SEM Assignment 1

Group 14B

November 2021

Chapter 1

Task 1: Software Architecture

1.1 Bounded Contexts

The following domains will be mapped to micro-services in our system.

- **User**: A *core domain* that represents a user in the system. A user can be a lecturer or a student. The role of user can be used to determine how they can interact with other domains in the system (the actions they can perform). A student can have different roles for different courses.
- **Course**: A *core domain* that contains all relevant information to a course. This domain defines which courses are open for TA applications and maintains all relevant course information that can be send to students in TA contracts.
- **Application**: A *core domain* manages all information regarding a TA application. This domain encapsulates the whole process from when a TA application is created to when a student receives a work contract for their TA position.
- **Notification**: A *generic domain* that supports the process of a student receiving updates regarding their TA applications.
- **Authorization**: A *generic domain* that is responsible for authenticating users when they try to access the system and validating user actions throughout the system. Users are given different privileges depending on the role they are authenticated to.
- **Rating**: A *supporting domain* that allows lecturers to rate TAs. The rating system will allow lecturers to modify attributes for students who were TAs to their courses.

1.2 Bounded Contexts map Microservices

User

The purpose of the user microservice is to store all information relevant users of the system and to hand out JWT tokens that can be used to validate clients to microservices. The main responsibilities handled by the microservice are listed below:

- Authenticates users on login into the system.
- Handles invalid login attempts when the credentials are incorrect or a user doesn't exist in the system.

- Provides clients with JWT tokens that can be used to validate their requests to other microservices.
- Provides a `user` object representing a user of the system given the `netId`.
- Provides the grade of a user for a particular course given the course code and student number.
- Provides the GPA of a user given the student number.

Course

The purpose of the course service is to store all information relevant to the course domain. The course microservice also provides data about a course that can be used by other services to validate requests. The main responsibilities handled by the microservice are listed below:

- Provides a `course` given the course code.
- Provides any field in the `course` object upon request given the course code.
- Provides a list of all courses open for TA applications.
- Allows clients to add courses to the system.
- Adds hired students to a course to track the student to TA ratio for a course.

Application

The purpose of the application microservice is to handle all interactions related TA applications and finalizing the applications. This process is also facilitated by the notification service. The main responsibilities handled by the microservice are listed below:

- Allows clients to create an application (the created application will be returned to the client).
- Allows lecturers to accept an application.
- Allows lecturers to get a list of pending applications for their course.
- Allows users to retract their applications.
- Allows users to get the status of their application.
- Sends application result to the email of applicants.
- Sends contracts to accepted applicants.

1.3 Microservices

The architecture of the whole system is **service-oriented** since we are using microservices.

Bounded context mapping to microservices

Though we identified **six bounded contexts**, we decided to merge some bounded contexts into one microservice and have a total of **three microservices**. The primary motivation was to reduce the high coupling between micro-services. The criterion for merging bounded contexts are listed below:

- Merge bounded context when they will need to frequently send data back and forth if they were split into different microservices. If such domains were separated, their communication may overload the network when the number of requests are very high.
- Merge bounded contexts when one of them only interacts with the other, i.e. If X only interacts with Y, then merge X into Y. Since X won't interact with any other microservices, it can even be viewed as a method or function in Y.
- A bounded context should be a microservice when it can group together a highly cohesive group of concerns.

Course Microservice

Using the criterion above, the course context was made into its own microservice since it can be used to handle all requests relating to a course, despite the scarcity of business logic. On the contrary, if we didn't have such as microservice and relied on the user service, we would need complex queries to perform a simple query such as obtaining all TA's for course. Nevertheless, we could have kept one course table in another service, but this would cause this service occupied by requests not relevant to its own domain.

User Microservice

The user microservice is the aggregation of the user domain and the authorization domain. The domains were merged since due to the high amount of bidirectional communication. For example, the user and authentication domains have been merged because authentication relies heavily on data about users and user data needs to be authenticated. This microservice is necessary for tracking the roles of each user and giving permissions to resources.

Application Microservice

The application microservice needs to exist to achieve the primary aim of TA hiring. This service groups together all responsibilities in the process of creating an application to sending a contract. The notification bounded context supports application by sending an email when the status of an application changes. Notification does not not need to interact with other services or provide other functionality.

Internal Architecture of Microservices

Within each microservice, we use a **layered architecture**. In the microservice, there are controllers, services, repositories, and communicators. The `controller`'s are responsible for receiving requests from the client or other microservices. They will extract data from the requests and forward it to the next layer. In the next layer, the `service`'s contain all business logic, which includes checks for whether the request can be executed and then logic to execute the requests. It will also be responsible for invoking the methods in `communicator`'s to communicate with other microservices. The `repository`'s are responsible for persisting the changes caused by the request.

Security Architecture

We assume that all user who are permitted to use the system have their netIds and passwords saved in a database accessible to the system, meaning that new accounts can not be created.

The authentication component of the system is in the user microservice. When a user provides their credentials (netId and password), the system checks whether this user exists and whether the credentials are correct. Once a user is successfully logged in, the system will determine its role as this will decide what resources they can access.

After login, the user service will return two JWT tokens to the client. The first is an access token that is valid for 15 minutes. The second is a refresh token that is valid for 2 hours. Both tokens are generated with hashcodes. Once the access token expires, another access token can be generated using the refresh token through the user service. When the refresh token expires, the client will need to provide their credentials again.

These two tokens need to be kept by the client and sent in the header of each HTTP request to any of the microservices. Microservices in the system will share the same secret key so that they all can check the validity of a token in the same way. Each microservice will have filters to accomplish this.

The strength of this architecture is that we don't need to store any tokens in a database for each microservice to validate it. Moreover, a security check per microservice is more flexible since we have the option to use different security approach per microservice. This will not be possible with a gateway that generalises over all microservices.

Communication between services

Communication between microservices is **synchronous**. Micro-services request data from each other through REST endpoints using HTTP. Though some requests, such as creating an application or accepting an application, may need to be processed by many services before it is finalized, we chose synchronous communication due to limited resources.

For each service that a microservice has to communicate with, there is a class that will handle all communication with that microservice. The port and url of that microservice is hardcoded in a variable of the class. For example, the **application** microservice has a `UserCommunication` class that handles all communication with the **user** microservice. Methods for creating all necessary HTTP requests are all in this class.

Outline of microservice implementation

The service-oriented architecture is implemented by keeping each service as its own project. All microservices are placed inside a `package` of the main project. Inside the package, each microservice is a `module` that is its own gradle project. Since each microservice has a layered architecture, we keep each layer of the service in a different `package`. In addition, the communication classes and exceptions are also kept in its own package.

1.4 UML Component Diagram

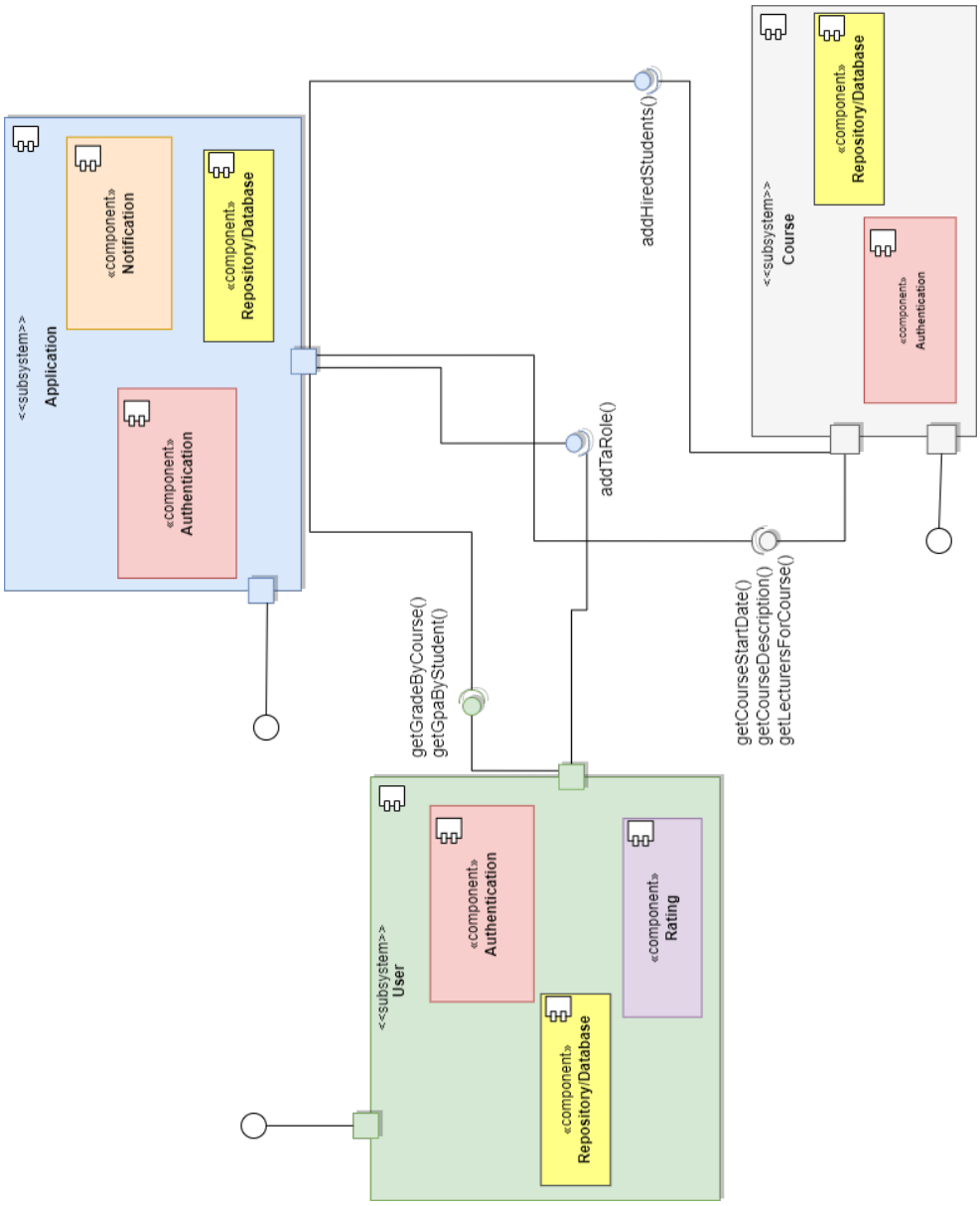

Figure 1.1: UML Component Diagram (excluding design patterns)

Chapter 2

Task 2: Design Patterns

2.1 Design Patterns

2.1.1 Strategy Pattern

We use this pattern in the Application micro-service to recommend students as TA's to lecturers judging by different factors (grade, experience, rating).

We have chosen this design pattern as we deem it an appropriate way to interchange different constrains without duplicating code and to have the possibility of easy extendibility (adding a new way of recommending students). It gives a lecturer more freedom to choose the TAs that they feel are appropriate for their course by selecting the criteria that are the most important to them. This pattern also help us reduce conditional statements as there is no need to check manually what type of strategy we are dealing with.

The strategy pattern also makes our code cleaner and more efficient as it removes the need for sub-classes by implementing an efficient interface instead.

Class Diagram of Strategy Design Pattern

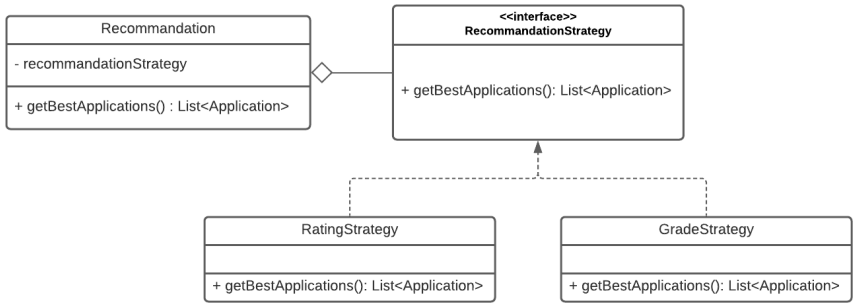

Figure 2.1: Strategy pattern in application microservice

2.1.2 Chain of Responsibility

We use this pattern throughout our system - though currently its fully functional and tested on the user microservice. When a user logs in successfully, this user gets a JWT token. This token is validated at every request the user makes through a layer of filters. If this validation is not successful, an exception is thrown (exception depends on which part of the validation fails).

We have chosen this design pattern to be able to authenticate our users, and validate them each time they make a request to the server. We do this in order to guarantee the safety of the server and database. This pattern provides a natural way to validate the user through a layer of different filters that check if the user is actually the owner of the JWT token and if the content of the JWT token hasn't changed, without knowing how the actual request should be handled and without knowing the actual user so that the user only has to authenticate themselves once throughout their session.

Spring Security already contains some necessary filters which are called at every request. We implemented one of their interfaces to extend security of each micro-service. The general structure remained, each filter knows only about its successor and each filter has its own responsibilities. We added 2 filters for User MicroService which are JWTPublisher (to publish the JWT token for authenticated users) and JWTFilter (to check validness of the JWT token for each request). For the other microservices we added one filter to only validate the JWT token. When the JWT token is invalid we block access to any resources for that specific user (throwing an exception).

Class Diagram of Chain of Responsibility Design Pattern

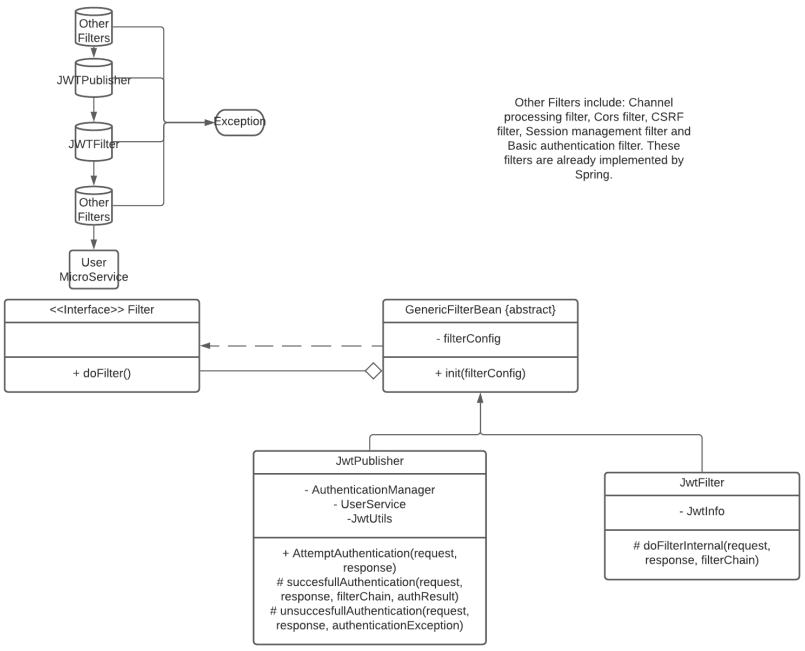

Figure 2.2: Chain of responsibility design pattern in user microservice