Software Engineering Methods: Lab Assignment 1

Group 18A

December 2021

1 Introduction

The system we are going to build allows lecturers to hire teaching assistants (TAs) for their courses. Lecturers can see a list of applications, and reject or accept TAs. Lecturers may leave reviews about their experience with TAs to help other lecturers decide whether to hire the TA. To build this system in an efficient and scalable way we will use microservices and various design patterns. Each microservice has access to their own databases and can have multiple instances to distribute the information load. In this document we will be discussing the design of our microservices and how we incorporated several design patterns.

2 Architecture Plan

In this section, we will describe and motivate the architecture of our system. First, we give an overview of the microservices and their responsibilities. Secondly, we provide a detailed explanation of the internal architecture of each microservice as well as a motivation for the division of the microservices.

2.1 Bounded Contexts

The bounded contexts we have identified are: Applications, User (Authentication and Roles), Courses, Notification, and TA. We translated these contexts into microservices. However, for the overall architecture of our system, we also incorporated two extra services: Gateway and Service Discovery.

2.2 Microservice Overview

Application Service

This microservice handles all functionality related to the employment process of teaching assistants. An overview of the main responsibilities would be:

- Allow students to apply for a teaching position.
- Allow lecturers to select candidates.
- Provides recommendations of potential teaching assistants.
- Communicates with the TA service when an application is successful.

Authentication Service

This microservice handles all functionality regarding the authentication and authorisation process of the system. All user information and roles are stored by the authentication service. An overview of the main responsibilities would be:

- Stores user login information (stored in encrypted manner in database).
- Provides access token to client, which can be used to determine the role of the client.
- Communicates with other services to verify the role of the client.

Course Service

This microservice provides all information related to courses and functionality that can affect how the application process for a particular course changes. An overview of the main responsibilities would be:

- Provides the grade of the student to the Application Service, when requested by a lecturer.
- Determines whether a course is still open recruiting teaching assistants.
- Accumulates the average work hours required for a TA when employed for this course.

Discovery Service

Services need to communicate with one another. In a traditional distributed system, services run at fixed locations with predefined hosts and ports, communicating with HTTP requests. On the other hand, in a modern microservice-based implementation, the number of services and their location may change as the application is running. We use the Netflix Eureka Discovery service to facilitate for these dynamic and real time changes. By keeping a service registry, each microservice registers itself when it becomes available for requests and de-registers once it becomes unavailable. This allows microservices to have multiple instances that run in parallel and it makes them easily available to each other.

Gateway Service

This is a microservice that handles all the requests and performs the dynamic routing of microservice applications. An overview of the main responsibilities would be:

- Dynamic Routing
- Load balancing
- Resiliency
- Security

Notification Service

This microservice provides the client with information updates relevant to the client. It handles all information updates provided by other services. This service is **pull-based**, hence will only send an notification when a client asks for new notifications. An overview of the main responsibilities would be:

- Provides notification about status updates on the application of a client.
- Provides notifications about the approval/disapproval of contracts and workload hours.
- Informs, through notifications, the teaching assistant whether the indicated workload hours need adjustment.

TA Service

This microservice provides all functionality relevant to the teaching assistant. An overview of the main responsibilities would be:

- Provides each teaching assistant with a contract with information about the number of work hours required and the course information.
- Allows the teaching assistants to declare the number of hours they spent on the course.
- Allows lecturers to write reviews about their experience with a teaching assistant.

2.3 Microservice Architecture

In this subsection, we will motivate our design choices regarding the microservices and describe their internal architecture.

To determine which microservices should exist in our system, we focused on *three* principle properties that a microservice should have: **Cohesion**, **Low Coupling**, and **covers a Subdomain**. We will now explain how each of our microservices have these three properties and therefore deserve to be an independent service.

Application Service

Application Service maps to a central domain object: *Application*. Two important observations are: the application process aggregates many functionalities and is in fact rather independent of any other domain object. Hence, to allow for more cohesion, it is reasonable to combine all functionalities related to applying for a TA position within one microservice. Moreover, while we could incorporate more elements into this service, such as **Course** information, this would significantly increase the amount of coupling in the system.

Authentication Service

Authentication Service handles the authentication process of clients using the system. This is represents the mapping of the User domain to the system, and incorporates it with the authentication process. The authentication service stores User information and distributes tokens that can be used to represent the role of a User/client. By having a central service, we simplify the security of our system. If each microservice had its own authentication process, the task of identifying the *Role* would be unnecessarily complex.

Course Service

Course Service handles all information to courses, which are required by both the TA service and the Application Service. It needs to be an independent service because it centralises information updates about courses. If the course information was distributed between the multiple services, we would require multiple queries for simple information retrieval or we would store duplicate data. Hence, creating this independent service significantly reduces the coupling of our system.

Notification Service

Notification Service needs to be an independent service. While this may be a small component within the system, many microservice need to *send* and allow clients to *receive* notifications. Instead of having every microservice implement its own notification system, we have a dedicated microservice for notifications. This allows for cohesion as the client can use one central endpoint to receive all relevant notifications.

TA Service

We also make TA Service an independent service. The functionalities required by teaching assistants are grouped within this service. Furthermore, this service maps to a central domain object: *Teaching Assistant*, hence cohering with Domain Driven Design Principles. This service is an *aggregation* of all the functionalities relevant to teaching assistants.

Discovery Service

The discovery service needs to keep the service registry and thus needs to be independent.

Gateway Service

By keeping the Gateway service as an independent service, we can centralise the entry point into the system. The gateway centralises the process of authentication and routing the requests of clients and various microservices. This centralisation prevents all microservices from implementing its own entry point and authentication process, hence allowing our system to be more scalable in the long run.

2.4 Internal Architecture of Microservices

We have decided to employ the **Layered Architecture** for each microservice. Microservices have the following components/layers: **Controllers**, **Communicators**, **Repositories**, and **Entities**.

The **Controller Layer** handles the HTTP requests from the client and other microservices. It checks the validity of the request, according to the role of the client and the functionality he/she is trying to invoke. If the request is valid, then the Controller Layer will call the **Repository Layer** to perform the intended task. The Repository Layer interacts with the database.

In some cases, the Controller Layer may require additional information from other microservices, and will therefore invoke the **Communicator Layer**. The Communicator Layer sends HTTP requests to other microservices.

Note that the Communicator Layer will invoke the Controller Layer of *another* service. However, this differs from regular requests from the client using the application. In case the endpoint of a Controller Layer is only accessible to other microservices, we do not verify the role of the client.

2.5 Communication Between Services

Microservices within our system communicate with HTTP requests and responses. The communication between services in **synchronous**.

In order to reduce the communication overhead between services, we have define a **shared** module which contains all the entities used for communication. An example would be `SelectInfo` which is used by the **Application Service** to communicate a new teaching assistant to the **TA Service**. `SelectInfo` contains the minimum amount of information needed for the TA service to save a new TA into the database.

2.6 Security of Application

The security of our system is conferred through the use of a Json Web Token (JWT). The whole authentication pipeline is mainly focused on the interaction of the authentication service with the `User` of our system, and it works as follows:

- User: the user sends a request to get a token containing all of their credentials.
- Server (Authentication Service): the server checks and validates the credentials and sends back the token
- User: with every request, the user has to provide the token which will be validated by the server.

After this, the only question remaining is how the other services will use the authentication service to stay in accordance with the role of the user and what functionality they have access to. The answer is pretty straight-forward: the checks and validation will happen at the gateway level. The main idea is that the authentication service will validate user credentials and issue tokens before allowing any request to go further to any of the services. To sum all up, we are blocking the requests unless the user is authenticated.

2.7 Navigating the Project

Our project can be found on GitLab. To identify the microservices, they are stored as subprojects within the parent project called `template`. The microservices can be identified with name such as `*service`.

The components within one microservice are represented by `packages`, such as **Communicator**, **Controller**, **Repository**, and **Entities**. Other modules such as `Util` are utilities required by the components and are not main aspects of the Layered Architecture.

3 UML

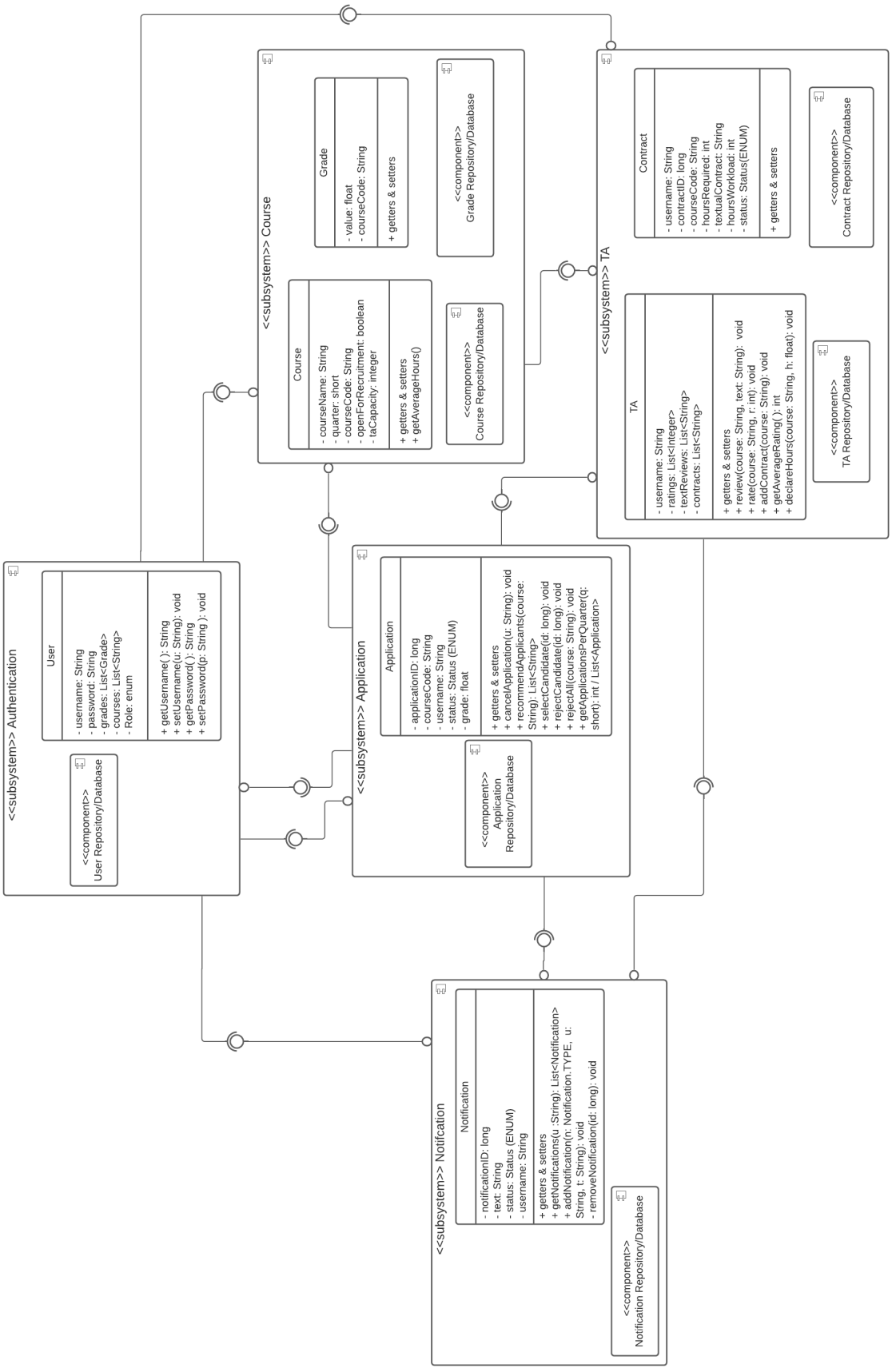

Figure 1: UML Diagram excluding Design Patterns

4 Design Pattern

4.1 Strategy Pattern - Candidate Recommendation

The Strategy Pattern will be applied in implementing the Candidate Recommendation feature.

The Candidate Recommendation feature allows the lecturer to select different criteria for recommending potential TAs. By employing this feature, the lecturer will receive a list of application sorted upon the selected criterion.

This pattern is implemented in the **Application Service**. Upon the request of the lecturer to recommend candidates given the selection criterion, we will employ the appropriate algorithm to recommend candidates. The algorithms are individual classes that take lists of application information and sorts these applications accordingly. We instantiate different algorithm classes depending on the request of the lecturer. All the specific algorithm classes implement the overarching interface `Recommendation`.

We believe that this does require different classes as we intend on introducing more complex selection criterion, such as combining the experience in a particular courses and overall ratings. Such algorithms cannot be implemented simply with standard `Java` libraries, without introducing high coupling. Moreover, it allows us to employ different sorting algorithms that require external information from other services. By decomposing these algorithms to different classes, we allow our code to be more maintainable.

4.2 Class Diagram of Pattern 1

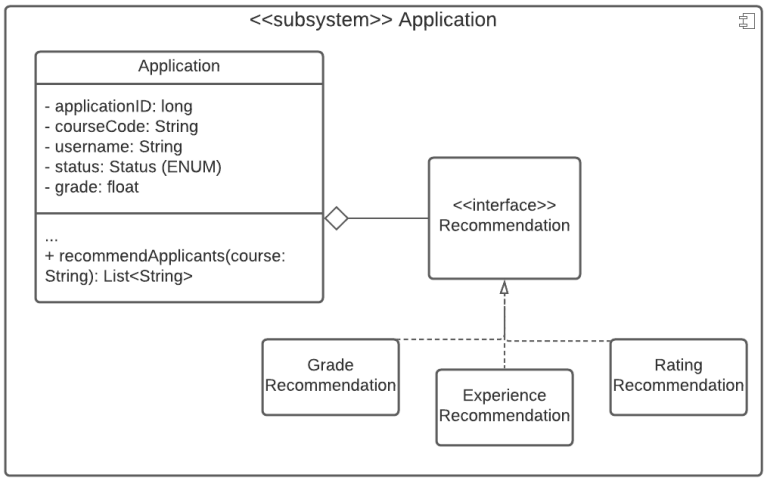

Figure 2: Strategy Pattern Recommendations

4.3 Chain of Responsibility - Spring Security

The Chain of Responsibility implies chaining the receiving objects and pass the request along the chain until one (or more) objects handle it. Immediately after acknowledging this definition, we thought that this formally describes the process of Spring Security which is formed by 2 main components:

- Authentication: process of verifying the identity of a user, based on provided credentials
- Authorization: process of determining if a user has proper permission to perform a particular action or read particular data, assuming that the user is successfully authenticated.

The main reason why we chose to implement this design pattern is because we need to handle a request through more than one object, the handler being determined dynamically. Also, the receiver and sender don't have to know each other, which is a characteristic of this pattern. How is this implemented? You can easily follow Figure 3 which explains the idea behind the chain:

- Firstly, the user needs to authenticate using their credentials.

– Then, if they are correct, we move to the next step which is authorization, where their role is being checked so that the system knows which permissions this user should have.

– If the access is granted we get to the validation component where the JWT is created and validated and sent to the user. This token will further be used for access.

– Lastly, the user is being successfully logged in.

– On success, access will be granted. If there is an exception, access will be denied and the user will receive a message accordingly.

For implementation check authenticationService in our repository. It relies on the whole implementation of the service, but it is well exemplified in the AuthenticationController in the '/login' endpoint.

4.4 Class Diagram of Pattern 2

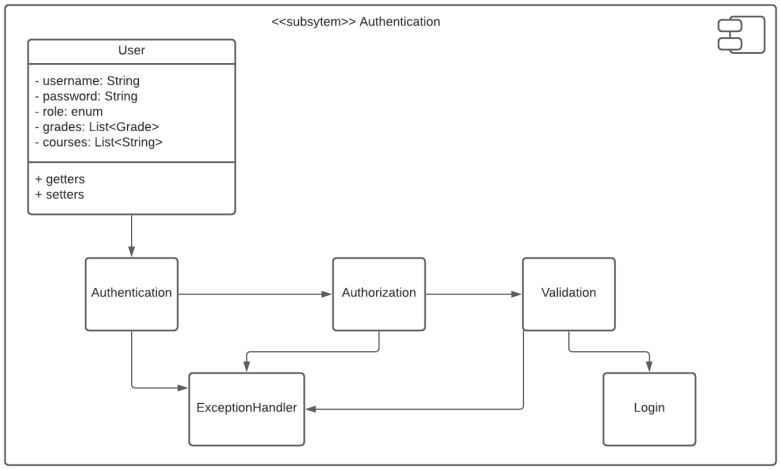

Figure 3: Chain of Responsibility Pattern