SOFTWARE ENGINEERING METHODS

Assignment 1

Group OP5-SEM10:

Arjun Vilakathara

Stijn Coppens

Adriaan Brands

Gideon Bot

Ethan Keller

Noyan Toksoy

Task 1: Software architecture

For the project a hexagonal microservice oriented architecture was chosen for several reasons which shall be outlined in this report by comparing the chosen architecture to the several other viable options. The microservice architecture was mandatory but a detailed comparison with the other options is useful for developing an understanding and reasoning as to why the chosen architecture is desirable.

First option out there and probably the easiest would be the monolithic architecture. There are several reasons why this architecture is not ideal for the given project. In a monolithic application all business logic is stored in a single location, namely a single server. This server would handle ALL incoming requests to the api and deal with them respectively. This is a disadvantage not only due to increased costs with development as several services intertwine leading to multiple developers working on the same parts of code while doing completely different tasks. Another major issue we would then get into is upkeep. Finding bugs becomes increasingly difficult in big monolithic applications due to the large amount of code at the same location. And perhaps most important doing updates of the server becomes a huge hassle. As updating a small part of the code needs a full, complete new deployment of the last server code there is a very big possibility for bugs to go unfound while pushing to production which then leads to many difficulties as the current running server is broken. This all shows that monolithic systems tend to be small applications. To stay scalable and loosely coupled this clearly isn't a very good fit.

The next viable option is the Service Oriented Application (SOA). This architecture provides more benefits than a monolithic application in the case of the given project, although it too has some disadvantages that need consideration. The major difference between a SOA system and a microservice system is it's scope and this is very visible in the way a SOA system is built. Firstly, an entire communication bus would be needed in order to handle cross service communication. Seeing as how the project's communication between microservices simply needs to consist of simple HTTP messages between the different services instead of complicated and varying types of messages, this would definitely be overkill and an unnecessary amount of work. Another major issue would be security, which stems from SOA systems' nature of being a lot bigger. There is a lot more communication that needs to be properly protected and especially with a bus this can be a major issue. But now on to the positives of a SOA system. The upkeep would be a lot easier to do due to the fragmentation of code in smaller services. This fragmentation also leads to a better scalability as adding new code becomes a lot easier in a new service or in even an existing service due to the smaller scope of the code located there. These are all advantages found in a microservice based system as well if said system is not part of a larger enterprise scale system. If the system were to be integrated into a larger university management service then a SOA architecture would be used for the interservice communication while the smaller services would use microservices. In conclusion, for the level of granularity in the system a SOA would be severe overkill with too many high level risks.

Then finally onto the option that was chosen; microservices. There is a whole heap of advantages that this architecture provides that aren't present in the alternative architectures discussed above. First and foremost the system is modular. By splitting the entire service

into smaller microservices the codebase becomes a lot more maintainable and makes it easier to test, understand and fix bugs. While the SOA architecture also does provide the mentioned advantages, the scale of a microservice is a lot smaller and therefore more ideal for our project. Due to this modularity, there is the advantage of increased scalability, which also happens to be a requirement of the system (microservices are mostly independent thus new ones can be added without having to change any other preexisting code). The architecture also opens up the option to integrate with older legacy systems because of this modularity and while this has not been used yet, this could be a major advantage while implementing the Single Sign On (SSO) from TU Delft in the future. The last big advantage is the option to utilize distributed development. Development teams can be split up between services and barely have to work together until integration testing. This allows for a lot speedier development cycles as there is less bureaucracy to go through due to the limited amount of people working on a service at any time. Having considered the advantages, naturally there are some disadvantages such as security or a larger network overhead as messages need to be sent between microservices thus opening a larger area of attack. These disadvantages however can be worked around. It is also firmly believed that the advantages outweigh the disadvantages in our case.

Some other specifics include the synchronous hexagonal layout due to the easy maintainability of this, the fact that Spring already has a lot of built-in features for this layout which handles issues with and about creating adaptors and an actual HTTP message protocol and allowing us to inject loosely coupled dependencies that only go inwards (thus less overhead). The synchronous communication was chosen because a lot of our communication is needed instantly when requested and a lot of other functionality within the microservices require real time data from the others. There's a few examples where asynchronous communication could have be beneficial (getting all the grades to calculate a mean or standard deviation on, …) but it was felt that there were many more reasons to go for synchronous communication and decided the few specific cases where synchronous would be better didn't warrant the effort needed to implement it. On top of all this we would be required to create communication queues which are severe overkill and very vulnerability prone.

As explained above the architecture chosen for the project was microservices. The idea behind which is to have smaller loosely coupled services/applications as opposed to one monolithic application.

Each service handles one particular job and interacts with each other in case it needs functionality of the other services. Adhering to this principle the microservices in the project were divided/discovered following these steps:

1. From the given requirements produce a set of user stories that represent each functional requirement.
2. From the given user stories produce a set of system operations that represent an external request i.e. possible requests from an application/user that will use the service.
3. To define the microservices of the problem domain, it was first noticed by the group that the domain could be split into a users domain, security domain, grades domain and a coursed domain.

4. Having defined the domains i.e. microservices, the system operations identified in step 2 were split among the domains to the one in which they belonged to.

Having followed the steps above, the following microservices were identified: AuthenticationService, GradeService, UserService and CourseService.The resulting model of the application can be seen in figure 1.1. From the figure it can be seen how each service is separate to each other, having their own database and the only connection to it being an api. It can also be noticed how some microserices are expected to be connecting with each other.

Authentication Service handles everything to do with authentication of a user as a valid user of a given type. Each user in order to use the service must request a bearer token from this service. All other microservices will receive this bearer token as the Authentication header of a request made to them. Each microservice then must validate the token, and to do so it will communicate with the Authentication Service endpoint /validateToken to validate the token, as seen in the diagram. This method of validating a token was a design choice that was made, since in most cases each microservice would have a validateToken() method implemented within itself. However, considering the authentication methodology chosen, it was decided that having each microservice connect to the authentication service to validate the token would be the safest, this is due to the need of a secret key used in producing the token, which the group reasoned would be best to be stored only in one service.

User Service handles the storage and manipulation of user entities. The users are stored in the microservices database. It provides a list of endpoints that allow other microservices and or authenticated users to interact with the users database. Currently in the application only authentication service communicates with userservice in order to authenticate a user as seen in the diagram. Other authenticated users however can insert, remove or view the users in the userservice database depending on their level of access.

Grade Service handles storage, manipulation and calculation of grades and calculations based on the grades. The grades are stored in the grading service database, and a teacher may insert/view/delete/update grades whereas a student may only view their grades. This microservice also handles some functionality on the grades, such as providing a pass rate for a course, a list of courses a student passes and some statistics such as variance and mean of a course. In order to function, it must communicate with the course service to receive information about the weights of each examination in order to calculate a grade for a student, this relationship can be seen in the diagram.

Course Service handles everything to do with a course. Courses, as well as their grading formulas are stored in the grading service database. A user with the required authentication is able to insert/view/delete/update courses and their grading formulas.

Furthermore, to provide another perspective of the overview of the inter-relation between the microservices, figure 1.2 shows the database schema of the application, divided up into microservices. From the figure it can be seen how each microservice has its own database as explained above. The interconnection between the individual tables i.e. how primary keys

of the tables link together show an overview of the interconnection between the data in each microservice. This relationship in the data allows one to more easily understand what each microservice may need from each other, thus resulting in the architecture that was chosen.

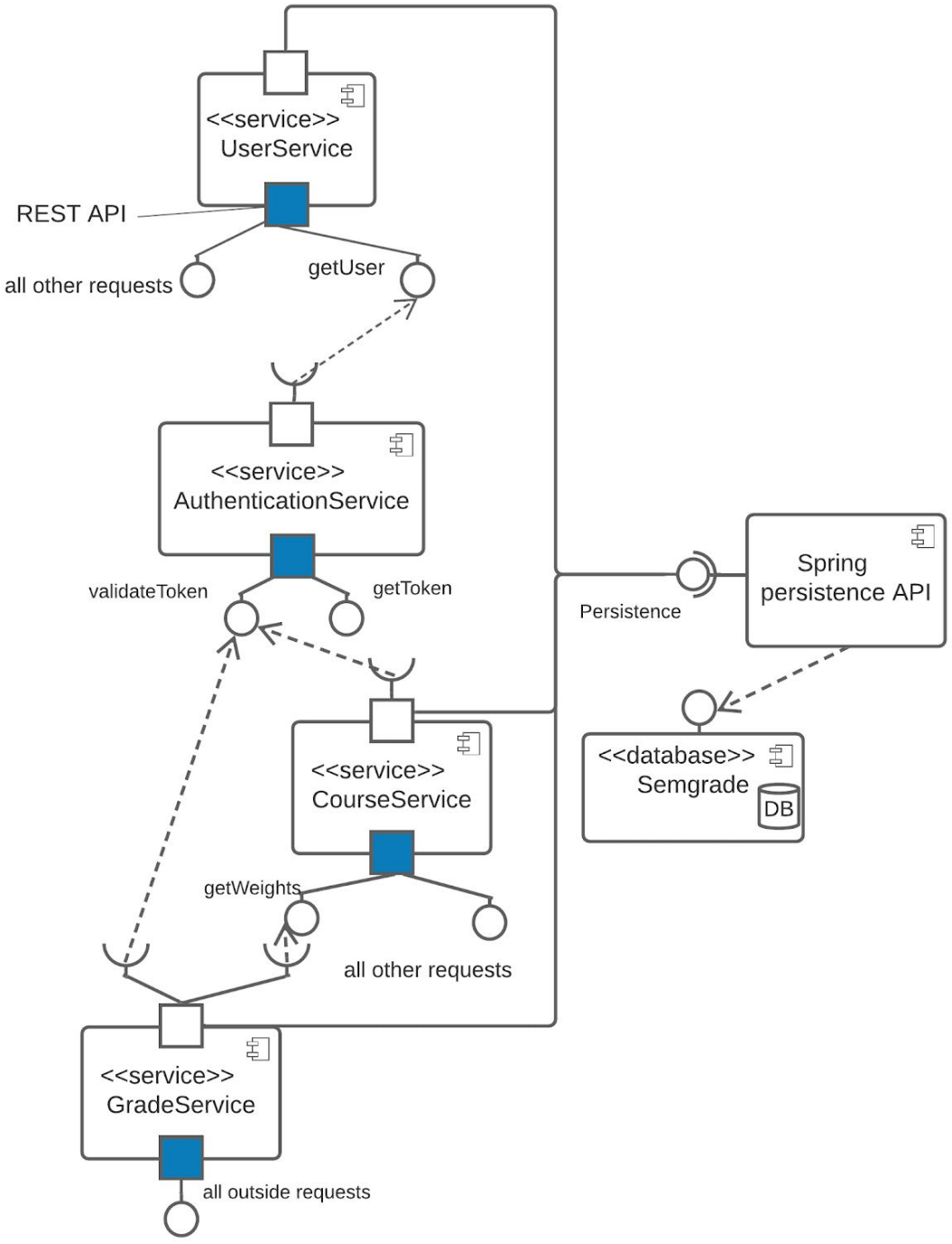


Figure 1.1: System architecture overview

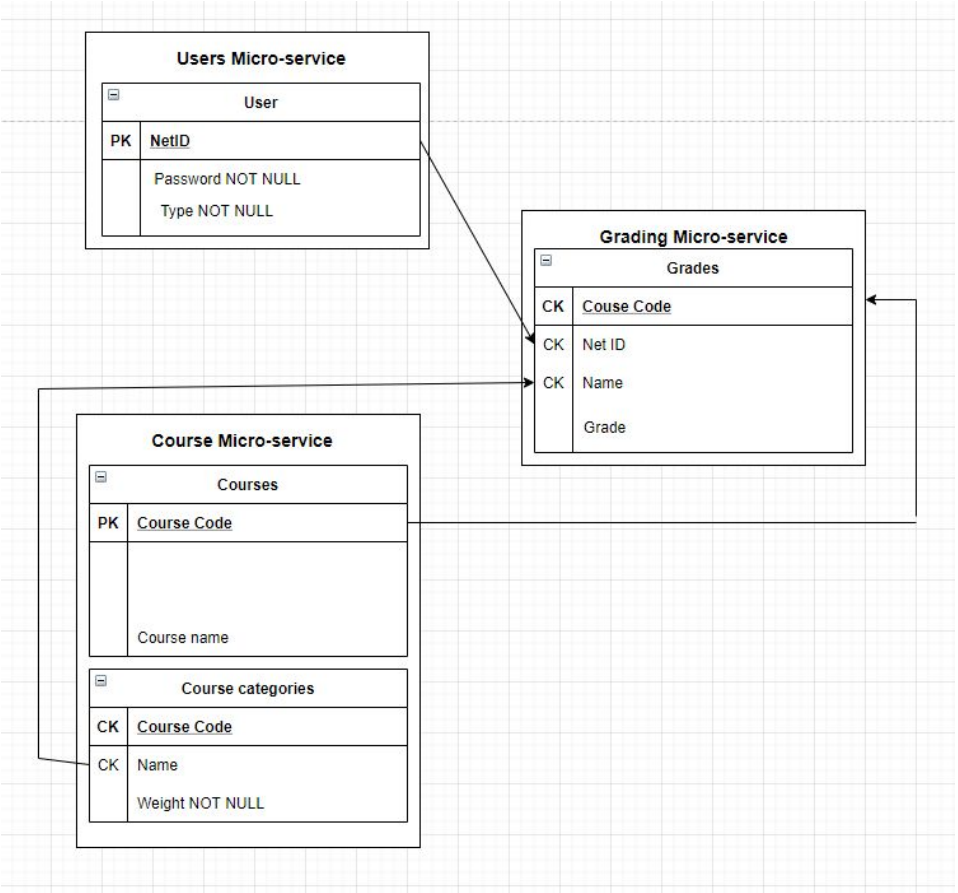


Figure 1.2: Database schema

Task 2: Design patterns

Bridge pattern

1) Pattern description

The course service provides three interfaces, the AuthService[1], CourseService[2] and CategoryService[3]. The classes making use of these (CourseController[4], CategoryController[5]) do not have any knowledge about its implementation, just the methods specified by the interface. This allows for any type implementing these interfaces to be used. This feature is also known as Liskov's substitution principle.

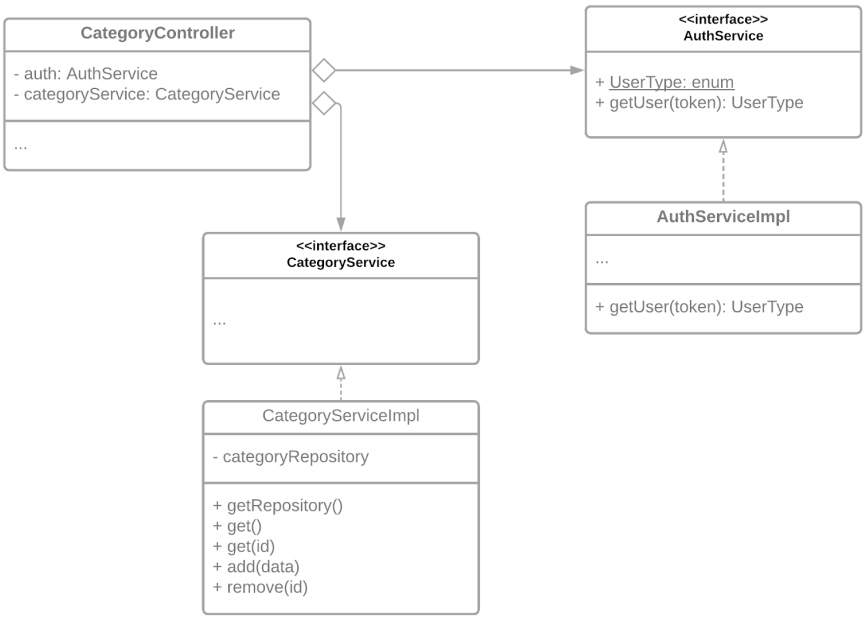
The Spring framework makes sure to create and inject an instance of classes implementing these interfaces when requested by classes consuming the service. The instance used can be defined in the application's properties file[6], which Spring reads in the DataSourceConfig[7] class. Currently, only one implementation for each interface exists, but theoretically more can be provided.

The classes providing these services are available as a bean. Currently, the classes providing the service are AuthServiceImpl[8], CourseServiceImpl[9] and CategoryServiceImpl[10]. Only one implementation per service is provided currently because for the application's functionality no extra implementations are required.

We chose this kind of implementation to help us follow the open-close principle, which in turns helps us follow the project requirements of extensibility and flexibility. We may also want to switch the implementation of the services that use the bridge design pattern dynamically. This is so we can choose a more efficient implementation in order to make the CourseService better for running in parallel, making our software more scalable.

[1] nl.tudelft.sem10.courseservice.application.AuthService
[2] nl.tudelft.sem10.courseservice.application.CourseService
[3] nl.tudelft.sem10.courseservice.application.CategoryService
[4] nl.tudelft.sem10.courseservice.framework.CourseController
[5] nl.tudelft.sem10.courseservice.framework.CategoryController
[6] application.properties
[7] nl.tudelft.sem10.courseservice.application.DataSourceConfig
[8] nl.tudelft.sem10.courseservice.application.AuthServiceImpl
[9] nl.tudelft.sem10.courseservice.application.CourseServiceImpl
[10] nl.tudelft.sem10.courseservice.application.CategoryServiceImpl

2) Class diagram



This works similarly for CourseController and the CourseService and AuthService instances contained in it.

3) Pattern implementation

All used classes are mentioned in the first part of this assignment (and full class paths are referenced in the footnotes).

Facade pattern

1) Pattern description

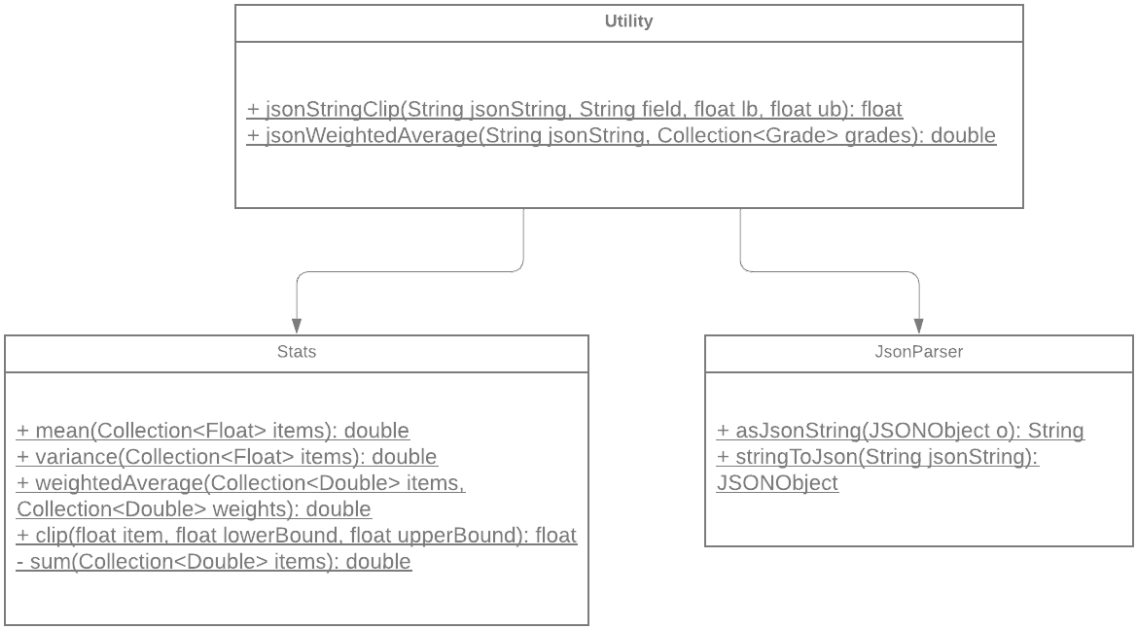
The grading microservice[11] uses a lot of utility functionality like JSON parsing or statistics functions (mean, variance, etc.). These utilities would often be used together and this is why a facade was a very interesting design pattern. It would hide the complex implementations for parsing JSON data and computing statistics. The facade provides an easy-to-use interface for the developers while the more complex computations happen under the hood. Additionally, this facade design pattern completely respects our scalability requirement. If during the development iterations we see more utility functions popping up we can implement those complex functions in their respective classes. Then we just have to tie those to the facade and now the developers can use the same facade. Of course an important downside might be that the facade becomes a ‘God class’. This is a class that holds way too much

[11] nl.tudelft.sem10.gradingservice

functionality and becomes too concentrated to properly work with. A facade must solely provide an easy-to-use interface for the developers to abstract away the more complex implementations that happen underneath. Another pro of the facade design pattern is that developers can keep using the facade interface while the more complex implementation 'behind' the facade can be updated at any time. This allows for easy refactoring. But again it is important to remember that the methods that are available in the facade are to ease the use of the combination of complex underlying methods. The facade should not be a system to delegate methods elsewhere. If that happens, it becomes an antipattern.

In the grading microservice there are a lot of methods that make use of auxiliary utility functionalities. We already identified statistics and JSON parsing but others could still be added in the future like encoding, security, etc.. Because JSON parsing and the statistical calculations were often used close to each other and together we decided to use the façade pattern. We created a package inside the domain package of our hexagonal architecture called 'utilities'[12] where we added these statistics[13] and JSON parsing[14] implementations. Afterwards we created a facade class called 'Utility'[15] which provides simple methods while underneath calling the respective utility classes that do the more complex computations.

2) Class diagram



[12] nl.tudelft.sem10.gradingservice.domain.utilities
[13] nl.tudelft.sem10.gradingservice.domain.utilities.Stats
[14] nl.tudelft.sem10.gradingservice.domain.utilities.JsonParsing
[15] nl.tudelft.sem10.gradingservice.domain.Utility

3) Pattern implementation

All used classes are mentioned in the first part of this assignment (and full class paths are referenced in the footnotes).