Assignment 1

OP10-SEM20

1. Software Architecture

As the component diagram of our system shows (figure 1), our application is divided into three microservices: the Authentication Server, the Board Server, and the Content Server. Each of these is built using the Spring framework: there is a repository which simplifies communication with the database, a service which handles the logic of processing a request, and a controller which calls the service methods and provides API endpoints. Details regarding each of the microservices and how they communicate with each other are outlined below.

1.1 The Authentication Server

Responsibility. The Authentication Server handles all operations and requirements related to users: registration, logging in, and security. Special request classes are used as parameters for methods which handle registration and sign in. After registering, a user can log in using their email and password, which is hashed and matched with an entry in the database, to determine if the login is successful. Afterwards, a randomly generated token is associated with the user while he/she is logged in, and discarded when he/she logs out. To enable this, the *UserService* interacts with a repository responsible for authentication tokens.

Role in the overall system. To communicate with other microservices, the methods of the *UserController* in the Authentication Microservice return special response objects, which are included in the shared library module of our project. Each of the other microservices knows how to interpret these and determine the identity and role of a user. Likewise, all these other microservices send special request objects, containing a token, to the Authentication Server, which can extract information about a user given this token.

The component diagram of our system shows that the Authentication Server provides an interface for the other two microservices, but does not require any interface. As is described in their corresponding sections, the other two servers send messages to the Authentication Server to assess the permissions associated with a user token, and its validity.

Motivation. There is a clear distinction between the Authentication Microservice and the other two microservices in our application, namely that users should be separated from the content. Naturally, there are many responsibilities related to users, which are summarized at the beginning of this section, and their number and complexity can only grow as the application becomes more popular and more widely used.

Decoupling the Authentication Server from other microservices also means that different teams can work on different microservices, without having to communicate much with each other. The Board and Content Servers can use the interface provided by this microservice while being unaware of the internal workings of token generation and security.

1.2 The Board Server

Responsibility. The Board Microservice enables creation, retrieval, and updating of these boards. It is also worth mentioning that the parameters passed to the service methods are not in fact instances of the Board class, but of other classes created specifically for requests, which sometimes omit certain details such as the id of the board, or the user who created it.

Role in the overall system. The Board Microservice communicates with other microservices. These relationships are illustrated by the component diagram of our system (figure 1).

Firstly, we can see that the Board Server requires the interface provided by the Authentication Server. We need to check that a user trying to create or edit a board is a registered user, and that they have permission to do so. The *BoardService* class uses Spring's *RestTemplate* to make calls to the Authentication Server. As explained in the section related to the Authentication Server, the responses are objects from the shared module, and they provide information for the Board Microservice to determine if the user is a teacher, or if he/she is the creator of a board. Unsuccessful requests throw exceptions, which are handled by a *ControllerAdvice* class. An adequate *ResponseEntity* is returned, containing the error message, or, in case of success, a confirmation, and an HTTP status code.

Secondly, the Board Microservice provides an interface for the Content Microservice. This consists of one method, which specifies whether a board is locked or not. A board that is locked would no longer allow the creation of threads associated with it.

Motivation. There are many ways to justify the need for a standalone microservice for boards. Unlike threads and posts, which represent the content, boards are mostly used to organize this content depending on its topic. Hence, they are not as tightly coupled as threads and posts, and, while they certainly play an important role in making the forum more readable, an early prototype could be released even if the Board Microservice is not fully implemented. We also thought about future developments of our application, and one that naturally comes to mind is to introduce one more component: courses. It makes sense that each course has multiple boards (perhaps for each lecture, or each week), and the Board Microservice could thus be updated to take on more responsibilities. Had we implemented boards, threads, and posts as one server, this addition would make it overcrowded.

As explained in the section regarding the Authentication Server, it also benefits the maintainability of the system to keep the users in a separate microservice, therefore it is clear why boards and users are separated.

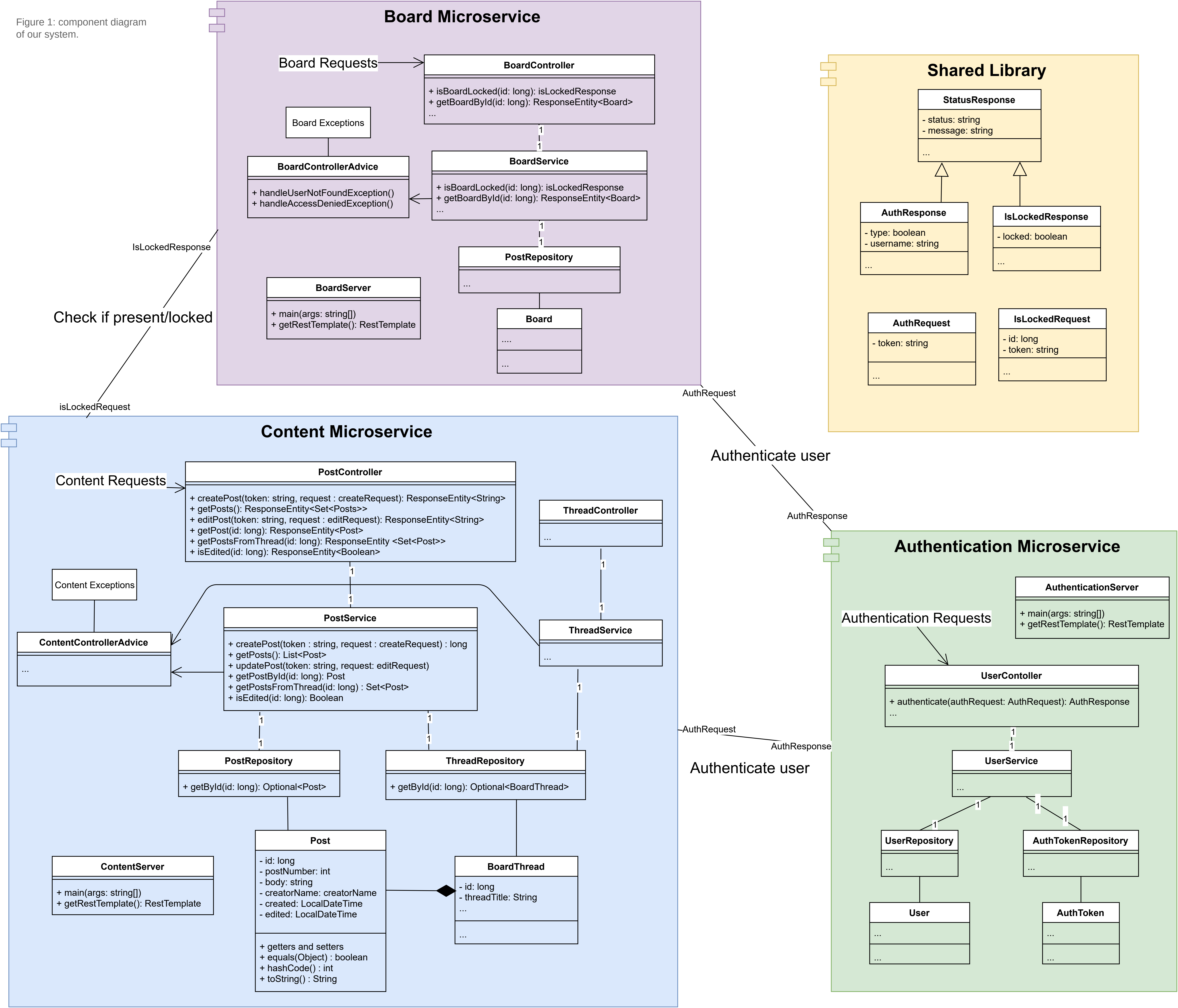
1.3 The Content Server

Responsibility. Threads and posts represent the Content Microservice. This service enables creation, retrieval and updating of threads and posts. An unauthenticated user is only able to retrieve threads and posts. An authenticated student can retrieve, create, and update threads and posts. An authenticated teacher, in addition to the operations that a student is able to perform, can also lock and unlock threads.

Role in the overall system. The Content Microservice communicates with all the other microservices. From the diagram (figure 1) we see that it requires interfaces to be provided by both the Authentication Server and the Board Server. This, in addition to its internal responsibilities, makes the Content Microservice a crucial microservice in our project. The *PostService* class actively makes use of a *ThreadRepository* when creating, updating, and retrieving posts per thread. This indicates that the decision to put posts and threads into one microservice was right, as otherwise posts would have to make requests to threads over the network. Both threads and posts use request and response classes from the shared module: one for requests to boards and one to enable communication with the Authentication Server. Authentication is needed to validate the user and their actions, in case they want to create or update threads or posts. The Board Microservice is needed to allow creation of threads and posts, depending on whether a board is locked.

Motivation. The decision to put threads and posts together in one microservice is intuitive, because, in principle, a thread is simply a chain of posts. This makes threads and posts tightly coupled. Though it can be argued that we could have put boards, threads and posts together, as they represent the content (which would give the resulting microservice too many responsibilities and decrease modularity, scalability and make distributed development harder), we definitely had to create a separate Authentication Microservice, as it contains everything related to authentication and users, which are independent of the content of the forum itself.

Figure 1: component diagram of our system.



2. Design Patterns

Both design patterns that we have implemented are part of the Content Server. The code for the *Builder* and *Chain of Responsibility* resides in a separate package, called *architecturePatterns*. This includes the *Chain of Responsibility* handlers and the *Builder* to construct one of these chains.

Some changes were made to the original implementation to accommodate the use of these new patterns. These can be found in the following way (in the Content Server): *services -> ThreadService*, lines 112-122 and 144-151. The implementations themselves are also accompanied by comments, which provide further explanation.

A diagram summarizing the patterns is included below:

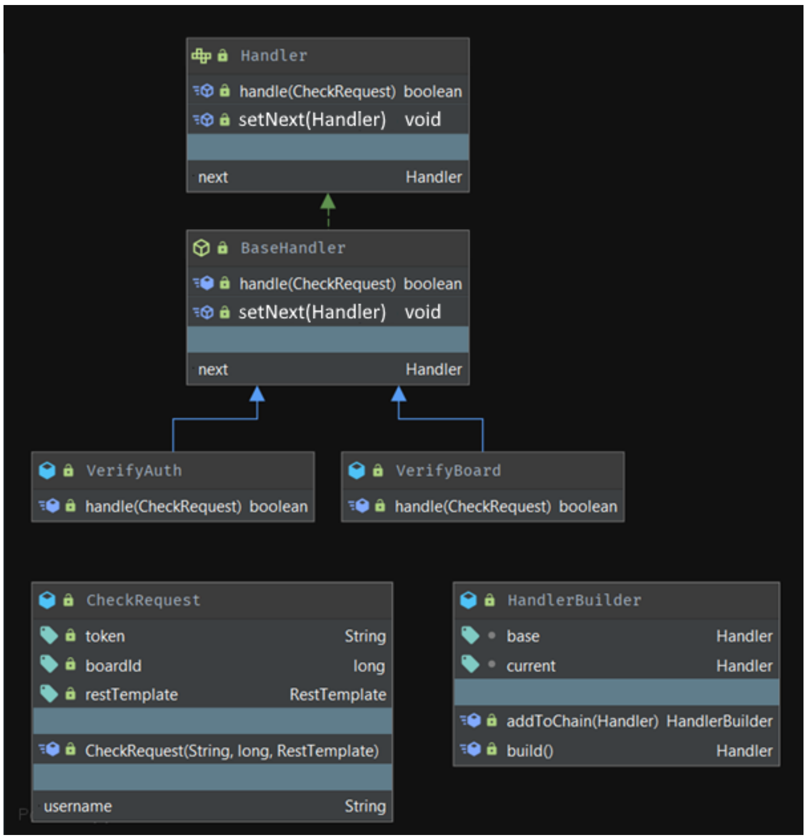


Figure 2: overview of design patterns.

2.1 Chain of Responsibility

The *Chain of Responsibility* is used for creating a thread within a board. Before a user can create or edit a thread, multiple things must be checked consecutively, thus justifying that this choice of design pattern fits this scenario and makes it more scalable. To create a thread, the system first authenticates the user with a token whose generation is the responsibility of the Authentication Microservice. Once the user is authenticated, the existence of the board to which the new thread is associated must be validated, and its status (locked or unlocked) needs to be evaluated: no new threads should be linked to a board that is locked.

We can chain these steps of verification by generating handlers for each requirement. The first handler checks if the user is logged in, and is a valid authenticated user. The second handler is tasked with checking if the corresponding board exists and is not locked. If these two succeed, the creation of a new thread is allowed.

This pattern is implemented using an interface class with a *setNext* and *handle* method. This interface outlines the behaviour of each handler. We also created an abstract class: the *BaseHandler*. This class contains the basic field *nextHandler*, to store the next handler in the chain of responsibility, and the basic logic for the *setNext* and *handle* methods. Concrete handlers extend this class, such as the *VerifyAuth* handler. This handler overwrites the *handle* method to contain the actual logic to authenticate a user.

In the case that *VerifyAuth* decides the request is valid, the result of the basic *handle* method is returned, with a call to *super*; a natural consequence of the *Chain of Responsibility* design pattern is that the request is then either passed on to the next handler, or an adequate boolean value indicating the success or failure of the request is returned, if this handler is the last in the chain. If at any point one of the handlers fails, an exception is thrown, which in turn issues an appropriate course of action, making use of Spring's *ControllerAdvice*. This is also an important part of the chain of responsibility, that is, making sure that an invalid request does not go through.

2.2 Builder

For creating a chain of responsibility, a number of handlers have to be tied together. This could be done manually, but the builder architecture provides a better way to do it, by improving flexibility. When deciding to delegate a task to a chain of responsibility, we do not know the length of the chain beforehand, or the length might need to be extended if future changes in the code require extra layers of tasks. It is hence intuitive to think of these chains as final products of a *Builder* pattern.

We created a *Builder* class with a method designed to chain handlers to the end of the chain. This method returns the builder itself, so it can be stacked. To actually retrieve the fully built chain, we call the *build* method, which returns the final chain of responsibilities. The usefulness of having a *Builder* would become more evident as the application grows and these request handling chains grow more complex and difficult to maintain.