Delft University of Technology

SUBJECT: Software Engineering Methods
COURSE CODE: CSE2115

Assignment 1

Authors:
Ben Carp
Denis Corlade
Andrejs Karklins
Daniel Poolman
Justin Rademaker
Auke Sonneveld

OP15-SEM29
January 8, 2021

1 Software Architecture

The application we have written relies on microservices. Microservices can be seen as smaller independent programs that each has a specific functionality. All microservices together form the complete application. Firstly we will give an overview of the different microservices our application has and how these microservices interact. After that we will go more in depth about each specific microservice and its functionality.

1.1 Microservice Architecture

One of the hardest requirements we had to deal with was having multiple microservices in our online assessment service, as we were guided by the non-functional requirements of the program. The problem was that there were some ambiguities within the team in regards to how many microservices to split up on and more precisely, on which of them, such that we could have a scalable system. If there is a need of a new component, it would be easy to expand on it in our program as the microservices are largely independent from each other, thus making the whole process much easier. We thus split the service into six microservices we considered as the main components needed for our online assessment application: the Authentication Service, Authorisation Service, Student Service, Teacher Service, Course Service and Exam Service. The way our microservices interact with each other can be seen in Figure 1. When a client first accesses our service, he or she needs to sign up or log in. This can be done by accessing Authentication Service. Assuming the client succeeds in logging in, he or she gets back a token and depending on whether it is a teacher or a student, the client accesses either the Teacher Service or the Student Service respectively. The Teacher Service and Student Service don't contain any business logic but instead redirect the call made by the client to either the Course Service or the Exam Service, depending on where it needs to go. Before executing the call made by the client, the Course and Exam Services will check whether or not the client is authorized to execute the made call. Included in every call is the session token the user received from the Authentication Service when logging in. Using this token the Authorisation Service is called to check if the user is authorised.

1.2 Authentication Service and Authorisation Service

When a user wants to use the application the first thing it needs to do is log in. He or she can do so by sending their credentials, consisting of their netId and their password, to the Authentication Service. All user login credentials are stored securely in the database connected to the Authentication Service by using the 'bcrypt' algorithm. It will check whether or not the credentials are valid and if so, it will send back a Java Web Token (JWT). This JWT is a way to encode certain information, such as the users netId and role, in a secure way. In our JWT-session we used the HMAC-SHA-512 Algorithm which makes use of a hash function, as well as a single key (called secret) to encode the information. When successfully logged in, the user wants to make use of other services and in order to do so, it sends along this JWT with every call it makes. The other services need to verify first if the user is allowed to perform certain actions. Therefore, it will send the JWT to the Authorisation Service, together with the required role for the action. The decoding of the JWT requires the same secret as for decoding. The Authorisation Service then checks if the required role is higher or equal to the role stored in the session and if the session is no more than a 24 hours old. If so, it sends back a permission. The other services can now finalise their actions and send back the response to the user.

1.3 Student Service and Teacher Service

The Student Service and the teacher service both act as a front-end for users of these respective roles. By only exposing the methods these users are allowed to access, they mainly serve the purpose of abstraction. They also provide consistent mappings to the user, who no longer has to worry about the implementation of the course and exam services, and include detailed Javadoc comments on what these methods accept and return. The course and exam services handle business logic and database access, and check whether a user has the correct role to be authorized for a certain task. That is necessary, since nothing prevents a hacker, or a technically inclined but still unauthorized legitimate user, to monitor traffic and use the endpoints exposed by the course and exam services directly. The abstraction provided by the student service (and also the teacher service) is therefore not meant as a security-by-obscurity feature, all that happens is the forwarding of http requests to the right service and subsequently the relaying of that service's http response to the user. To the casual observer it might seem as if these services are completely redundant for the operation of the online assessment system, and they would be correct in a strictly technical sense. We have asked ourselves the same question in our development team, and came up with multiple reasons to justify keeping the student and teacher services as independent microservices: For one, they will make life for a hypothetical UI or GUI developer a lot easier, as the comments and selection of methods in the student and teacher services offer a blueprint of which functionality to expose to whom. For another, this split of functionality follows the concept of the facade software-design pattern quite closely, which is covered in greater detail in section *2.1 Facade pattern*.

1.4 Exam Service

The main goal of the Exam service is to carry out the process of taking an exam as a student. Furthermore, the Exam service is will take care of statistics for the exams as well as creating an exam, something that can be done only by teachers. It also includes CRUD operations for ExamQuestions, ExamAnswers and StudentExam. Students are only allowed read-access permission while teachers have control over the whole CRUD operations. The Exam Service has a direct connection with the Course Service which it uses to retrieve Questions and Answers. The Exam Service calls the Course Service passing as a parameter only a courseId. The Course Service is then responsible for returning 10 Questions and 40 corresponding Answers related to the topics of that course. These are needed to create an Exam for the students. For every topic related to the course, there is at least one question returned from that topic so that all topics are covered in the exam. Besides that, students are able to see their grades immediately after submitting their assessment as the grade is automatically calculated after the 20 minutes have passed or the assessment has been submitted. Not only that but they also have the permission to see their answers and the average grade of the exam. Regarding statistics, teachers are able to get the average grade by exam among all the students, get the individual grade of each student, see the two worst answered questions in the exam, while also being able to see the total number of students participating in the exam.

1.5 Course Service

The purpose of the Course Service is to take care of anything and everything related to Courses. This includes the Topics that belong to a Course, as well as the Questions and Answers related to these Topics. Furthermore, the Course Service handles Enrollments for Courses. The decision to split up the logic in this way between Exam Service and Course Service is based on functionality. As explained in section *1.4 Exam service* the Exam Service main goal is to handle the taking of an Exam. Everything else is handled by the Course Service. This includes CRUD operations for Answers, Questions, Topics, Courses and Enrollments. Students are only allowed read-access whereas Teachers are allowed full CRUD access. The only exception here is that students are allowed to enroll and unenroll themselves in courses. Furthermore the Course Service provides the functionality of creating and returning a StudentExam when given a CourseCode. The Course Service fills this StudentExam with ten Questions and forty Answers as described in section *1.4 Exam Service.* This method is called by the Exam Service when a Student wants to take an Exam. Linked to the Course Service is a database where all previously mentioned entities are stored and persisted. Within the Course Service we have Controllers, Services (these are Spring Boot Services and should not be confused with the microservices) and Repositories. When a call is made to the Course Service it arrives in the correct controller. This controller retrieves the relevant data from the HTTP Request and with it calls the corresponding service. All business logic is implemented in the services. When the service needs to access the database it uses the repositories for this. When any method in any service is called, first the authorization is checked. If the user making the call is not authorized the call will not be executed.

1.6 Communication

Until now we have talked about certain microservices calling other microservices and passing information, but we have not explained how this actually works. All communication between microservices, as well as between the client and the microservices, is done using Http Requests and Http Responses. The JWT-session is encapsulated in the header of each Http Request made. Any data that needs to be send with the call is included in as a Json String in the body. When a teacher for instance wants to create a new Answer, he or she includes a Json representation of the Answer to be created in the body of the Http Request. All endpoints are using Post Mappings. The reason for using Post Mapping regardless of the type of operation we want to do is twofold: firstly this is to obtain a uniformity across all calls which removes a level of complexity, secondly this allows us to retrieve information without passing variables in the url-path. If instead we were to use Get Mappings for retrieving information we wouldn't be able to pass information in the body, since Get Requests don't have a body. This way the format for making a call is always the same. When a microservice sends back a Http Response there are two things to look at. The first is the HTTP response status code. This indicates whether or not the operation succeeded, and if not what went wrong. Examples of this are 200 (OK) and 401

(Unauthorized). Furthermore, if the operation succeeded, similar as before data will be passed in Json format in the body of the Http Response.

2 Design Patterns

2.1 Facade Pattern

One of the already implemented design patterns is the Facade. The microservices 'StudentService' and 'TeacherService' make up the facade. The implementation can be found in the homonymous modules in the source code.

The main reason to implement this design pattern is that we deliver an application without a client-side implementation or a Graphical User Interface. Yet, it should be easy to use the application directly, with little knowledge about the internals. Also, it is probable that a client-side application will be written in the future. This design pattern facilitates that process.

To reach these goals, the facade has been split up based on the student and teacher role. It could be extended to more roles in the future. Every microservice contains only methods that are relevant for that role. The microservices do not contain any business logic themselves. They simply route the request to the relevant business logic. This is done by sending HTTP-requests to the proper endpoints in different microservices.

There is a noteworthy difference between the traditional Facade design pattern and our implementation. Traditionally, the user will not be able to access any functionality behind the facade. This might be desirable for security reasons. However, in our implementation, users can easily bypass the facade and this is not a problem. The main goal of our implementation is to facilitate the user and improve the ease of use. The user may choose for itself if it uses the facade.

Finally, a UML class diagram for the facade has been made. These can be found in figure 2 and 3. Only the method names are shown. The parameters and return type have been omitted since these are the same for any method. The parameters are a JSON-String of data in the body and a session-token in the header. The return type is a ResponseEntity<String> for every method.

2.2 Method Factory pattern

To improve the application, the Method Factory design pattern has been implemented for the serialization and deserialization of the entities.

The original implementation supported (de)serialization with regards to JSON Strings. However, this functionality was not scalable. Adding extra formats like XML of CSV would lead to large code refactorings and high maintenance costs. The new design pattern has been used to solve this problem.

The UML class diagram in figure 4 represents our implementation. it contains a 'Serializer' interface and an 'SerializerFactory' abstract class. To create a serializer of any format, the SerializerFactory should be extended with a concrete class. This class can then create the actual serializer object. This object will be an implementation of the Serializer interface, meaning that they have implemented the serialize and deserialize method.

With factory methods implemented, it is very easy to change the way of serialization. Only one line of code needs to be changed: the factory. All occurences of the 'createFactory' method will dynamically create the class corresponding to factory that was chosen.

3 Images

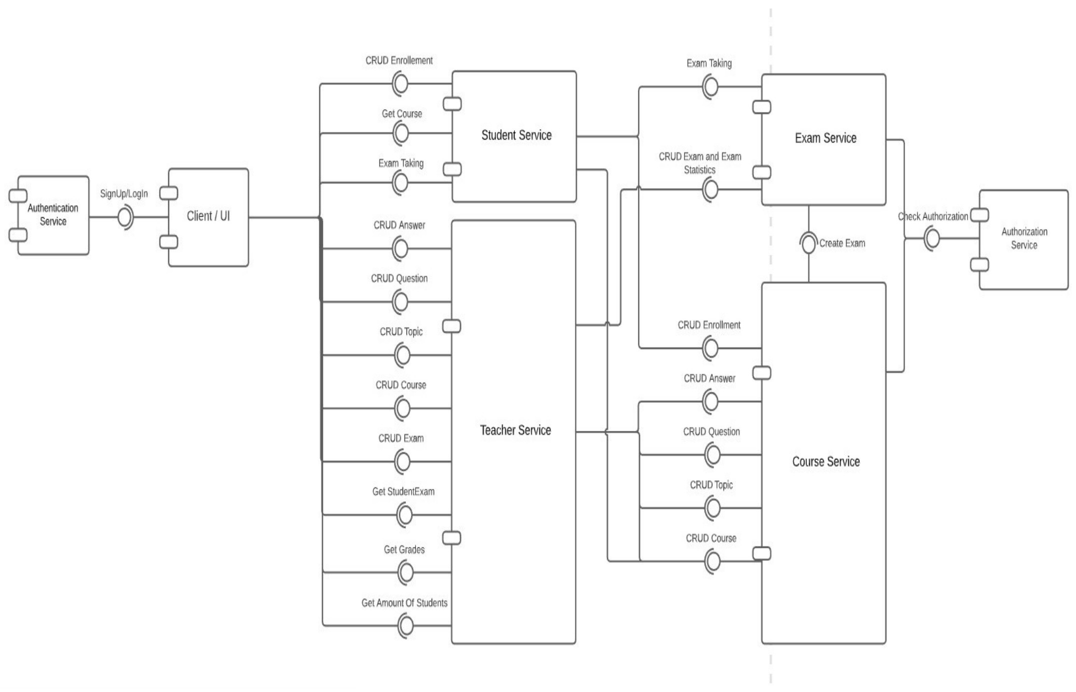

Figure 1: UML Diagram of our system architecture

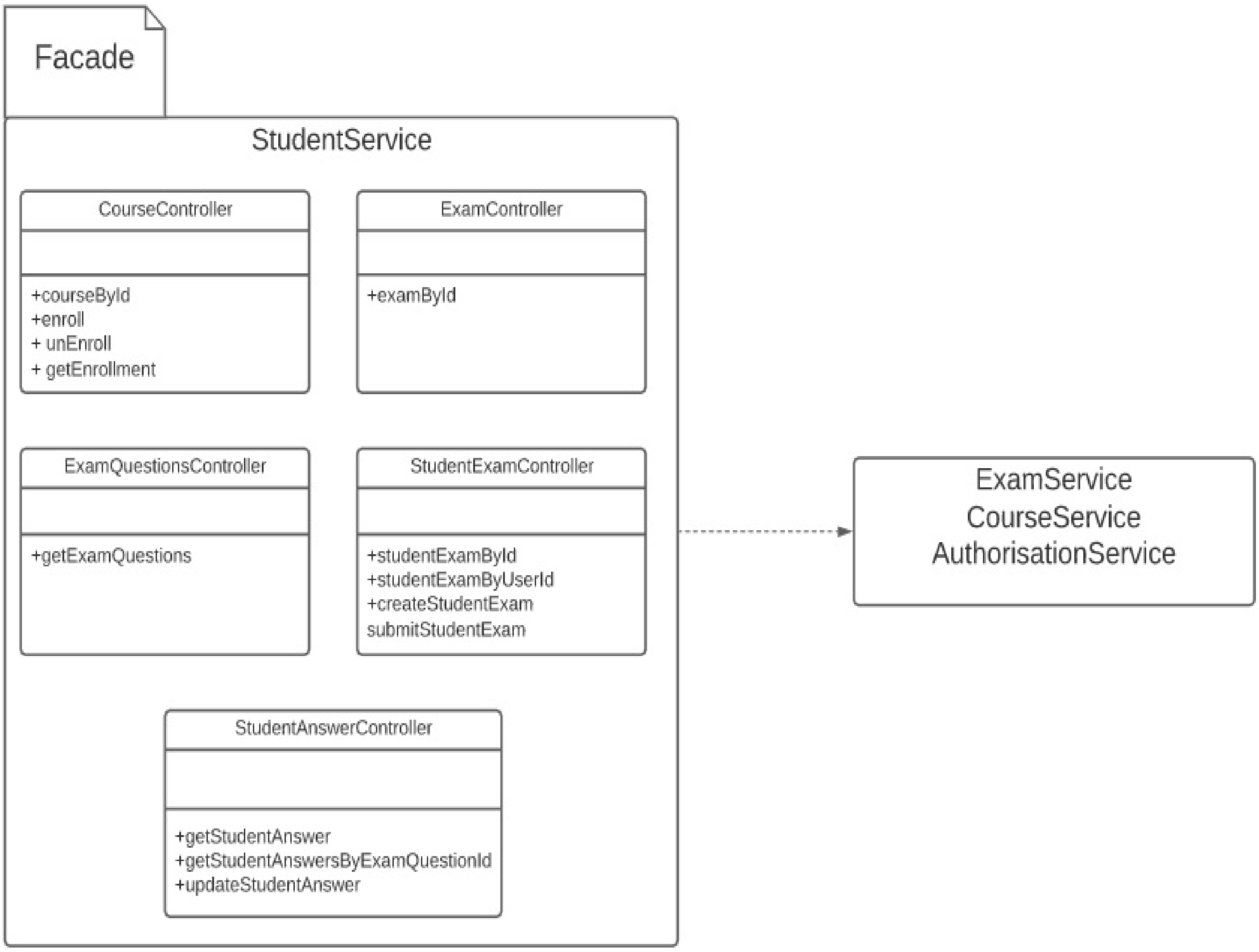

Figure 2: Diagram of the StudentService Facade pattern

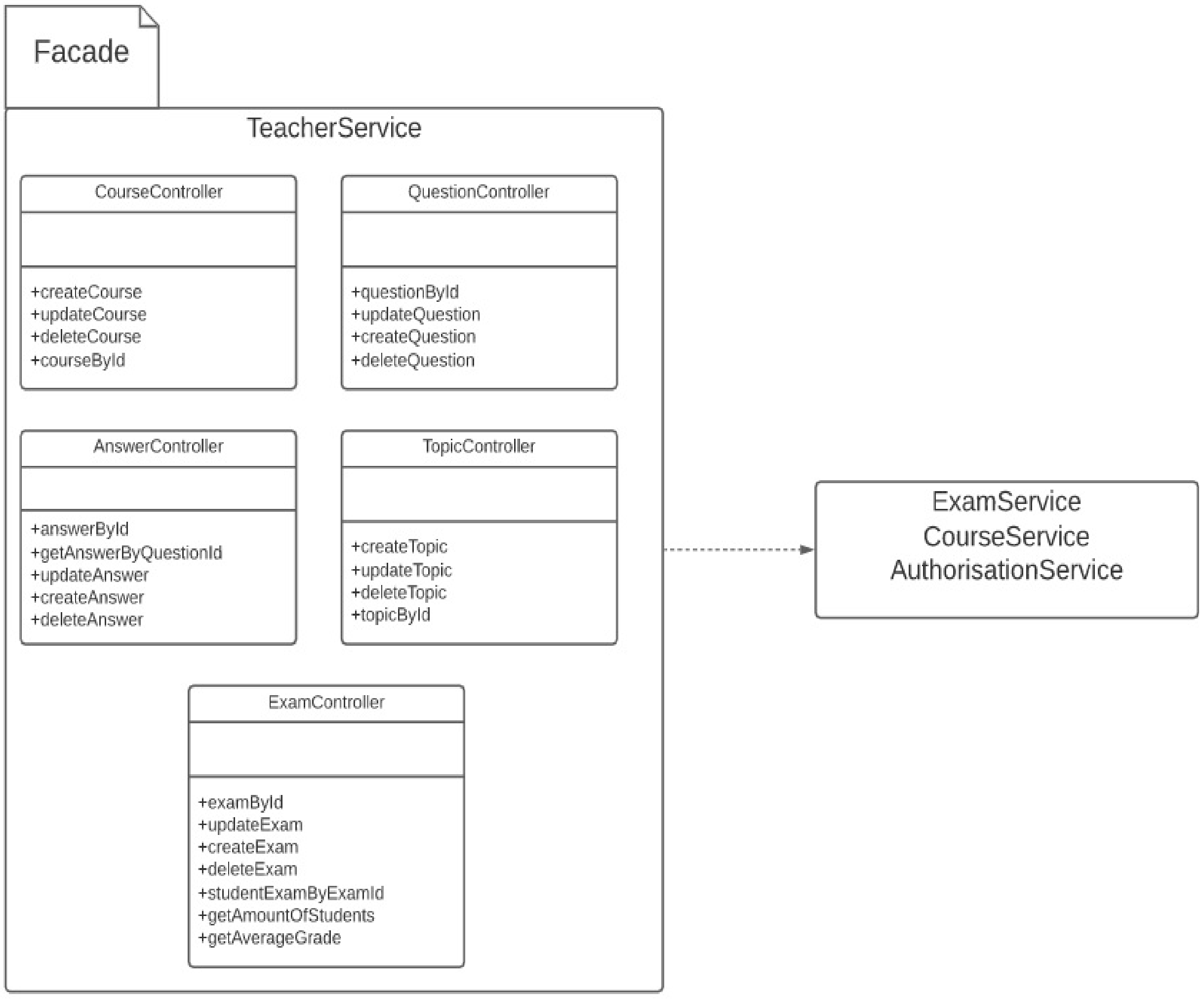

Figure 3: Diagram of the TeacherService Facade pattern

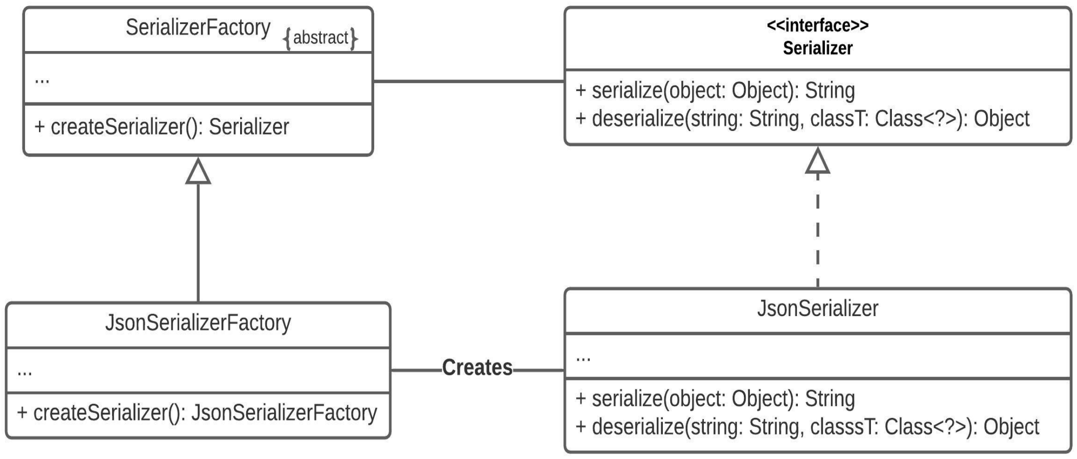
SerializerFactory {abstract}
...
+ createSerializer(): Serializer
<<interface>>
Serializer
+ serialize(object: Object): String
+ deserialize(string: String, classT: Class<?>): Object
JsonSerializerFactory
...
+ createSerializer(): JsonSerializerFactory
Creates
JsonSerializer
...
+ serialize(object: Object): String
+ deserialize(string: String, classsT: Class<?>): Object

Figure 4: Diagram of Factory Method pattern