Delft University of Technology

Software Engineering Methods
CSE2115

Assignment 1 Group 03B

Authors:
Antonios Barotsis
Nathan Klumpenaar
Dan Sochirca
Rado Todorov
Miloš Ristić
December 17, 2021

Introduction

For the SEM Project we were asked to develop a platform where students can post freelancing services after which companies can hire them to perform these jobs for them. When hired, a contract is formed between the two parties and after the contract expires both parties can rate each other and leave feedback in textual form. In this document we will explain the microservices we implemented, their functionality, but also why we chose to implement them like this and why other choices were not the one we ultimately chose.

Bounded contexts

The most important core domains we identified were the user and offering domain. The user domain is needed for creating and managing the accounts of users that use the platform . The offering domain contains the main functionality of the system, students posting offers and companies offering them jobs. Two additional core domains we identified were the contracting and feedback domain. These are dependent on the offering domain, but big enough to be separated from it. A generic domain we identified is the authentication domain which will provide security to the system.

The core domains have all been mapped into standalone micro-services and are managed by a discovery server that uses Eureka. The authentication domain has been merged with the user service and the Eureka server, the reason why will be explained in a later section.

Microservices

User service and authentication

The user service manages everything that has to do with user creation, modification, deletion and authentication. There are three types of users: Students, companies and admins. Companies can sign up to the system at the user service after which they are stored in the user database. Admins are the ones managing the system and they can perform CRUD operations on other users.

One other important functionality the user service provides is authentication of the user. Users can send their login credentials here and if valid they will be provided a JWT-token. In all future request this JWT-token will be read on the Eureka server and the name and role of the user making the request will be added as headers to the request. This way the micro-service the request is forwarded to doesn't have to make any more requests to other services to try and figure out who is making the request, instead they will have the credentials of the user readily available in the header at all times. Since the JWT token is signed with a secret only the user and Eureka server know no further communication is needed between the two to verify whether the JWT token is valid saving on overhead.

The user service does not depend on any other service and therefore does not contain any communication with other microservices on its own service. There are however other microservices making calls to the user service and all these calls have to do with getting details on a specific user. Mainly they are aimed at identifying whether a user exists.

Offering service

As stated before the offer service provides the main functionality of the system. Here students can post offers for services they provide and companies can request them to work for them using targeted company offers which target the student offers. Companies can also post generic requests in search for employees which students can go through and apply to specifying a price they would do the job for. The first dependency on another microservice can be found here, in the creation of offers. The platform wants to ensure that both students and companies deal with fair partners that deliver on the promises they make. Therefore only users that have not been rated badly, in our case under 2.5 stars on average, can post offers. This average rating of a user is fetched from the feedback micro-service. In case a user is not rated at all the feedback service will return -1, otherwise just the average rating. The offering service will then perform the check of whether this rating is high enough to perform the action and if not the user will be notified that they in fact can not create any more offers.

After posting a non-targeted offer as a company or a regular offer as a student, other users will hopefully

apply to that offer hoping to become a partner of the author of the offer. If the author of the offer is happy with such an application they can accept it, after which automatically a contract is generated between the two parties. Here is where we can find the second communication the offering microservice makes. This contract is not created on the offer service but rather a call is made to the contracting micro-service which returns the generated contract if successful. On the contracting microservice there are also validation checks in place and if the call is not succesful a bad request will be returned to the offering microservice, which in turn will return a bad request to the user that originally made the call with the error message included in the body.

A third communication form can be found when searching for student offers of a specific user. The microservice will first make a call to the user service to validate that the user actually exists. If they do not exist then a 404 NOT FOUND is returned to the original user making the request.

Contract service

The contracting microservice is responsible for maintaining both active and inactive contracts of users of the platform. Contracts have a startdate and endingdate, two partners and specifics on how many hours the student works and what their monetary compensation is. Contracts can be changed if both parties agree on what needs to change. If a user wants to change something they can propose this to the other user. The other user can then accept, but only if the proposal is valid. One example of an invalid proposal would be if a user wanted to extend the contract to an enddate that is more than 6 months after the end date. In this case the service would return a bad request.

As mentioned above the contracting service will be called whenever a student offer or non targeted company offer is accepted. For added security the endpoint where contracts are created can only be called internally by another microservice, users on the outside are not able to access this. This is done by adding an authorization header to the call made from one service to another. Someone one the outside could also put this header to be equal and try to penetrate the system this way but the same header is used for the JWT-token and since all requests go through the Eureka server the header will be overwritten.

As was with the proposals, contracts have certain conditions that they have to meet i.e. two parties can not have more than one active contract at a time, students can not work more than 20 hours per week etc. If one of the conditions is not met the service will return a 400 BAD REQUEST response to the offering service which in turn will send this back to the user making the request. Included is also the reason for failing so that the user can change and resubmit their request if needed.

Feedback service

The feedback service is meant for quality control on the users of the platform. Here users can leave feedback on a partner they worked with in the past. Aside from a numerical rating between 0 and 5 they can leave feedback in the form of text, which described in more depth what they think of a user. Any future users wanting to work with a user will then be able to look up what previous users thought of the person and can then be further encouraged, or discouraged depending on what they find.

As mentioned, feedback is only for partners you worked with in the past, and therefore feedback can only be left on contracts that are no longer active. For this, the feedback micro-service has to request a contract on the contracting micro-service. After requesting the contract is sent and then a check on the feedback microservice will happen to verify the contract is indeed no longer active.

Eureka server

Lastly a brief explanation of the Eureka server. All calls will initially be made to here after which they are forwarded to the corresponding microservice. As mentioned before the Eureka server is also responsible for decoding the JWT tokens. It will append two headers to the request and if the JWT token is valid these will have the name and role of the user making the request, otherwise they will just be empty

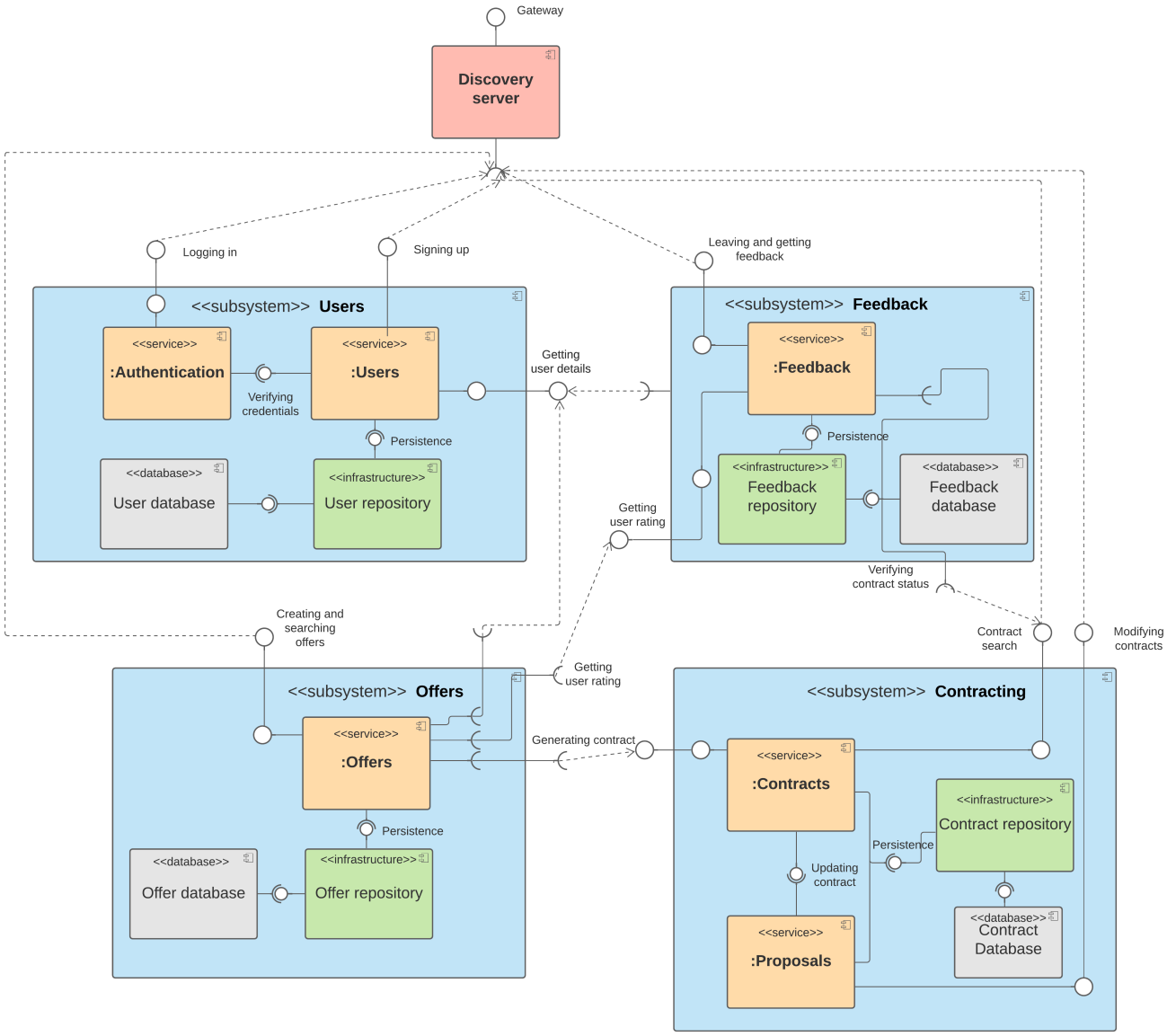

Figure 1: Component diagram

Architecture explanation

Now that we have explained what microservices the architecture of our platform is composed of we will explain why we did not opt for other designs.

At first the authentication domain was also mapped into a micro-service and the idea was that every microservice would authenticate the user by requesting authentication of every request at the authentication service. We realized however that since the authentication server was heavily dependent on the user service to get user details, and since all requests go through the discovery server, a better solution would be to distribute the authentication functionality over the user and Eureka server. As explained before, the user service will do the initial authentication and the Eureka server is concerned with decoding the JWT token of every other request that user makes after this initial request.

Another change one could think of is to merge the offering microservice with the contracting microservice. The latter can be seen as an extension of the first one as some might think it is only a confirmation of the offers accepted. However the contracting microservice provides more functionality as users can also request changes to existing contracts they have. This turns out to be quite complicated and has nothing to do with posting or viewing active offers. The way we see it, contracting is just a new stage after the offering stage and since there

is a lot of logic on both services that do not concern the other service we felt it was good to split the two.

Something else that you could consider is to remove the feedback microservice and add the functionality elsewhere. We believe however that there is not really any other microservice where the feedback has a proper place. It does not have much to do with active offers other than the fact that posting offers depends on having good feedback. The contracting service has enough functionality which does not relate to the feedback at all. We see feedback as another stage after contracting and therefore this combination makes no sense either.

Maybe the most logical place would be to put it in the user service. After all, the whole point is for a user to rate another user, the only outside dependency is the contracting service. This combination however is undesirable as well as the user service is such a vital component in the entire architecture whereas the feedbacking service is a rather minor feature. If we want to add some minor change to the feedback service this would take a whole new deployment of the entire user microservice too which adds a major unwanted risk. Therefore it is better to split the two up as we do not want to risk losing our most important microservice.

Another big change you could think of is to split the offering microservice into microservices for every type of offer. We felt however that this would not make a lot of sense for multiple reasons. First of all, the different offers share a lot of fields which hints more to the use of inheritance to avoid code duplication. Also they depend on each other, the TargetedCompanyOffer targets the StudentOffer. This would add unnecessary overhead when creating the offers on different services. Also the actions that can be performed on these offers are very similar. Since they are all similar and heavily tied together we felt it made more sense to put them all into one microservice.

The user and offering microservice are vital components of the system and very different from each other, therefore it makes no sense to put these two anywhere else. We believe that with the changes above we have mentioned all the most logical changes one could think of when designing our system, and we hope we have explained enough to you why we did not opt for these changes.

Design patterns

Façade Design Pattern

Initially, our application had a lot of functionalities which the client had to access using different paths. For instance, when a company wants to accept an application and a student wants to accept a request, they would have to access different endpoints. Our team decided that this is not desirable, because this makes it complicated for the clients to use. This is how we ended up implementing the Façade Design Pattern.

Our façade implementation is a layer on top of the currently existing controllers which takes requests from all clients and based on their identity decides which of the already existing methods to redirect the request to. By doing this we achieve the core functionality of the façade design pattern - creating a higher-level interface that makes the subsystem easier to use. Clients do not have to even know that our service treats users differently based on their role.

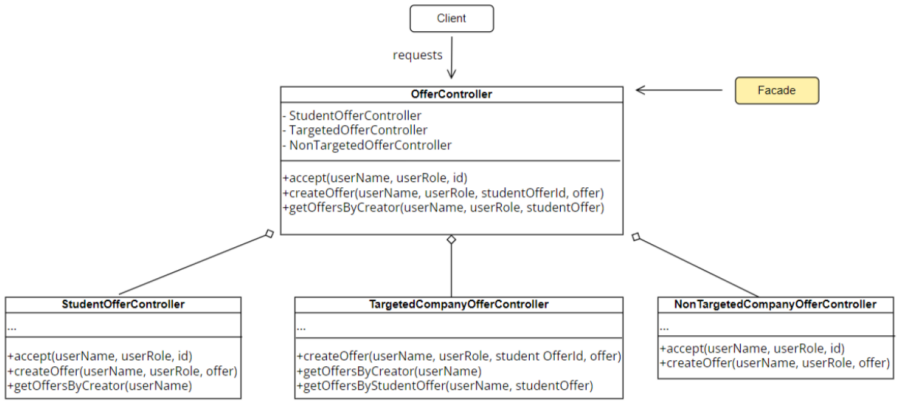

Figure 2: Class diagram of this Design Pattern

Factory Design Pattern

Our application has a few different user types; Student, Company and Admin. In order to promote loose coupling, make it easy to add new User classes in the future and to abstract the process of instantiating these classes we used the Factory Method pattern.

We created an abstract User class and an abstract UserFactory with a createUser method. We then created classes for each one of the three user types that extend User and UserFactory.

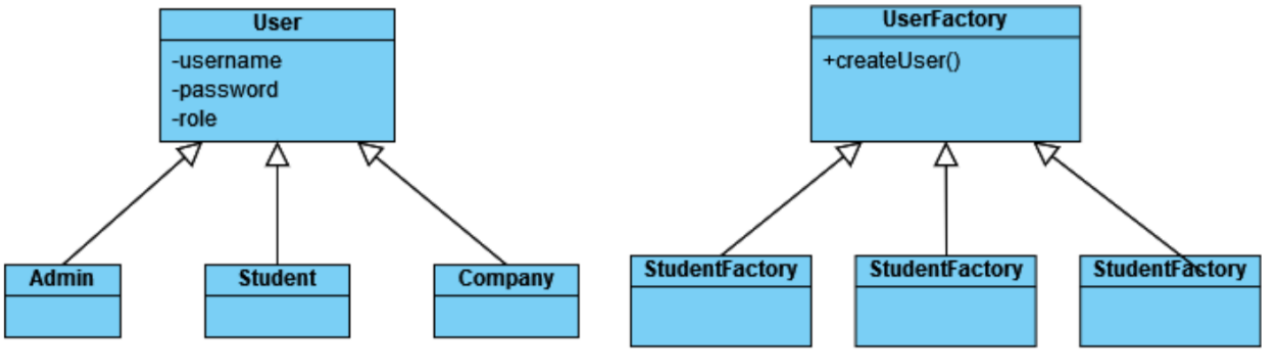

Figure 3: Class diagram of this Design Pattern