Assignment 1 - Software Engineering Methods
Group 17A

Andrei Tociu, Pauline Hengst, Gijs van de Linde
Iarina Tudor, Vincent van Vliet, Ravi Snellenberg

December 2021

1 Software Architecture

1.1 Introduction

We were assigned to make a TA hiring system for the Delft University of Technology, using a microservice architecture. In short, the goal of the system is to simplify the TA application process, to assist in finding and hiring the best assistants for courses and to manage TA's payment.

We started designing the system by formulating the exact functionalities into requirements, from where we divided the system into microservices using domain driven design. The next sections will describe how we used domain driven design, and which microservices this resulted in.

1.2 Domains and Microservices

The goal of domain driven design is to split the main problem into bounded contexts which encapsulate subdomains with clearly defined boundaries. In terms of logic, we could split the main problem into three subdomains: involved parties, type of processes and data distribution. We find suitable bounded contexts by splitting subdomains such that they have as little overlap as possible.

The involved users can be divided into the lecturers hiring assistants and the students becoming TA's. These parties have a completely different view of and interactions with the system, as they have different functionalities available to them when it comes to applying and hiring for TA positions.

The processes in the system can be divided into the process of hiring TA's and the process of managing existing TA's. Hiring TA's can further be split into: allowing students to apply, finding and selecting the most suitable TA, generating a contract and the notifications about TA changes. Managing TA's can be split into logging a TA's work hours and keeping track of a Ta's reviews and experience.

Data can be split into three parts; User data (describing users of the system), Application data (describing pending applications), TA data (describing the existing TA's their contracts and notifications). Application data and TA data overlap in that they both use data describing courses (eg. which student follows which course, starting and ending dates of courses in relation to applications opening and closing). Therefore courses can also be split into another data domain.

These bounded contexts are not really the end goal itself, but they are supposed to help dividing the system into different microservices. When deciding on the microservices, there is more to take into account than just separation, since microservices treat domains like an independent system. It would be ideal to have only one entry point. Therefore, most of the authentication and protection can be focused on one spot. It would also not be practical to have a microservice for a tiny domain. Below we created a short overview of the microservices we decided upon and their functionalities. We also included some motivation of choosing only these microservices.

1.2.1 User

1. **Microservice :** Represents the entry point for users to the system. Manages user accounts and all their possible actions. Stores user account data. There are three types of users with different privileges: Student, Lecturer, Admin. Manages user authentication when a user logs in to the system. When a user logs in, checks if there are unread notifications.

2. **Interaction :** This microservice acts as an entry point and interacts with all other components. All users send further requests to retrieve and alter data from other services. Students send requests to submit applications, lecturers ask for applications overview and decide upon which students to hire and admins manage the databases for each service. The User microservice sends requests to other components and the others only respond to them. The TA microservice also offers notifications to this service and allows for declaring/approving TA's work hours.

1.2.2 Application

1. **Microservice :** Manages process of applying for a TA position in a course. Stores all the active applications. Creates applications when a student applies for available courses, on request of User and using data from Course and TA. Triggers TA's creation when an application is accepted by the lecturer. Provides applications per course for lecturers upon request. Can recommend certain TA's to lectures using the data in the applications. Triggers notification creation when selection is done.

2. **Interaction :** This microservice also interacts with all the others. From the users it accepts applications, approve them if the lecturers indicate so. It also retrieves data from the Course and TA microservices in order to create an overview of applicants information and recommend them if requested by lecturers. Therefore, it gets requests from lecturers to see a candidate profile and forwards the requests to the others microservices to retrieve extra data for that profile - since an application only holds the data submitted by the student, the rest information is requested from other components.

1.2.3 TA's

1. **Microservice :** Manages existing and past TAs. Creates a new TA an application is accepted. Keeps track of logged hours per TA for current contracts. Lectures can rate TA's, for their courses, based on their experience. Stores the ratings. Stores contract information. Creates contracts for TAs when they are hired and forwards them to the owners. Stores notifications of the selection procedure results, which will later be visible to the users. Manages the past TA's actual spent hour indication for future candidates.

2. **Interaction :** This microservice interacts with Application, since once an application is accepted, a TA and a Contract are created. The contract can be modified by the User's requests, in case its role is lecturer. It sends notifications to the User microservice and accepts hour declarations and approvals from the same microservice. It does not interact with the Course microservice.

1.2.4 Course

1. **Microservice :** Stores all courses including past editions. Stores the grades students have gotten for those courses.Admin can create courses and add grades for students per course. Stores course start and ending dates. Provides information to other micro services when needed.

2. **Interaction :** This microservice offers, to the Application, information about the grades of the students and which courses are open for receiving applications. It also interacts with the User microservice by acting upon course creation and grades management by admins. Although, its main purpose is to allow applications to be submitted.

Courses needs to exist, so that the students can submit their candidature for the role of TA. This service does not interact with the TA component.

1.2.5 Common features

Microservices communicate using an API. This API handles the sending and receiving of requests but also checks if the user is actually authorized to execute the desired request. If this is not the case the API will not forward its request to the corresponding microservices. This is done as an extra layer of protection for each microservice.

Token validation is necessary for every microservices, because we do not want malicious requests to be executed by the system. Thus, we decided to have a common package that will include a validate function handled by Spring Security. This method will allow only for valid token users to perform requests. We will add the Spring Security to the shared module, so that it can be available across all microservices. Therefore, authorization for roles and token validation will be possible for each request.

We also realised sharing data across microservices is necessary. Of course, only some data should be shared since our desire is to create a Microservice system, but a perfect high cohesion architecture will require an incredibly amount of code duplication. Therefore, we addressed this issue by making a compromise and allowing a level of coupling. We will design these Data Transfer Objects inside the common package and they will help us retrieve information for application overview (student grades , TA's rating)

1.2.6 Binned domains

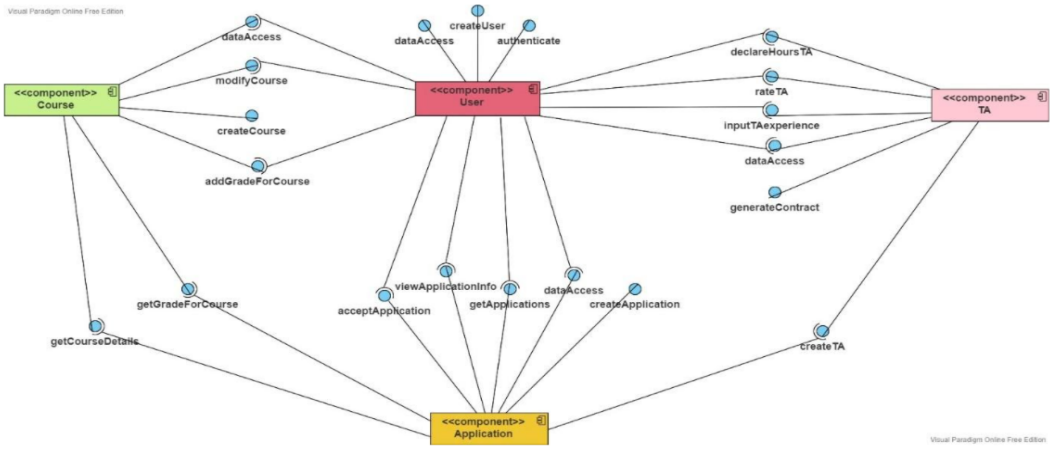
At first we decided to have 6 services, the ones we have shown with two additional microservices: Contact and Authentication.

1. Contract would have managed contract generation for the TA service and would forward the contract back to the User service. This could be a valid approach as the process of generating a contract is very specific and could be separated from the rest of TA's functionality, but we decided to incorporate it into the TA. This is motivated by the fact that this microservice is used only by the TA's one and shares a lot of data with it. Thus, there was no need for a separate database that will store redundant information.

2. Authentication microservice would have managed authentication for the users. We thought it could be a good approach to have it separately to provide more separation. Since Spring Security is a requirement and we are not building security from scratch, we decided to incorporate this into the User micro service. We will only handle login into accounts already existing in the database; registration we assume is done by another party: TU Delft. This approach will end up with having one secure entry point for users and after validation, they will already be able to perform desired requests.

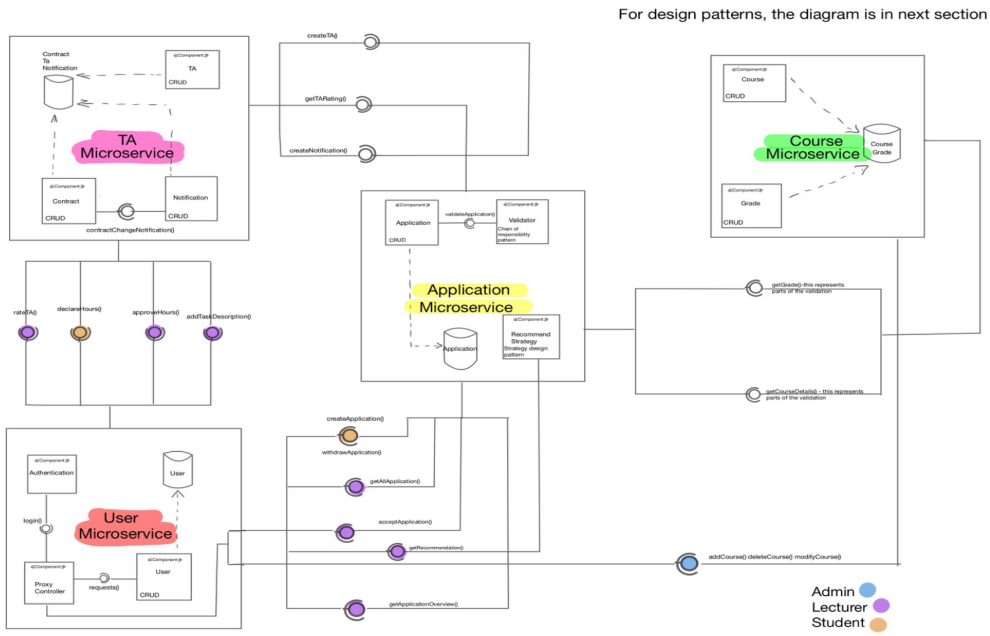
1.2.7 High Level Diagram

Here we present a high level UML diagram, displaying the components and their basic interaction.



1.2.8 Low Level Diagram

This is the low level diagram of the system. Each component has its own database, entities, controllers and services. Requests performed by the users have different colors indicating the role authorized for the specific request. For the subcomponents which include design patterns, the diagrams are available in the next section.



2 Design patterns

There are two design patterns that will be used in the application: The strategy and chain of responsibility design patterns.

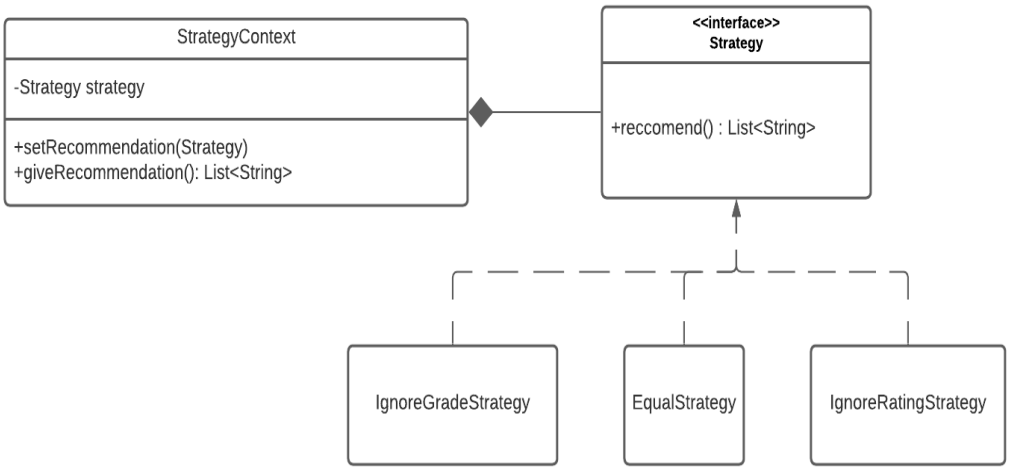
2.1 Strategy design pattern

Our system had the functional requirement "Lecturers can receive recommendations of prominent TAs to help them in the selection procedure." To satisfy this requirement, we decided to model a recommendation system. This system should be able to recommend pending applications in a way the lecturer finds useful, which can change with the wishes per lecturer.

To achieve this behavior we decided to use the behavioral Strategy design pattern, which in our case determines strategies for recommending TA's per recommendation request from a lecturer.

We implemented the design pattern in the Application Microservice, as this has direct access to application data. The implementation works as follows: we have a class StrategyContext with a strategy as an attribute, and methods giveRecommendation() and setRecommendation(). setRecommendations() is used when processing a request to set the desired recommendation Strategy. The strategy interface requires a valid strategy to be able to return a list of strings by recommend(). The StrategyContext will use its strategy's implementation of recommend() when giveRecommendation() is called, so the returned list has applications as recommended by the desired strategy.

When requesting recommendations, a lecturer can choose a strategy to recommend by. The interface Strategy is really simple and therefore flexible in how a strategy can be implemented. The strategies we currently have implemented work by weighing the grades and Ta rating of an application differently. For example IgnoreGradeStrategy will determine the best TA solely based on the applicants' TA ratings, and EqualStrategy will weigh grade and Ta rating equally. By nature of the design pattern, more recommendations strategies can be added and removed, so we are able to add more strategies if we want more functionality from our recommendation system.



2.2 Chain of responsibility design pattern

The chain of responsibility design pattern will be implemented in the Application microservice. It will be responsible for checking if an application can be submitted. The requirements we will use are the following: passing grade, course existence, course availability of accepting applications and unique application submissions(to make sure a student is not submitting duplicate applications). Having all these requirements checked by separate classes in a chain of responsibility reduces coupling and it also easily allows for adding extra requirements in the future. This can be done by adding new classes to the validation chain.

We design it by creating an interface called Validator which is implemented by the classes that are part of the chain. In this interface we specify 3 methods: handle, setNext() and setLast(). Handle is the method that is responsible for performing the task that specific class is created for. SetNext() and setLast() are both responsible for adding another Validator class to the chain. Since every class should be able to use these methods, we created an abstract class (from the validator interface) that actually gives an implementation of those methods. The abstract class also has an extra method called checkNext() which is responsible for traversing the chain and returning true if the end of the chain is reached. All the classes in the chain override the handle method for their own tasks.

This is the diagram of the chain of responsibility pattern we implemented.