested resource, but also a set of hypermedia links (i.e. URLs), giving the client a list of actions they can now do with the data. It's going to be more easily understandable with a concrete example. Here is what a response which adheres to HATEOAS might look like:

```
GET http://localhost:8080/api/users/15
Accept: application/json


200 OK
Content-Type: application/json

{
  "data": {
    "id": 15,
    "username": "neal.wisozk",
  }
  "links": {
    "guilds": "http://localhost:8080/api/users/15/guilds"
    "lectures": "http://localhost:8080/api/users/15/lectures"
  }
}
```

Contrast that with a response which does not adhere to HATEOAS:

```
GET http://localhost:8080/api/users/15
Accept: application/json


200 OK
Content-Type: application/json

{
  "id": 15,
  "username": "neal.wisozk",
}
```

In the first case, it's clear that the retrieved user also has collections of "guilds" and "lectures", which can be retrieved by appending `/guilds` or `/lectures`, respectively, to the base user URI.

An API which does not adhere to the HATEOAS principle requires additional external documentation for clients to be able to effectively work with it (not to say that HATEOAS-adhering APIs don't need documentation – they certainly do, but it becomes even more vital when client cannot infer future actions from server's response).

The decision to not adhere to HATEOAS was conscious. Given the fact that our application is relatively compact and is unlikely to scale up to support a wide variety of clients, investing additional time and resources into adding proper HATEOAS support was judged as unnecessary.

4.2.1 Controller layer as the implementation of the API

Now that we have established that the exposed API will be only REST-like and not entirely RESTful, let us take a look at the process of implementing it.

Most of the API is built within the Controller layer, where individual Controller classes model semantically-linked sets of endpoints. Controllers handle exposing individual endpoints, mapping incoming requests to DTOs which can be used by the server, and returning correct responses to the client.

"Semantically-linked sets of endpoints" in our case means a group of endpoints which are related to a single resource. For example, endpoints for which process `GET`, `POST`, `PUT`, and `DELETE` requests for lectures are all contained within a single `LectureController` class. Such grouping is the natural and default way of implementing endpoints in a Spring web application. It also provides the benefit of keeping relevant things close together in the code – which makes further development easier and reduces the chance of introducing bugs into the application when adding a new feature or modifying an existing one.

There is one thing which is not modeled by a Controller – that is the authentication endpoint, which is handled by a separate web filter. More on that in chapter 6.

4.2.2 Exposing data via JSON endpoints

Clients require a standardized format for responses they expect to get from the server. For ease of implementation, JSON was chosen as the only data format for the API.

Given the fact that most of the API works with resources, we can define several standard situations which are commonly processed by the server:

1. Client requests a single domain entity
2. Client requests many similar domain entities
3. Client requests data which is not a domain entity (i.e. metadata)

Client requests a single domain entity

The first case is the simplest one. Each domain entity which is available through the API has a single DTO, which defines how a publicly available representation of the given domain entity should look. When client requests data about a particular domain entity (by accessing the URI which uniquely identifies is), the server:

1. Fetches the domain entity from the underlying database.
2. Maps it to an internal representation.
3. Fetches and processes any additional required data (for example, reads file contents of an Attachment from the filesystem)
4. Maps all of the collected data to a DTO.
5. Finally, serializes DTO to JSON, constructs an HTTP response, and sends it back to the client.

Therefore, the process of exposing a single resource via the API is fairly straightforward.

Client requests many similar domain entities

The second case is slightly more complex. The algorithm remains similar to fetching a single domain entity, however additional complexity arises from the new concepts of *filtering* and *pagination*.

Very frequently, client does not know the exact IDs of the domain entities it needs (and indeed may not care about IDs at all), but instead has a list of conditions which all of the entities must satisfy. A good example would be "lectures in the Java Guild containing the word 'security' in the title". This leads to a necessity of implementing a way to not only return many resources simultaneously, but also a way to for the client to specify filtering criteria.

In our application, client can specify these criteria via HTTP query parameters. If we assume that the Java Guild is known by the application as "guild with ID = 5", then the client can request all lectures in the Java Guild with the word 'security' in the title by sending a GET request to `/api/lectures?guild=5&search=security`. The internal workings of how filtering is implemented on the server will be discussed in section 4.3.1.

Pagination is the second complication which comes into view only when dealing with collections of resources. Pagination is a way of splitting a single collection into several "pages". This is done because processing whole collections has severe drawbacks:

- The server may not have enough processing power or memory to prepare a response with a very large collection.
- If the server can successfully respond with a very large collection, the payload of such a response will be very large and thus will take a long time for the client to download.
- Even if the server can respond with a very large collection and the payload is successfully downloaded, the client can also experience problems with insufficient memory or processing power.

Given the fact that speed and responsiveness of the application is of great importance to the end user, it is indeed a much smarter choice to return a small subset of the requested collection along with a bit of information on how to get the rest, rather than to return the entire collection simultaneously. For added convenience, the clients should also be capable of specifying how many resources they are willing to get within a single request (with some reasonable limits set by the server, so for example the client cannot request one million lectures per page). In the application, specifying pagination is also done through HTTP query parameters (more specifically through `size`, which specifies how large a single page of resources is, and `page`, which specifies which page to retrieve).

Given those complications, the final algorithm for processing a request for a collection looks the following way:

1. Map client's request to internally recognizable data formats, including all query parameters for pagination and filtering.
2. Map filtering parameters to a Specification, which is an internal representation of an idea of "this entity should have the following qualities" (see section 4.3.1 for more).
3. Retrieve the domain entities which satisfy filtering criteria from the database as a single page and map it to an internal representation.
4. Map each of the retrieved resources to its DTO representation and collect them into a generic *collection DTO* (more on this below).
5. Serialize the collection DTO, construct HTTP response, and return it to the client.

The *collection DTO* which was mentioned in the penultimate step of the algorithm is a template for how a response containing a collection of resources should look. It contains a list of the requested resources along with metadata about the way those resources were filtered and paginated. It is generic in the sense that it's agnostic to what kinds of resources it contains: a collection DTO is exactly the same for a list of Lectures and a list of Guilds, the only difference is that the former contains Lecture DTOs, and the latter – Guild DTOs. Within the application, Spring's default `Page` object is used as a collection DTO.

Client requests data which is not a domain entity

This remaining category of requests is a collective grouping for client wanting anything which isn't inherently linked to a domain entity. There is no single way to process requests for this kind of data, so how the server processes them and how the responses look is entirely dependent on the specific situation. That is, of course, aside from all of these responses obeying some fundamental conventions like "response must always be valid JSON".

A very simple example of such an "ad hoc" response is getting server's current version:

```json
{
  "version": "v0.0.1"
}
```

Server's version is evidently not a resource as it's not a representation of our domain. Indeed, the "version of a server" is not even a concept which holds any meaning when we are thinking about lectures, guilds, and users. Nevertheless it is the kind of information which is necessary for the client, so it has to be represented in some way.

In this particular case, the server exposes a separate endpoint via `VersionController` class, which accesses the properties file generated from the project's git repository (more on this in section 4.4.2), retrieves the version from it, and maps it to the JSON response.

4.2.3 CRUD operations for domain entities

Due to the fact that the main purpose of the application is to allow users to not only view lectures, but also create new ones, it was necessary to implement a way to modify the internal domain entities via the exposed web API. Most of this work was done in the semestral project [6].

To summarize, the this is done by exposing several endpoints for each of the editable resources. For example, client can add a new Lecture by sending a `POST` request to `/lectures`. This request must contain a valid JSON body which is equivalent to server's `LectureDto` class (i.e. it can be read by the server). In a similar way, client can send a `PUT` request to `/lectures/<lecture-id>`, where `<lecture-id>` is a unique identifier of a particular lecture, to modify any of the publicly available fields of that lecture.

4.2.4 Error handling

Sometimes things go wrong when the server is processing a request. The clients make a variety of mistakes ranging from trying to access a secured endpoint without proper authentication to simply providing filtering parameters in the wrong format. Sometimes, the server can also encounter an unrecoverable error due to which a request from a client cannot be properly processed. In all of these cases, the server must still be able to return a reasonable response to the client.

In Java, internal errors in the workings of the server exist as Exceptions, which may be thrown at any point during the execution of the application to signify that something unexpected had happened. Because of this, it's indeed a simple choice to leverage exceptions and, if anything happens when processing client's request, to retrieve the underlying exception and map it to a response. Spring Boot is responsible for most of the heavy lifting in this process, as it already maps all of the unhandled exceptions to a default JSON error responses. Here is an example of how a default error response looks for the case when the client is trying to access a secured endpoint without proper authorization:

```
{
  "timestamp": "2020-04-27T15:09:51.762+0000",
  "status": 401,
  "error": "Unauthorized",
  "message": "Unauthorized",
  "path": "/api/users/15"
}
```

The problem with Spring Boot's default error responses is that, if failure occurred on the server's part, the exception's stack trace is also serialized into the error response. This is not a particularly large concern for our application, however it's generally not advisable as this reveals a lot of information about server's internal workings to the client. This naturally makes the server more vulnerable.

To mitigate this and out of the desire to only return the information which is useful to the client, a custom exception mapping was implemented. This was done in the `error` package, mainly in the `GuildManagerExceptionHandler` class. This class implements Spring's abstract Exception handler which can map exceptions to HTTP responses, and makes sure that certain exceptions are mapped to a custom error response (which is implemented in the `ErrorResponseDto` class).

As the result, the same Unauthorized error looks the following way in the current state of the application:

```
{
  "code": "UNAUTHORIZED",
  "description": "Current user is unauthorized to
   perform this operation"
}
```

4.3 Business logic

4.3.1 Complex entity filtering

As we have discussed in section 4.2.2 when describing the types of requests a server must be capable of handling, the client requires a way to not only fetch collections of data, but also to filter them by a set independent criteria. This covers a basic use-case of user trying to find a lecture by a keyword.

The naive solution is to fetch an unfiltered collection of items into memory, and then filter it by each individual criterion provided by the client. Such an implementation has obvious drawbacks: the initial unfiltered collection can be quite large, and may not fit into the available memory. The filtering process can also become slow because of the necessity to iterate over the whole large collection (especially during the first filtering steps).

A much better alternative would be to not filter anything at all on the server, and instead to delegate this work to the database, which is build specifically to handle this kind of data filtering very efficiently. The only thing that has to be covered in this case is how to map client's filtering parameters to a database query. Within the application, this was achieved via Springs *Specifications*. A Specification is basically a predicate – it's a declarative description of desired qualities of an object. The following is an example of a specification for a Guild entity, which reads "A guild whose 'name' property contains the String called 'query'":

```
public static Specification<Guild> nameContains(String query) {
  return (guild, criteriaQuery, builder) -> (query == null)
    ? null
    : builder.like(guild.get("name"), '%' + query + '%');
}
```

Specifications can be chained together, which is a very important quality which allows us to handle complex filters based on several independent parameters. The following is a showcase of how Lectures are filtered internally – a combination of filtering by status, guild, lecturer, and full-text search within all of lecture's text fields:

```
var spec = where(hasStatus(status))
    .and(belongsToGuild(guildId))
    .and(byLecturer(userId))
    .and(where(titleContains(sanitizedSearchQuery))
        .or(guildTitleContains(sanitizedSearchQuery))
        .or(lecturerNameContains(sanitizedSearchQuery)));
```

4.3.2 Managing lectures

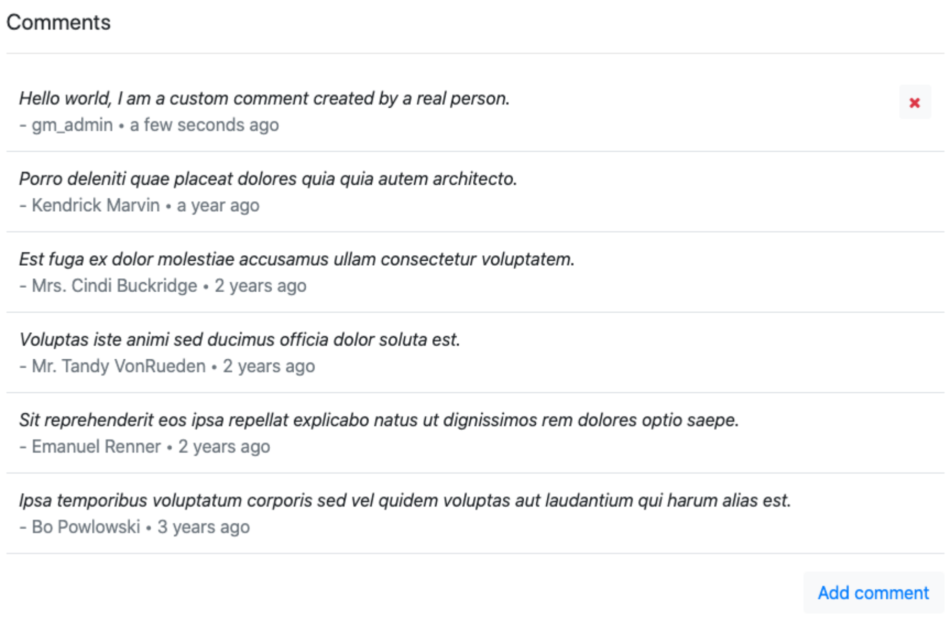
Lectures go through four stages during their lifetime:

1. **Suggestions** – these lectures model the individual user's suggestions about potential future lectures.
2. **Confirmed lectures** – lectures, which have been agreed upon by the guild, and have a defined future date, when it will be held and a lecturer, whose job is to prepare the lecture.
3. **Awaiting feedback** – lectures, which have been successfully finished and are currently awaiting feedback from other users. Currently it's being debated whether this stage is necessary (which will be talked about in more detail in subsection 4.3.3).
4. **Archive** – the final stage of a lecture, at which point it is preserved on the server with all of the necessary information and is accessible for future users.

Initially, it was thought that a separate voting system would have to be implemented to allow users to collectively vote on the future lectures of the guild. However, as the guilds developed within the company and more feedback was collected, it became evident that most of the voting happens during verbal conversations between the guild members. Because of this, the initial plan of creating a voting system has been abandoned in favor of a simpler suggestion system: any user can create a suggestion for a lecture, which can then be confirmed by the guild's moderator, after which point it becomes a confirmed lecture.

4.3.3 Giving feedback on lectures

The initial plan for the way users would give feedback on lectures was to prompt them with a predefined set of questions about the lecture after it was concluded. For example, one of the questions could have been "On a scale of 1 to 5, how useful was this lecture?" or "Do you think the lecture was too short, too long, or about the right length?". However, it was later decided that a simpler system would suffice – both because there were some doubts about how many users would actually take their time to fill out all of the questions, and because other, more pressing concerns were taking the implementation



Hello world, I am a custom comment created by a real person.
- gm_admin • a few seconds ago

Porro deleniti quae placeat dolores quia quia autem architecto.
- Kendrick Marvin • a year ago

Est fuga ex dolor molestiae accusamus ullam consectetur voluptatem.
- Mrs. Cindi Buckridge • 2 years ago

Voluptas iste animi sed ducimus officia dolor soluta est.
- Mr. Tandy VonRueden • 2 years ago

Sit reprehenderit eos ipsa repellat explicabo natus ut dignissimos rem dolores optio saepe.
- Emanuel Renner • 2 years ago

Ipsa temporibus voluptatum corporis sed vel quidem voluptas aut laudantium qui harum alias est.
- Bo Powlowski • 3 years ago

Figure 4.1: Example of how comments are displayed on the client. Note that the top comment has a button to remove it, because it has been created by the same user as the one that was logged into the application when the screenshot was taken.

time. So, instead of a complete questionnaire, a simpler system of comments was implemented.

A Comment is a written note a user can leave on a lecture. It contains three important fields: body text, timestamp, and author. Users can add comments in lecture details via a simple modal window. It was debated whether it would be better to make comments anonymous or no – but ultimately it was decided that keeping the names of the authors public would be better, as then lecturers could more easily get in contact with comments' authors, as well as it would promote those writing the comments to be more polite.

On the client, comments are displayed in lecture details as a chronological list – the most recent comments are added to the top of the list. You can see the example of how this looks on Fig. 4.1

Users can currently add comments to any lectures (and delete their own comments afterwards). This is the primary reason why it's currently being debated whether lectures should even have the state of "awaiting feedback" – since Comments are the implementation of the idea of "providing feedback", and currently it seems wise to not restrict their creating or modification to a single stage during a lecture's lifetime.

4.3.4 Attachments and file persistence

Archiving lectures implies also saving any additional information which was created for that particular lecture. Most frequently, this information takes form of a presentation or a set of lecturer's notes. The solution to persisting them was the idea of **Attachments**.

Attachments are external files, which can be linked to a lecture. On the server side, they are implemented as domain entities with a separate file system persistence logic. This was done so that the database didn't have to hold the file's contents, and so that, if anything goes wrong during the regular run of the application, the attachments could be accessed by accessing the file system of the server.

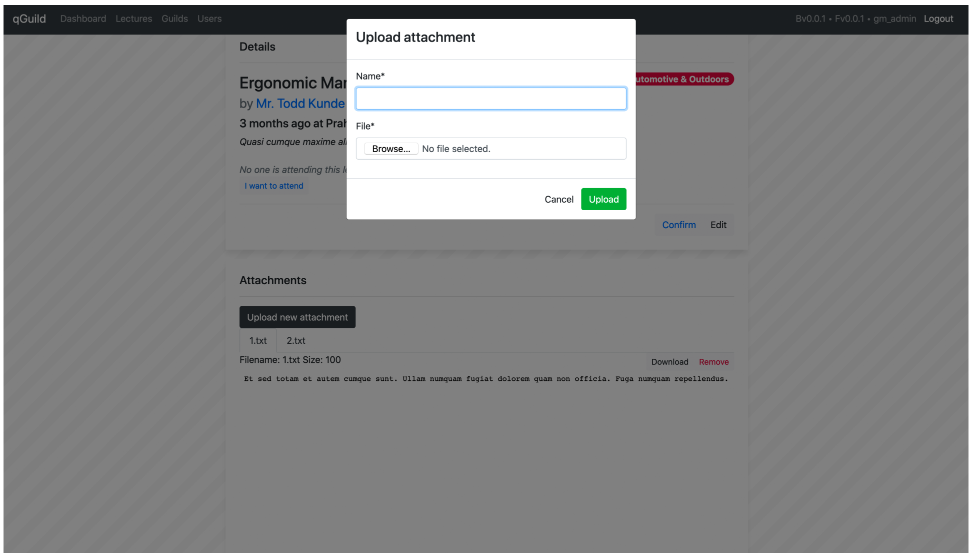
One of the problems which arose when implementing attachments was transferring attachment's contents between the server and the client. This is necessary to do because the users have to be able to upload their own attachments to their lectures, and they should also be able to download and preview the attachments which already exist on the server. Two potential solutions were examined: multipart messages and base64-encoded strings within JSON. The latter one was chosen, mainly because it was a better fit for the existing JSON-only API, as well as it was more convenient to serialize and deserialize the attachment's contents from base64 both on the client and on the server. The trade-off of this method is that base64-encoded content takes roughly 133% more space than the original binary data[7]. But, given the fact that the attachments that the users are expected to make are not large, this increase in transferred size was considered justifiable.

On Fig. 4.2 you can the example of how the GUI for uploading attachments looks on the client.

4.4 Integration

4.4.1 LDAP – user authentication

One of the problems which had to be solved is registering all employees as users which could use the application. Problem of registering employees in internal



Quasi cumque maxime al

No one is attending this l

No file selected.

Filename: 1.txt Size: 100

Et sed totam et autem cumque sunt. Ullam numquam fugiat dolorem quam non officia. Fuga numquam repellendus.

Figure 4.2: Modal window for uploading files to the server

services is not particularly unique. Because companies are almost guaranteed to have several distinct internal services which all require authentications, this problem has seen several mature solutions already. The overarching idea here is that it makes a lot of sense to have a single repository of credentials, so an employee can use a single set of credentials to access all applications within the company.

This information is usually stored in a *directory server*, which are frequently used by organizations as the central repository for user information. This is where *Lightweight Directory Access Protocol* (LDAP for short) comes into picture. LDAP is a protocol which standardizes the way an application many communicate with a directory service, and allows them to efficiently search for the necessary information.

The company which inspired this project currently maintains and uses its own LDAP server to manage the information about user's employees. Fortunately for us, Spring Boot has robust support for LDAP, which was used. Within the application, a separate configuration was written, which makes Spring connect to the provided LDAP server to check the credentials sent by the client:

```
auth.ldapAuthentication()
  .userSearchBase(securityParameters.userSearchBase)
  .userSearchFilter(securityParameters.userSearchFilter)
  .contextSource()
  .managerDn(securityParameters.managerDn)
  .managerPassword(securityParameters.managerPassword)
  .url(securityParameters.ldapUrl);
```

4.4.2 Git – application versioning via tags

Developing an application goes hand-in-hand with the necessity of versioning it. In the application's current state, server only exposes its version to the client, but in the future it's expected that it would also expose information about the latest change logs. This would be useful to give the users the ability to see what was changed in the last version of the application.

There are several ways to implement versioning of the application. The most naive way would be to explicitly write the current version into a constant variable within the code. This has several drawbacks, the most obvious being that there is now a bit of code which has to be updated every time a new version is released (and there is no way to know if it was updated or no aside from actually checking the code).

Git, however, provides an interesting alternative. It allows developers to create *tags* – permanent references to certain points in the history of the application. And the main way to use them is, indeed, to mark different versions of the application.

Git provides developers with two types of tags: lightweight tags and annotated tags. Lightweight tags only contain a name, which essentially makes them into a way to give a certain commit a name. Annotated tags store additional data alongside a tag name: the name and email of the tag's creator, date of creation, and a message. The latter is perfect for marking a new public release of the application.

Because our application exists within a git repository, it would be really good if we could automatically use git tags to keep track of versions. Ideally, we would do it in such a way that, when an application is built, this information is compiled right into the resulting `.jar` file.

To achieve this, a small external library called *Git Commit Id*, which was built specifically for Java applications running on Maven. It allows us to scrape the `.git` folder (which is where all git-related information is stored) and retrieve information about the last tag. This information is then written into a separate configuration file, which is then parsed by the application into an internal object and finally exposed by the `VersionController` as the `version` endpoint.

This low-level integration with Git greatly simplifies versioning of the server

– now every time a new version is released, the developer only has to add a tag (which they would do either way), and the next build of the application will automatically provide the clients with the up-to-date information about its version.

4.4.3 Email notifications

Because of prioritizing other features like application security and more user-friendly UI, email notifications were not implemented. A draft of the system has been implemented, which used Spring Email. However, as it was only a draft which never connected to a real email server, the base of the implementation has been removed from the project.

As of now, email notifications are not a high priority in the future development of the application, and therefore it may take some time before they will actually be implemented. Alternatively, as the application is tested more, it may become evident that they are not necessary at all.

4.4.4 Faking integration for development purposes

The database

The most important part of the application which had to be faked was the database – or, to be more precise, the data stored within the database. This becomes necessary during the development for obvious reasons: we cannot test the application if there is no data flowing through it. And simultaneously we do not want to keep the application filled with real "production" data either – it could contain sensitive information, which we may not want to keep on a local development machine. Therefore, a reasonable middle-ground solution must be used – to keep the application filled with sets of dummy test data.

The naive way to go about this would be to populate the database with dummy data by hand. This has obvious drawbacks in the sheer amount of work this would require the developer to do. This could be improved by writing, for example, a single SQL script, which would populate all tables with preset data. But this script would still have to be manually edited every time the database structure was altered, which would be terribly cumbersome.

A slightly more clever alternative is to generate the script automatically. And to make it even better, we would like to not only automatize feeding the database our fake data, but to make the process of generating the data itself automatic as well: after all, we care a lot more about the fact that there would be a hundred users saved in the database than which names they have. To top it off, we need a way to switch "database faking" on or off on our own accord: it would be beneficial to be able to sometimes turn it off to keep the same data set in the database across several application runs. This way we could, for example, test whether a database migration script works as expected, or add some data by hand which is only useful in the context of the feature we are currently developing.

Now we have arrived at a decent solution to the problem of faking database data. Two things are necessary for the actual implementation:

1. A way to execute bulk SQL statements from within the application.
2. A way to generate semi-believable data, so, for example, the names of generated users still look like real names.

It must also be noted that we do not want to make our fake data as precise as possible, i.e. it doesn't matter if the users' names are German, English, or Czech – we only care that users have a name at all. The requirements for the application are likely to change over time, the database's schema will also change over time – so we do not want to spend unnecessary time and effort now on something which would be discarded later. What we want instead is the cheapest way to generate "good-enough" fake data.

The first requirement – a way to execute SQL statements against the database – does not actually require the use of external tools and could be implemented using plain Java. Nevertheless, we will make use of a library called *Flyway*. The primary reason for using this library in our project is not executing SQL from within Java, but rather versioning database change, i.e. retaining the history of how the database was changing over time. Each semantically separate change in the schema is held within a separate SQL script (which is called a *migration*, because it "migrates" the database from state 1 before applying the script to state 2 after applying it), which are all saved in the same git repository as the server. The major benefit of doing this is that the way the database looks is now tightly tied to the server's version

– meaning that if, for example, we are working on two features which both change database's schema, we won't have to change the database manually when switching between those features. And what Flyway does in this whole idea is it automates taking those individual migrations and runs them against our existing database instance in the correct order – so our database is kept up-to-date with whatever new features are implemented.

By default, the library works with migrations written in SQL, but there is also support for writing them in Java – for cases when plain SQL is not convenient enough. And, by chance, the problem of generating fake database data is precisely that kind of problem.

This leads us to the second requirement for the complete database faking procedure – generating semi-believable data. For this, a second library called *Java Faker* was used, which is a lightweight dummy data generator. The library covers more than enough different kinds of data – from fake email accounts to fake locale-based address and bank account numbers, to quotes from popular movies – so it is more than enough to cover the necessities of our application.

With both requirements being solved, the actual implementation resigns in the class called `V30000101000000__testData`. The large number at the beginning signifies database migration version – the bigger the number, the later in order it will be executed by Flyway. These versions are generated simply by taking the timestamp of migration script creation – so a database change which was proposed on May 12 2020 at 13:45:05 would have version `V20200512134505`. The contents of the class are fairly primitive: for each table in the current version of the database, a single batch `INSERT INTO` statement is prepared, populated with a predefined amount of fake tuples of data, and then executed.

And to make this system switchable, the database faking class is only run if the application was started with the `local` Spring profile. Spring profiles can be thought of as on/off switches which developers define themselves. They are passed to the application as command line parameters during startup, and therefore cannot be changed while the application is running. The application can then detect if a particular profile is active and make decisions based on that. This is frequently used to change the way the application behaves on, for example, a local "development" machine, on the test server, and on the real production server.

LDAP

Our application performs an LDAP synchronization on startup, which keeps the employee information in the application's internal database up to date. However, to connect to a potential company's LDAP, the computer on which the application is being developed would have to be connected to the internal network of the company either directly or via a VPN, which complicates the boot of the application.

Our application connects to LDAP twice for two isolated reasons: The first connection happens when a user is trying to authenticate themselves (which was first described in section 4.4.1). For simplicity's sake we will refer to this process as *LDAP authentication*. The second reason is the aforementioned synchronization of the internal user database, which will be referred to as *LDAP user synchronization*. The internal database stores only the surface-level information about employees, such as their name, public email address, and internal username, without storing any sensitive information such as their private email addresses or bank accounts. Given all of this, to fake LDAP integration we would have to replace both of those processes with fake instances.

LDAP authentication is configured using a separate Spring Boot configuration class, which was shown in section 4.4.1. To replace it with a dummy implementation, writing a separate set of configuration was enough:

```
auth.inMemoryAuthentication()
    .withUser("gm_admin").password("{noop}admin").roles("ADMIN").
      and()
    .withUser("gm_user").password("{noop}user").roles("USER");
```

What this does is, register two dummy users which can log into the application: *gm_admin* with the ADMIN set of privileges, and *gm_user* with the USER set of privileges. No other users exist and none can be registered in addition to these (without changing the actual configuration). And just as with the database faking, switching between the real and the dummy configurations is implemented via Spring profiles. More specifically, if the `noLdap` profile is active, then the dummy configuration will be used instead of the real one.

For replacing LDAP user synchronization, instead of faking the connection, we entirely disable the synchronization if `noLdap` profile is active. But where will the user information be gotten from instead? Instead of fetching user information from somewhere, we generate users completely internally with a simple SQL batch, which is a part of the database faking which was already discussed.

Chapter 5

Client implementation

During the very early stages of designing the application, the decision was made to separate the underlying business logic (the server) from data persistence (the database) and the user interface and presentation logic (the client). This chapter contains an overview of the implementation of the latter: a short summary what was prepared during the semestral project, what problems the initial implementation had, how they were improved upon, and the thought process behind the current UI design.

5.1 Component-based architecture

The Java-based server was divided into several isolated layers, which laid the foundation to making it maintainable. The approach on the client was similar but not exactly the same. The main task of the client is presenting data to the user and allowing them to modify in a reasonable and comprehensible manner. The differences in the conventions of how to structure an Angular application versus a Java application also played a part in the architecture of the client.

The bulk of the client is divided into isolated *components*, which can be considered large building blocks of the user interface. A single component is composed of an HTML template, an Angular code-behind, which is the place for the component's internal logic and data. Components can nest other components within their HTML templates, which allows the developers to create generic "blocks" for their interfaces, which can then be reused in multiple places. This approach has the benefit of reducing duplication within the code, which is beneficial because it makes making future changes cheaper and less likely to introduce new bugs. Most of the client is made up of such atomic components.

Just as the server has some logic which does not fit into the domain entities, there is a thin layer of client logic which does not fit into a component. These are the classes responsible for directly communicating with the server and mapping its responses to internal data structure. To keep the code tidy, they are kept in the separate `_services` directory.

5.2 Consuming server's API

The solution of communicating with the server was one of the results of the semestral project [6], so this will only be a short overview of how it works.

The client implements a separate service to communicate with the server, called `RestService`. This class provides simple public methods (for example, `getLectures(params)` or `getGuilds(params)`) for calling the server for various resources. The following is an example of one such method, which requests a list of lectures from the server:

```
getLectures(params: HttpParams): Observable<any> {
  return this.http
    .get(baseUrl + '/lectures', {params})
    .pipe(map(this.extractData));
}
```

The specifics of the TypeScript syntax are not important, what this method does is, basically, create a `GET` request to the /lectures endpoint with the provided HTTP query parameters, send it to the server, receive the response and map it to an internal data structure. Because this logic is hidden within this single method, other classes which actually need to pull collections of lectures from the server can simply call this method and not worry about *how* the lectures are actually retrieved. This, by the way, also means that if we wanted to fake getting lectures from the server, we would need to change this single method.

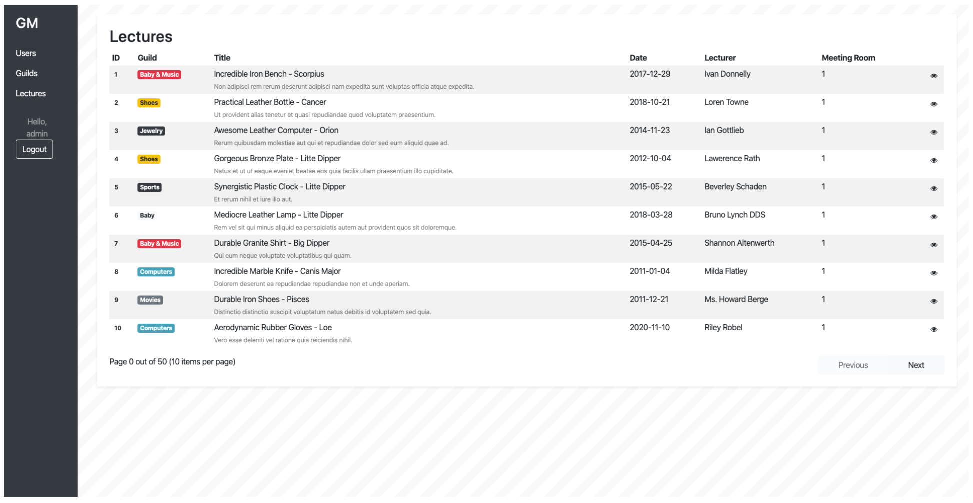
Just as with the example of getting lectures, similar methods are implemented for all other endpoints which the server exposes.

5.3 User interface

5.3.1 Design philosophy – Ease of use

During the development of the foundation of the application, the design of the user interface was not a priority[6], and therefore it was made only to be "good enough" to be usable.

The initial UI mostly consisted of tables and navigation buttons, which were stretched across the entirety of the user's screen (on some pages in two columns). We will illustrate some of the problems which the initial "proof of concept" client had on the page with the list of lectures, which the user will see quite frequently. By doing so, we will also showcase how these problems were solved in the current version of the application, and which problems still remain to be solved at some point in the future. You can see an example of the old lecture list UI on Fig. 5.1 and the updated version on Fig. 5.2.

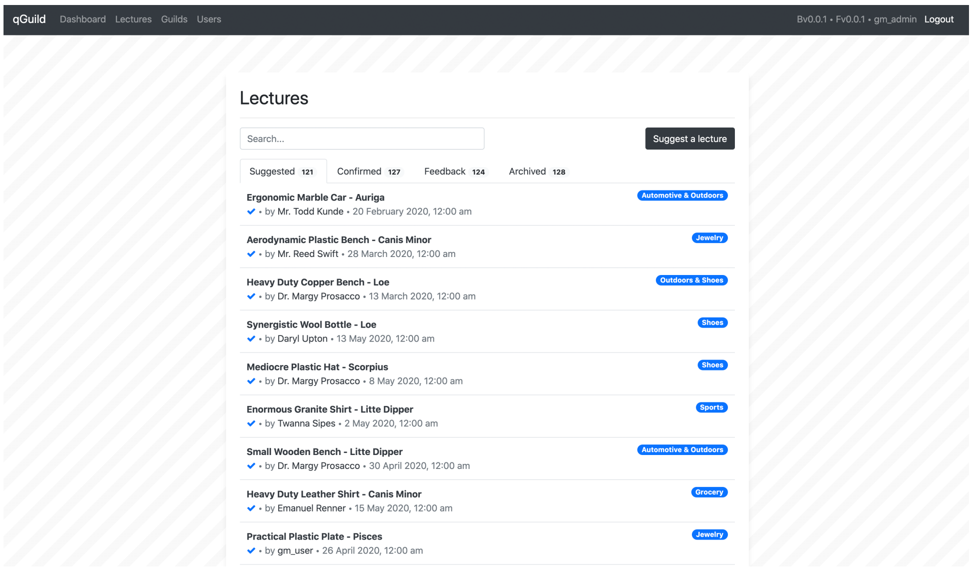


Lectures

ID	Guild	Title	Date	Lecturer	Meeting Room	
1	Baby & Music	Incredible Iron Bench - Scorpius Non adipisci rem rerum deserunt adipisci nam expedita sunt voluptas officia atque expedita.	2017-12-29	Ivan Donnelly	1	👁
2	Shoes	Practical Leather Bottle - Cancer Ut provident alias tenetur et quasi repudiandae quod voluptatem praesentium.	2018-10-21	Loren Towne	1	👁
3	Jewelry	Awesome Leather Computer - Orion Rerum quibusdam molestiae aut qui et repudiandae dolor sed eum aliquid quae ad.	2014-11-23	Ian Gottlieb	1	👁
4	Shoes	Gorgeous Bronze Plate - Litte Dipper Natus et ut ut eaque eveniet beatae eos quia facilis ullam praesentium illo cupiditate.	2012-10-04	Lawerence Rath	1	👁
5	Sports	Synergistic Plastic Clock - Litte Dipper Et rerum nihil et iure illo aut.	2015-05-22	Beverley Schaden	1	👁
6	Baby	Mediocre Leather Lamp - Litte Dipper Rem vel sit qui minus aliquid ea perspiciatis autem aut provident quos sit doloremque.	2018-03-28	Bruno Lynch DDS	1	👁
7	Baby & Music	Durable Granite Shirt - Big Dipper Qui eum neque voluptate voluptatibus qui quam.	2015-04-25	Shannon Altenwerth	1	👁
8	Computers	Incredible Marble Knife - Canis Major Dolorem deserunt ea repudiandae repudiandae non et unde aperiam.	2011-01-04	Milda Flatley	1	👁
9	Movies	Durable Iron Shoes - Pisces Distinctio distinctio suscipit voluptatum natus debitis id voluptatem sed quia.	2011-12-21	Ms. Howard Berge	1	👁
10	Computers	Aerodynamic Rubber Gloves - Loe Vero esse deleniti vel ratione quia reiciendis nihil.	2020-11-10	Riley Robel	1	👁

Page 0 out of 50 (10 items per page)

Figure 5.1: The old version of the lecture list UI



Lectures

Suggested 121 | Confirmed 127 | Feedback 124 | Archived 128

Ergonomic Marble Car - Auriga Automotive & Outdoors
✓ • by Mr. Todd Kunde • 20 February 2020, 12:00 am

Aerodynamic Plastic Bench - Canis Minor Jewelry
✓ • by Mr. Reed Swift • 28 March 2020, 12:00 am

Heavy Duty Copper Bench - Loe Outdoors & Shoes
✓ • by Dr. Margy Prosacco • 13 March 2020, 12:00 am

Synergistic Wool Bottle - Loe Shoes
✓ • by Daryl Upton • 13 May 2020, 12:00 am

Mediocre Plastic Hat - Scorpius Shoes
✓ • by Dr. Margy Prosacco • 8 May 2020, 12:00 am

Enormous Granite Shirt - Litte Dipper Sports
✓ • by Twanna Sipes • 2 May 2020, 12:00 am

Small Wooden Bench - Litte Dipper Automotive & Outdoors
✓ • by Dr. Margy Prosacco • 30 April 2020, 12:00 am

Heavy Duty Leather Shirt - Canis Minor Grocery
✓ • by Emanuel Renner • 15 May 2020, 12:00 am

Practical Plastic Plate - Pisces Jewelry
✓ • by gm_user • 26 April 2020, 12:00 am

Figure 5.2: The new version of the lecture list UI

The first problem of the old UI was its excessive width. The lecture table stretched across the entirety of the display, making the user have to move their eyes across the display to parse information about a single lecture. The solution to this problem was to limit the maximum width of the table and put it in the middle of the screen.

Another improvement in usability was acquired by restructuring the way the information about a single lecture was displayed. As can be seen on the old screenshot, lecture's title, date, lecturer, and guild were separated into individual columns of the table. This visual fragmentation is excessive, as users will rarely scan lectures by a single individual column and will instead want to see the entirety of the information at the same time. To fix this, columns were scrapped entirely, and only the most important information was left visible: the lecture's title, the lecturer's name, and the date when it was done. The title, being the most important bit of information, was emboldened, while the font for the secondary information was made lighter to accent the fact that it's not as important. The margins between individual rows of the table were also increased, and the striped background removed, which made the table look cleaner.

The next issue was navigating to a lecture's details page. In the old version, to see detailed information about a lecture, users had to click the small eye icon to the very right of the row of the lecture. This is not intuitive for two reasons. First, lecture's most important information is grouped to the left of the screen (the guild and the title), but "View details" page is as far away from that as it can be – on the right end of the screen. Second, which was figured out during the testing, is that users frequently try to click on the title of the lecture to see the details. Both of these problems were fixed in the new version: the "View details" button was removed altogether, and the title, which is already the most prominent bit of information for each lecture, was made to act as the only link to the lecture's details page.

One more large improvement was the ability to filter lectures. Filtering is composed of two interactive elements: first, lectures are visually separated into four tabs according to their status; second, a large search bar was added at the very top of the table. Initially, creating separate search bars for lecture's title, guild name, and lecturer's name was considered. But with the simplification of the table, it soon made no sense to have such granular searching capabilities. Instead, the single present search bar will search across all text fields that the lecture has. This is quite fast due to efficient filtering implementation on the server, and allows the interface to remain as simple as possible.

The improvements mentioned above, along with several other ones, led to the following core ideas, which were adhered to (to the best of our ability) in the design of the rest of the user interface:

- **The interface is constrained to the middle of the screen** – this way, the user does not have to move their eyes across the whole screen.
- **All lecture, user, or guild references are links** – this allows users to more easily move between pages, which provides a more pleasant experience.
- **Colors are kept to a minimum** – most of the application is kept in grayscale, which is easy on the eyes and keeps the application looking clean. Limiting the use of color also makes the color that is present in the interface more visible.

5.4 Bootstrap

The user interface required proper styling, however creating a custom CSS solution from scratch was not within the scope of the project – this would have taken a lot of additional time which could not be justified. Therefore, the decision to use an external ready-to-use CSS library was made. And out of all of the libraries which are currently available, *Bootstrap* is one of the most well-known and robust, which is the reason for choosing it. The whole point of the library is to allow developers to quickly come up with a relatively good-looking interfaces without having to write their own CSS (or, at least, without having to write a large amount of it).

One of the problems which regular bootstrap does not solve is dynamic interface elements, such as tab groups or date pickers. Because of this, an additional smaller *Angular Bootstrap* library was also added, which, as is probably evident from the name, works on integrating the core Bootstrap library with Angular. This is exactly what provided us with templates for some of the more complex UI components, such as the aforementioned date pickers and tabs.

Chapter 6

Security

Having discussed the problems that were solved during the implementation of both the client and the server, the last large topic to talk about is the security of the application. But before we do that, What does "security" even mean in the context of a web application? To put it simply, it means that:

- Only the "good" users can access the application.
- Users which can have a limited set of actions they can do based on their privileges.
- Finally, these actions are validated before they are executed.

This is of course not an exhaustive list of what "security" really means, but it is the list of the problems which were focused during the project. The first one is a problem of user authentication, the second – user authorization, and the last – data validation. All of them will now be discussed in that same order.

6.1 Token-based user authentication

How do you make sure that the incoming requests come from a trusted source? This is the core of the problem of user authentication. The simple solution is to require all requests to have some sort of an "identity confirmation" attached to them, which the server could validate and reject to serve the request if the provided identity could not be verified. That sounds reasonable enough, and indeed this is the way authentication usually works: clients attach a special header or a cookie to their requests, which identifies them in a way that the server understands.

The next question is how do new clients obtain the "identity confirmation"? There are several true-and-tested approaches to this, ranging from very simple ones such as basic HTTP authentication to robust secure solutions such as OAuth2.

During the first phase of development of the application, there was no necessity to implement proper authentication[6], and therefore the first implementation used the simplest possible solution – basic HTTP authentication. The way it works is that the "identity confirmation" that the client must send with each request is a username-password pair, which is base64 encoded. This is far from ideal in terms of security – or, rather, this offers no security at all, as base64 encoding is exactly that – encoding, not encryption. If a request with basic HTTP authentication header is intercepted by an attacker, retrieving client's credentials is trivial – and, therefore, the concerns of security fall entirely on having a secure communication channel between the client and the server.

6.1.1 Stateful authentication

Basic HTTP authentication is, evidently, quite a sub-optimal solution. A much better approach is *token-based authentication*. The core idea here is that the client only has to send its credentials to a specific entry point of the server at the beginning of the communication session. The server then verifies their validity and, if they are valid, generates a unique key (commonly referred to as a *token*), which it sends back to the client. Each subsequent request that the client sends to the server must now contain the token. This token is then verified by the server to check the identity of the request's sender.

The token could just be a sufficiently large random string of bytes, which would be stored in the server's memory (or, in some cases, in the underlying database as well) with some additional data about which user this token refers to. This is what is called *stateful*, or *session-based*, authentication – because for each currently logged-in user, server has to save a little bit of information – their *state* (to put it another way, information about an ongoing communication *session* is stored). One of the major benefits of session-based authentication is that, if there's any reason to stop trusting any particular client, their session can be terminated immediately, effectively denying them any access as if they were never authenticated.

Persisting information about users on the server has a drawback of memory consumption scaling linearly with the amount of logged-in users. This is not a problem for a small-scale application, such as ours, but due to personal interest, an alternative solution was still sought.

Stateless authentication

The alternative solution arises from the idea of storing some user-related data within the authentication token itself. If the server can safely store, for example, client's username and a list of privileges within the token, then there is no need to save *any* information on the server. This kind of approach is called *stateless authentication*.

Stateless authentication is the system that was chosen for this project, with the main reasoning being, as already stated, the desire to experiment with various authentication processes.

To securely store data within a token, some sort of encryption is necessary. One of the currently widely used standards for these kinds of authentications are *JSON Web Tokens* (JWT for short) [8]. A JWT is composed of three parts: the *header*, which contains information about which algorithm was used for encryption of the data, the *payload*, which is where server stores client-related data, and the *signature*, which is used to verify that the JWT has not been modified or tampered with.

In the application, JWTs are used as authentication tokens, and the payload stores client's username and database ID. After receiving a request with a token, the server decrypts JWT, checks that it is valid, extracts client's ID and fetches their information from the database, and then confirms that the client is authorized to do what they are trying to do. The process of generating, encrypting and decrypting JWTs is beyond the scope of this project, so a robust external library was used to manage JWTs.

6.2 User authorization

After solving the problem of securely authenticating clients, the next problem is to define which users can perform which operations. A natural way of achieving that is grouping users into several categories and then assigning access rights to these groups.

An alternative approach which is sometimes used in larger applications is the inverse – instead of grouping users into separate categories, the actions users can take are grouped into categories and then users are assigned a list of categories of actions they are authorized to do. This was considered as a potential solution, but ultimately disregarded as the small amount and relative simplicity of access rights that had to be defined did not justify the additional complications of solving authentication in this way.

Therefore, the users had to be divided into groups based on what they could do with the application. The first separation (and the main one which would be set in this way) is between `ADMIN`s and `USER`s. The ADMIN category represents a small subset of people who will be responsible for managing the application's internal workings. Users from the ADMIN category have access to all endpoints and are allowed to perform any operations on the application – frequently those people would simultaneously have access to the application's underlying database and the source code, so they would be capable of fixing any problems which might arise during regular use of the application. USERs, on the other hand, are the majority of expected users of the application. These are regular employees, who can see only a subset of publicly available lectures and guilds, suggest lectures, and manage only their own lectures.

These two categories were implemented using Spring Security's role management capabilities [4]. This allowed for a simple, coarse-grained definition of which endpoints could be accessed only by admins, and which – by all users.

The USER group, however, had to be further divided due to the existence of an idea of a *guild moderator*. As mentioned in the beginning of this document, each guild has a single representative person who is responsible for managing their guild's internal workings – organizing meetings for lectures and discussions, having the final word on choosing the next subject – in short, keeping the guild running. These users naturally will have to have more access rights than just USERs, but less than ADMINs.

Creating a third `MODERATOR` category was considered first. However, as the actual rights of moderators were being examined more closely, this proved to be not a good enough solution: a guild manager must be capable of editing details of *their own guild*, not of any guilds in general. This constraint cannot be satisfied simply by saying that clients with the role of `MODERATOR`s should be able to edit guilds.

Instead, what was implemented was a custom finer-grained solution. Upon confirming that a request to modify a guild is coming from a valid client, the accessing user is further checked to be either an ADMIN or the guild's moderator. If the user is neither of those things, the request is denied with a `403 FORBIDDEN` response.

To summarize, the application divides all users into two coarse-grained categories of ADMINs and USERs, and then further divides USERs into finer-grained guild moderators and regular users.

6.3 Data access based on user permissions

There is another facet to permission-based security than just the actions the users can take. The kind of data they are able to retrieve in the form of collections of resources or single resources is also determined by their permissions.

This data access checking is done on the service layer of the server. A good example would be the guilds, which as we have previously defined, could either be be public or confidential. Confidential guilds should only be visible to admins and users which are already a part of the guild. To do this, the service class responsible for guilds does a quick check for exactly that: whether the request is coming from an administrator or a member of the guild. If it is, then the guild data is fetched, prepared, and returned to the client in the usual fashion. Otherwise, the processing of the request is terminated and an error response with `403 FORBIDDEN` status code is returned to the client.

6.4 Ensuring data validity

One last thing to keep in mind when talking about security is data validity. The concept is simple enough – at any point in time we must be certain that the data we are working with makes sense both from a universal point of view (for example, a person must have a name), as well as within the specific constraints of the business (for example, guilds can only have a single manager).

Data which the application works with is checked for validity several times in the different layers of the application:

1. First, the data which was input by the user is checked against a basic set of rules on the client: a request is not permitted to be made if a required field is missing. Using data-specific input fields also helps make sure that, for example, date fields are properly formatted.
2. Upon receiving a request from the client, if it contains a body, that body is mapped to an internal DTO and also checked for validity by the server. This check is made automatically by Spring by processing the Bean Validation annotations, which are used on DTO classes.
3. If everything looks valid so far, the next validation happens before persisting entities to the database. This is, again, done using Bean Validation annotations, only this time they are put on the internal domain classes, and are therefore more detailed than the ones on DTOs.
4. Finally, there is an entirely separate isolated set of constraints which are defined within the database itself, which check whether the data that is about to be written into a table makes sense in terms of entity relationships and all non-null fields.

6.5 Security in user interface

The most important part of the security is maintained by the server – if a user is not authorized, they will not be able to request the data or perform the actions they should not be able to. So the client's role in the security is more cosmetic than anything else: it only has to make sure that the user does not see the controls (such as buttons or menu items) they aren't authorized to use, and that they cannot get to the pages they aren't authorized to use.

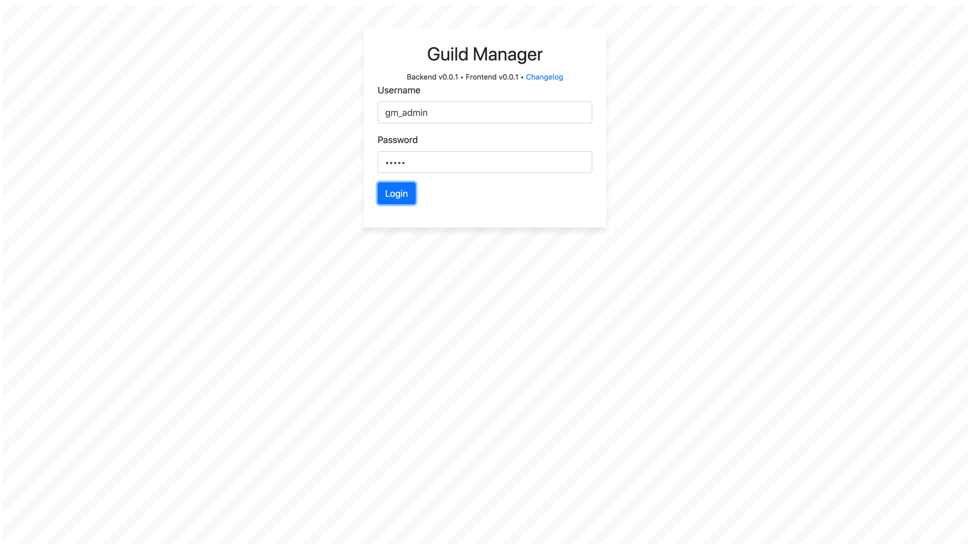


Figure 6.1: The current state of the login page

The basis of the client-level security was implemented along with the foundation of the client in the [6]. To give you a quick overview: the client initially forbids access to any of the application's pages to an unauthenticated user, redirecting them to `login` instead, the example of which you can see on Fig. 6.1 (which is done by intercepting any of the server's responses and catching any 401 UNAUTHORIZED status codes). There, the user can authenticate themselves by providing their credentials. If the credentials are valid and the server confirms user's authentication, the current user's token is persisted to browser's local storage and then used to authenticate all of the further actions the user takes. This functionality resides in the `BasicHttpAuthInterceptor`, whose purpose is to intercept any outgoing requests and enrich them with the authentication header. Saving the token to local storage also means that the user does not have to log in every time they open a new tab or a new browser window. After successfully logging in, they can access most of the client's other pages (the access to individual ones depends on the rights of the client).

Other than the login page, the only other security concern solved on the client is the aforementioned hiding of controls which the user is not permitted to see.

Chapter 7

Testing the application

During development, the application exists in an unstable state where features are created and modified very rapidly. This naturally means that bugs are also frequent and quite easy to introduce. This is a problem which can partly be solved by good planning before the implementation of each new change, however even that is liable to the human factor. Because of this, testing the application continuously during development is imperative.

Testing in general is vast and muddy, so for the sake of this project we will define two kinds of testing: *automated testing*, which is done by the computer, and *manual testing*, which is done by real people. Ideally, we would want the former type of testing to cover all of our needs – that way, we as developers wouldn't have to test anything by hand at all. Sadly, that is impossible, as, for example, automatically testing user interface is a large and currently unsolved problem. Nevertheless, we strive to make automated tests cover as much of our application as possible – if we have a robust set of automated tests, we can be reasonably certain that a new feature that we have just added didn't accidentally break half of the application.

Because of this, a suite of automated tests was created on the server, which check that the basic functionality of the application works.

7.1 Automated testing

Out of the two large separate parts of the application – the client and the server – the server is much easier to test automatically. Testing the server application is also devoid of the problem of testing graphical user interface, which, as already mentioned, is very troublesome to automate. Because of this, most of the automated tests were written for the server, while more manual testing was done on the client.

Within the context of small-scale enterprise application, the automated tests can be divided into two categories: *unit tests* and *integration tests*.

Unit tests are the simplest and most low-level of the two. Their goal is to taste that a single particular "bit" of the application (called a *unit*) works as expected. A single method or a group of methods which achieves a certain goal are examples of what can be considered a unit. The great thing about unit tests is that they are very cheap in terms of execution time – because they test a very focused part of the application, the application itself doesn't need to be running during the tests, and indeed very frequently it's enough to just instantiate the class under test. Because of this, it makes sense to have lost of unit tests and cover as much non-trivial application logic with them as possible.

Integrated tests are more high-level than unit tests. The goal of an integration test is to verify that several parts of the application are working together correctly and producing expected results. Within our application, integration tests cover the entire workflow of the server getting a particular request, processing it, and sending a response. However, these tests do not test that every single part of the process is correct – if they did, they would be enormous, very complex, and difficult to maintain. Instead, they verify that, if a request is made under certain conditions, the server responds in an expected way (for example, if a POST is made to create a guild, and the authenticated user is not an administrator, then the request is rejected with a 403 FORBIDDEN response status).

At the current stage, 54 automated integration tests were written using the popular Java testing framework called *JUnit*: 16 check the correctness of Lecture-related endpoints, 20 – Guild-related endpoints, 8 – Attachment-related endpoints, 6 – user-related endpoints, 2 – meeting room-related endpoints. 1 test was written to check that the server version endpoint works as expected, and 1 test was written which checks that the application even loads. This is 21 more than the after the first stage of the project, where there were only 33 tests [6]. Initially, we expected that with the implementation of the bulk of the business logic which happened during this project, unit tests would naturally arise. However, it turned out that most of the business logic was not complex and a lot of it was concerned with user permissions, which are the domain of integration tests, so no unit tests were written. Given all of that, it must be admitted that the currently existing suite of automated tests does not cover all of the important logic, and it could do with some improvement and enrichment. This section of the project will be improved on in the future, but it will be done outside the scope of this project.

7.2 Manual testing

Manual testing, in contrast to automated testing, was done primarily on the client, because this was the most straightforward way of testing the application. Nevertheless, some manual testing was also done on the server during the development of individual features.

Server-related manual testing took form of sending pre-made requests to a running instance of the server and observing whether the responses were correct. The bulk of such requests was done using an external application called *Postman*, which provides the developers with the ability to create ordered collections of HTTP requests.

Just as after the first phase of the project, not automated speed tests were written due to them not being of great importance to the working of the application. Nevertheless, because the amount of logic firing for individual endpoints has certainly increased since the first phase of the project, some statistics were collected to see if any endpoints began returning responses more slowly. For each comparison, both the old version and the new version was tested by sequentially sending 100 requests (each requesting only 10 of their respective resource i.e. 10 lectures or 10 guilds) to a running instance of the server. The comparison also does not include the resources which were not created during the semestral project:

	1st phase	Current state
Lectures	97ms	319ms
Guilds	120ms	140ms
Users	36ms	79ms

As can be seen, there has been an increase in the response times all across the board. This is to be expected – the size of the returned entities has increased as more of their functionality was implemented. The original estimation was that all response times would be kept within 100-200ms, and only getting all lectures has crossed that threshold. It is slightly noticeable in the user interface, however it hasn't had as bad an impact on the usability as originally expected, and therefore was not yet dealt with. A potential solution, which will be implemented at a later point in time, is to limit the fields which are returned by the endpoints only to those necessary to the client (or to allow the clients themselves to send a list of the fields they need).

The manual testing on the client involved going through the user interface and interacting with it in a way that a regular user might interact with it. This meant checking that, for example the login page works correctly, all of the links between pages redirect users to correct pages, and all components render properly. This kind of testing does not provide a lot of insight, but rather simply validates that the application is working as expected.

The other kind of manual testing involved giving other users the unfinished version of the application. This has been done sporadically, but it has nevertheless proven to be insightful, particularly in terms of the usability of the interface – for example, as discussed in section 5.3.1, it showcased the fact that users were much more inclined to click on the names of users, titles of guilds and titles of lectures to try to see more information about them. Because of this, the separate "view details" links were deprecated and titles were made into hyperlinks themselves.

Chapter 8

Conclusion

This short chapter, will give a high level overview of the project's goals, of what was achieved and what was not, and attempt to give reasonable judgment to the results.

The inspiration for this project was taken from the internal practices of the company called Quanti s.r.o., where employees are teaching each-other in the form of presentations within thematic groups called "guilds". The moderators of these guilds – the people responsible for managing individual lectures and keeping the guilds running – were primarily using ad hoc solutions to do their job. This made the process of moderating the guilds somewhat cumbersome and time-consuming. This project has set out to attempt to fix that.

First, the existing Learning Management Systems were analyzed to see whether none of them would be a right enough fit for the problem and whether it was not solved by somebody else beforehand. Four existing E-learning solutions were analyzed: Moodle, OpenOLAT, Canvas, and Sakai. The conclusion was that none of them fit the exact requirements this problem had.

After determining that, a custom solution was designed and built. The solution consisted of two separate applications: the client, written in Angular, and the server, implemented in Java and leaning heavily onto the family of Spring libraries. The bulk of the work which was done during the project was the implementation of these two applications. Their foundations were created as a part of the semstral project which preceded this project [6]. This project built on top of that foundation and:

- Implemented the bulk of the business logic on the server, such as the complex filtering of lectures and their management.
- Improved the security of the project by switching to a token-based authentication and enriching the access rights of various users
- Integrated the application with several other applications widely used within software development companies: LDAP, Git.
- Redesigned the bulky UI which was previously implemented and made it much more user-friendly

The project's main goal was the creation of the application which covered suggesting, managing, and archiving lectures along with their main relevant information: the presentation files, text notes, and external links. The application also had to be a repository of all of the existing lectures, so that users could have quick access to them. From this point of view, the project had met its main goal.

For testing purposes, the project has also been deployed to the test server within the company's internal network. Initially it was planned that it would also be deployed to the production server, however due to the unexpected situation with global pandemic, this was postponed to a date which falls after the conclusion of this project.

Initially, it was expected that a system for voting on lectures would be necessary. However, as more lectures were held within the company, it became evident that most of the decision-making for future lectures happened at the end of other lectures via verbal communication between the employees. Therefore, the initial plans for the voting system were discarded and a simpler system of Suggestions was implemented.

In conclusion, the project has achieved the main goals it had set out to achieve. The produced application, while not finished, is capable of archiving lectures, allowing access to them to different sets of users, and managing suggestions for future lectures. Despite the slight shortcomings, such as the delayed deployment, it is considered to be successful.

Chapter 9

Potential improvements

The application has reached a stable state. There are, however, several ways it could be developed further to encompass more of the user's needs and make it more convenient. One of the potential improvements is designing a way to handle large video files. One of the changes in the workings of guild lectures, which happened during this project, was that they began being recorded for archiving purposes. These recordings, after being processed, can weigh up to several gigabytes, which is far too large to handle with the currently implemented system of Attachments, and therefore the video recordings were not in the list of the initial features the application had to have.

Another potential improvement would be to integrate the notification system with Slack – a chat system which is widely used in software engineering companies. This may make for a less intrusive, lighter alternative for email notifications, although further testing would be required to decide whether that is the case.

A slightly less interesting, but certainly no less important improvement would be to write proper documentation for the server's web API. Or, as a more robust alternative, to have the documentation be automatically generated from the source code. Auto-generated documentation is a fairly well-established concept, and there are several tools which could be used to achieve that.

To summarize, there is definitely room for improvement of the application, however it falls outside the scope of this project.

Bibliography

[1] Nicholas S. Williams (2014) *Professional Java for Web Applications*, John Wiley and Sons

[2] Craig Walls (2011) *Spring in Action*, Manning, Edition 3

[3] Gierke, Darimont, Strobl, Paluch, Bryant (2019) *Spring Data JPA Documentation*, Online, available at: https://docs.spring.io/spring-data/jpa/docs/2.2.3.RELEASE/reference/html/#reference

[4] Alex et al. (2019) *Spring Security Documentation*, Online, available at: https://docs.spring.io/spring-security/site/docs/5.2.1.RELEASE/reference/htmlsingle

[5] Fielding, Roy Thomas (2000) *Architectural Styles and the Design of Network-based Software Architectures*, Doctoral dissertation, University of California, Irvine. Online, available at: https://www.ics.uci.edu/˜fielding/pubs/dissertation/top.htm

[6] Tsimafei Raro (2020) *Application for managing professional learning*, Semestral project, Czech technical university in Prague

[7] Simon Josefsson (2006) *The Base16, Base32, and Base64 Data Encodings*, Online, available at: https://tools.ietf.org/html/rfc4648

[8] Jones, et al. (2015) *JSON Web Token (JWT)*, Online, available at: https://tools.ietf.org/html/rfc7519