Assignment 1 (Draft)

SEM Group 01A

December 14, 2021

1 Task 1 - Software Architecture

The software design approach chosen for this project is Domain Driven Design. Its main focus is modelling the software according to the rules and business procedures of the underlying domain. This requires a meticulous analysis and understanding of the underlying domain with the guidance of domain experts.

1.1 Context Maps

As per the Domain Driven Design, we first commenced by identifying the bounded contexts in the scenario:

- **Users**

  The users are a central element of the system. They are the main stakeholders, as they will be making direct use of the software solution in a regular fashion. We have identified 2 types of users in the given scenario: students and companies (more specifically delegated employees of the companies' HR departments).

- **Service Posts**

  Service posts are the integral connecting platform between users, namely between students and companies. They allow students to advertise their skill-set and availability to companies, who in turn can respond to the service posts. Moreover, they also provide the reverse. They allow companies to post generic service requests to which students can respond to.

- **Contract (management)**

  Contracts are generated by the system once a service/generic request is accepted by both parties. Students and companies (upon mutual agreement) can extend, modify (e.g., adding more hours), or terminate existing contracts

- **Authentication**

  Authentication is a crucial part of the system. It allows students and companies to access their respective service posts and contracts within the system as well as differentiating students from companies when interacting with the system.

- **Feedback**

  Students and companies can provide feedback to each other at the end of a service period in order to better inform both parties when it comes to accepting a contract. The feedback consists of a textual feedback form and a rating.

2 Microservices

2.1 User

We have decided to include anything related to the user under one single microservice. It will manage all the users in the system, and also store all their login information and other user data, like name, NetID, role, feedback and their respective rating. The other microservices will be able to communicate with this microservice to get the user information that they need in order to function. Parts of the authentication will also go through User, where it will have a short route to get to the information required to check the incoming request, in an attempt to reduce potential delay.

By storing all users under one microservice, companies and students will be stored together. To be able to separate them, we have added a role field. We decided to make this field a String, instead of a Boolean, to be able to more easily expand the available roles in the future. All users are also grouped together as students and companies communicate between each other often. To reduce the risk of significant delay, we found it a better solution to group them rather than splitting them into separate microservices.

The User microservice stores the feedback and rating of each user. Originally we planned feedback as a separate microservice, but then decided to integrate it into User for several reasons. The most important reason was that every time you would fetch a user, you also needed to fetch the feedback connected to that user which is stored in a different service. This would lead to unnecessarily high delay to fetch a user, and becomes a bigger issue the more users you fetch at once. Feedback also did not contain any business logic, and therefore did not serve any purpose outside of working as a feedback data storage.

2.2 Generic Service Post

We split the posting into two separate services to reduce the load, as almost all communication goes through the posts. They are a central part of our application, and it is therefore also important they do not get overloaded or have significant delay.

The Generic Service Post microservice is specifically a microservice for companies posting job offers to multiple students, and only students can see these offers. Multiple students can answer to a job posted by a company, but the company can only accept one applicant. Therefore companies can decide which student to accept, based on the students data - expertise, hours per week and duration. Once a offer is accepted by a student, the information is send to the Contract microservice. This way, the company and student can further communicate and agree on the specifications of the job.

2.3 Student Service Post

The Student Service Post microservice is specifically for students posting their preferences and competence, and only companies can see these postings. The postings will contain information pulled from User about the student which includes their rating, and also the student's expertise, price, availability and competence. When a company finds a suitable student, they are able to send that specific student an offer with further details. The student will be able to see the offer once it is sent, and will also be able to negotiate changes to this offer. When both parties agree on an offer, the required information will be sent to the Contract microservice to write an official contract between the two parties.

2.4 Contract

The Contract microservice generates and manages contracts between students and companies. We chose to design Contract as a standalone microservice instead of being integrated within the Service Post microservices as it requires a non-trivial amount of business logic, such as but not limited to generating and modifying contracts, which, if incorporated within the respective Service Post microservices, would over-complicate them, add additional load (affecting scalability) and result in low cohesion microservices.

Whilst it does involve a higher degree of network communication between microservices as the Service Post microservices must call endpoints from the Contract microservice to generate/modify a contract as opposed to doing it directly within the respective microservices, since the Contract microservice is not called very often it does not pose a threat of creating a bottleneck.

In terms of load balance, the Contract microservice has to manage the contracts between students and companies. Large amounts of resources are not required to complete this action. Once companies find the students who best fit the job, they usually do not use this microservice during the contract period.

2.5 Authentication

To offer a better experience for users, we had to think about a security measure that assures the integrity of their data. We did not want to reinvent the wheel. Therefore, we chose to use spring security to augment the authentication process. Since spring is known to offer professional protection, we do not have to worry about the complexity of validating whether a user request tries to exploit our system.

The authentication microservice is essential for giving companies and users confidence to use our services. It is one of the simplest microservice in our architecture in terms of load balance. Replication and sharding of this microservice are not needed to be employed that much. As a user, you only interact with the authentication microservice once when you log in. Unlike other microservices in our architecture, it is harder to get the authentication microservice overloaded.

3 Micro-service Interactions

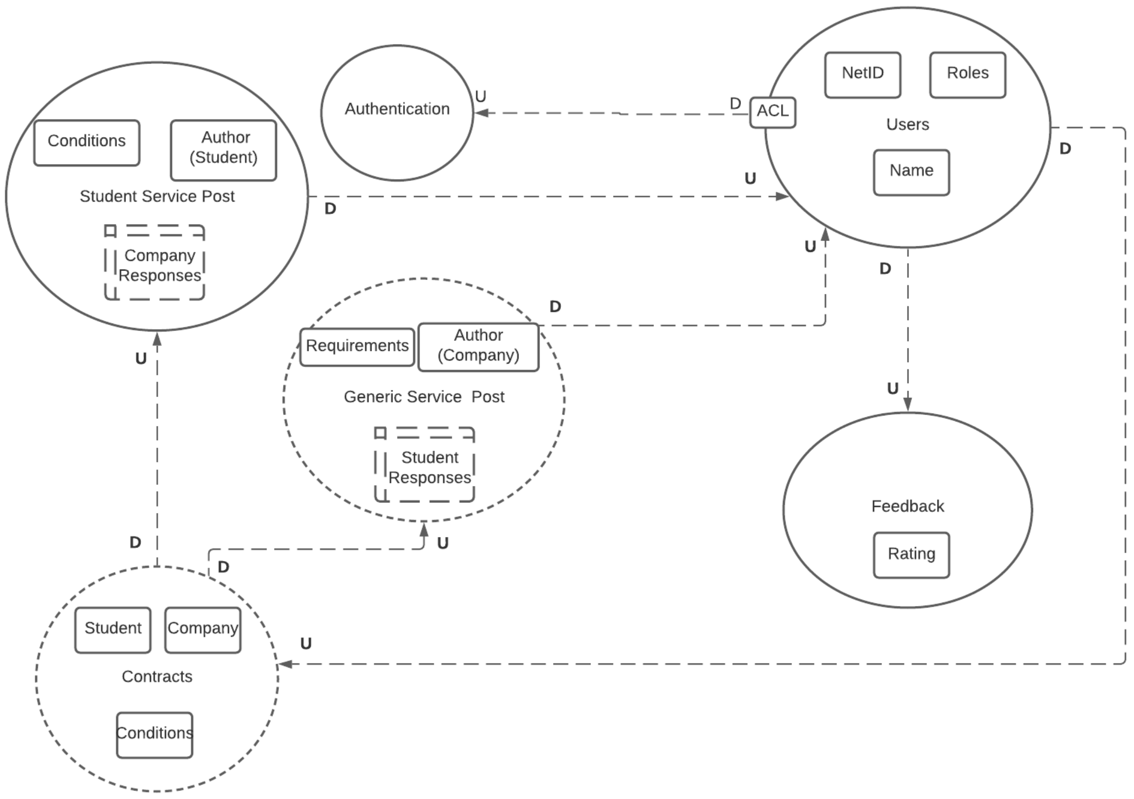


Figure 1: Context Map

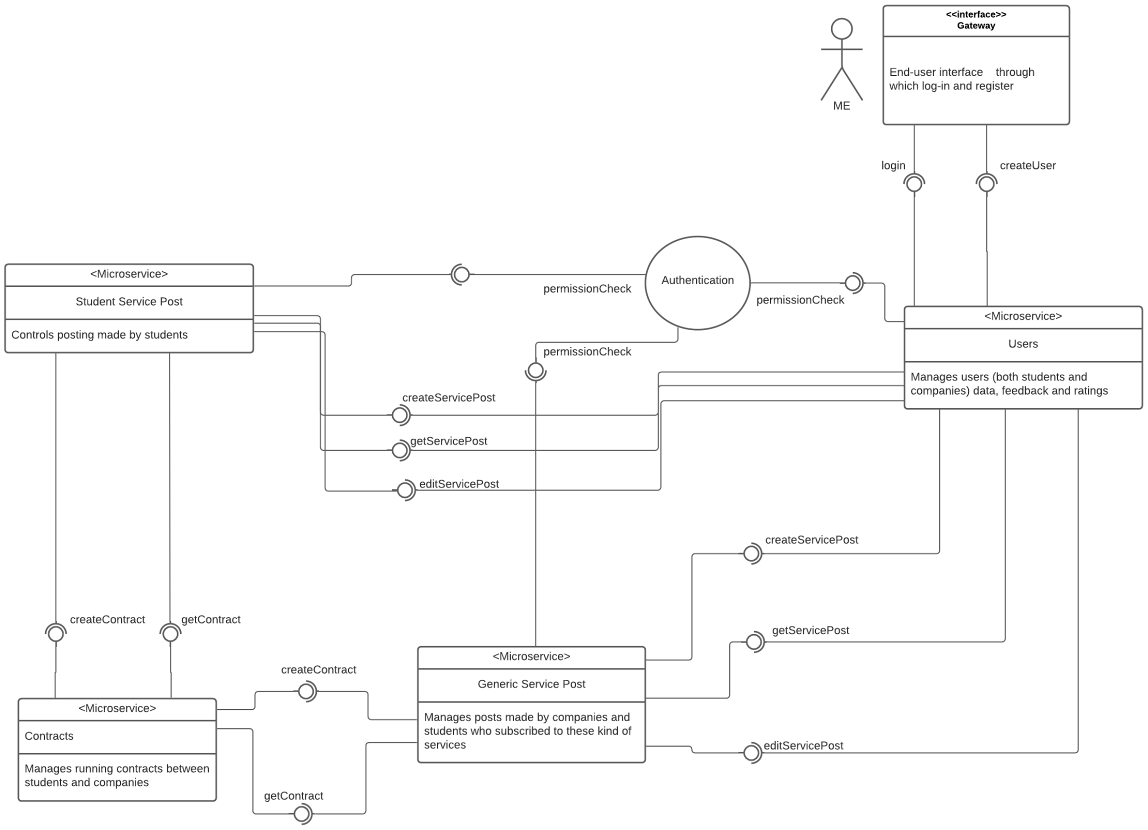
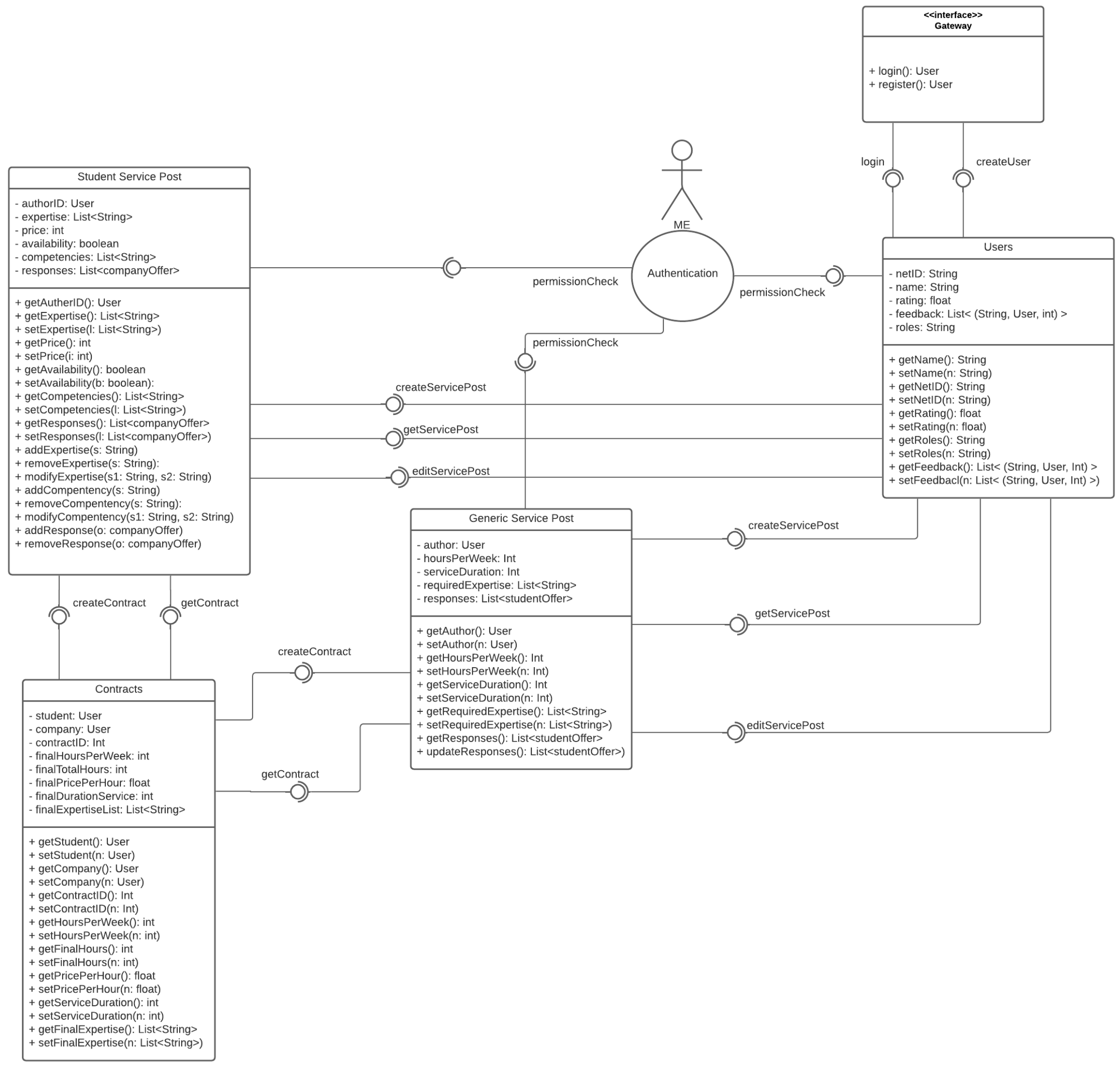


Figure 2: Component diagram of system Architecture



<<interface>>
Gateway
+ login(): User
+ register(): User
login
createUser
ME
Authentication
permissionCheck
permissionCheck
permissionCheck
Student Service Post
- authorID: User
- expertise: List<String>
- price: int
- availability: boolean
- competencies: List<String>
- responses: List<companyOffer>
+ getAutherID(): User
+ getExpertise(): List<String>
+ setExpertise(l: List<String>)
+ getPrice(): int
+ setPrice(i: int)
+ getAvailability(): boolean
+ setAvailability(b: boolean):
+ getCompetencies(): List<String>
+ setCompetencies(l: List<String>)
+ getResponses(): List<companyOffer>
+ setResponses(l: List<companyOffer>)
+ addExpertise(s: String)
+ removeExpertise(s: String):
+ modifyExpertise(s1: String, s2: String)
+ addCompentency(s: String)
+ removeCompentency(s: String):
+ modifyCompentency(s1: String, s2: String)
+ addResponse(o: companyOffer)
+ removeResponse(o: companyOffer)
Users
- netID: String
- name: String
- rating: float
- feedback: List< (String, User, int) >
- roles: String
+ getName(): String
+ setName(n: String)
+ getNetID(): String
+ setNetID(n: String)
+ getRating(): float
+ setRating(n: float)
+ getRoles(): String
+ setRoles(n: String)
+ getFeedback(): List< (String, User, Int) >
+ setFeedbacl(n: List< (String, User, Int) >)
createServicePost
getServicePost
editServicePost
Generic Service Post
- author: User
- hoursPerWeek: Int
- serviceDuration: Int
- requiredExpertise: List<String>
- responses: List<studentOffer>
+ getAuthor(): User
+ setAuthor(n: User)
+ getHoursPerWeek(): Int
+ setHoursPerWeek(n: Int)
+ getServiceDuration(): Int
+ setServiceDuration(n: Int)
+ getRequiredExpertise(): List<String>
+ setRequiredExpertise(n: List<String>)
+ getResponses(): List<studentOffer>
+ updateResponses(): List<studentOffer>)
createServicePost
getServicePost
editServicePost
createContract
getContract
createContract
getContract
Contracts
- student: User
- company: User
- contractID: Int
- finalHoursPerWeek: int
- finalTotalHours: int
- finalPricePerHour: float
- finalDurationService: int
- finalExpertiseList: List<String>
+ getStudent(): User
+ setStudent(n: User)
+ getCompany(): User
+ setCompany(n: User)
+ getContractID(): Int
+ setContractID(n: Int)
+ getHoursPerWeek(): int
+ setHoursPerWeek(n: int)
+ getFinalHours(): int
+ setFinalHours(n: int)
+ getPricePerHour(): float
+ setPricePerHour(n: float)
+ getServiceDuration(): int
+ setServiceDuration(n: int)
+ getFinalExpertise(): List<String>
+ setFinalExpertise(n: List<String>)

Figure 3: UML-class diagram of system Architecture

Micro-service interactions are best described by figure 2. Here we can identify these interactions by the lollipop and socket notation. E.g. the 'getContract' method is used by Generic Service post to get a contract from the Contract micro-service. Figure 5 shows each component in more detail.

4 Task 2: Design Patterns

4.1 Factory Pattern

We used the Factory design pattern to generate different types of users. To create a new user the functionality of the UserFactory class is employed. Besides the attributes of the object we want to create, we need to specify the role. The role determines the type of user we want to instantiate. The Factory design pattern is the most intuitive solution for our problem. We wanted to separate students and companies. However, we also needed to allow students to send feedback to companies and vice-versa. This design pattern allowed us to use a shared database and consequently shared controllers and services.

To further support our choice, we show one big advantage. Our implementation is easily and flawlessly scalable. Our classes extend the superclass User for which we have a repository, a controller and a service. That is all we need to persist data. A naive implementation would have proposed the creation of separate services, controllers and repositories for each class. One can see that the naive implementation makes it difficult for a student to send feedback to every created instance. As an example, consider ten different kinds of users. A student service would need to auto-wire ten repositories, excluding the feedback repository, so that they can send feedback to any instances. On the other hand, our implementation has to instantiate only one repository.

4.2 Proxy Pattern

Another design pattern that we decided to implement in our product is the Proxy Design Pattern. As we were planning the design of the Student Post Service, we came to the realization that we needed a way to interface with the User object, which is stored in a different microservice. The Remote Proxy design pattern is a perfect match for this, as it allows us to represent the remote object in a local object which contains the same information as the remote object. This allows us to fetch the remote object only when it is needed. As a result, we prevent having to store redundant information in 2 separate microservices, which can quickly lead to data inconsistency.

We implemented this by creating a UserProxy class, which is an attribute of a Student Post and Company Offers that acts as a local in-memory copy of the User. When the User attribute is required, such as when all information about a certain Post must be fetched, the UserProxy makes an API request to the User microservice using the authorId, instantiating a new object with the returned data. As such, the intricacies of fetching the Users is abstracted away from the Student Post class.

4.2.1 Class diagram

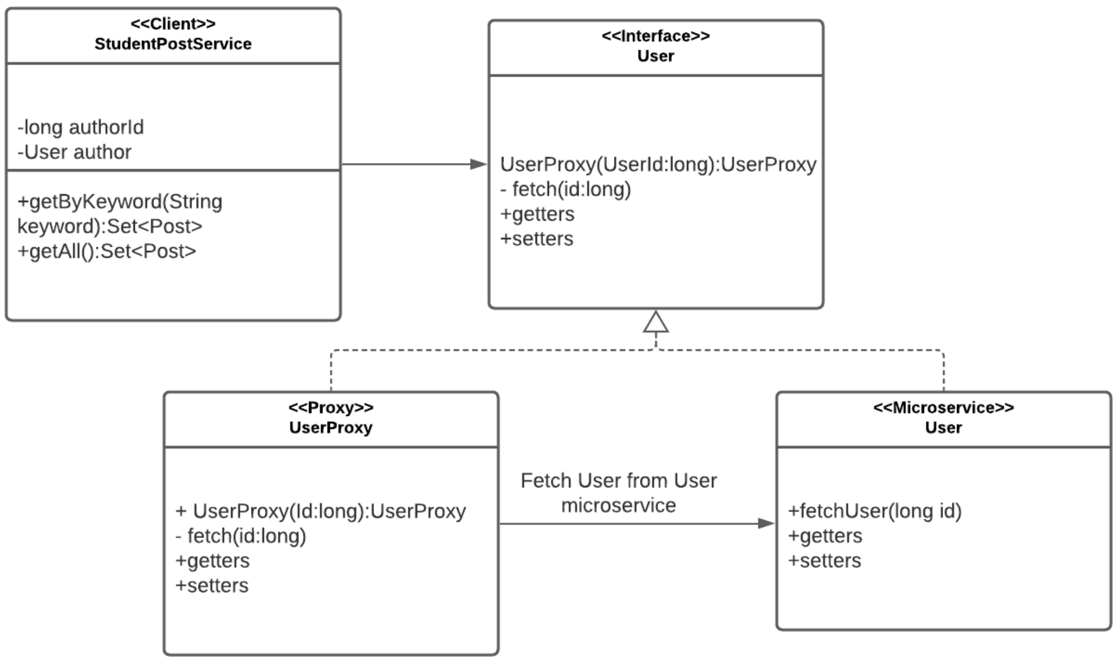


Figure 4: UML-class diagram of Proxy Design implemented in Student Post microservice

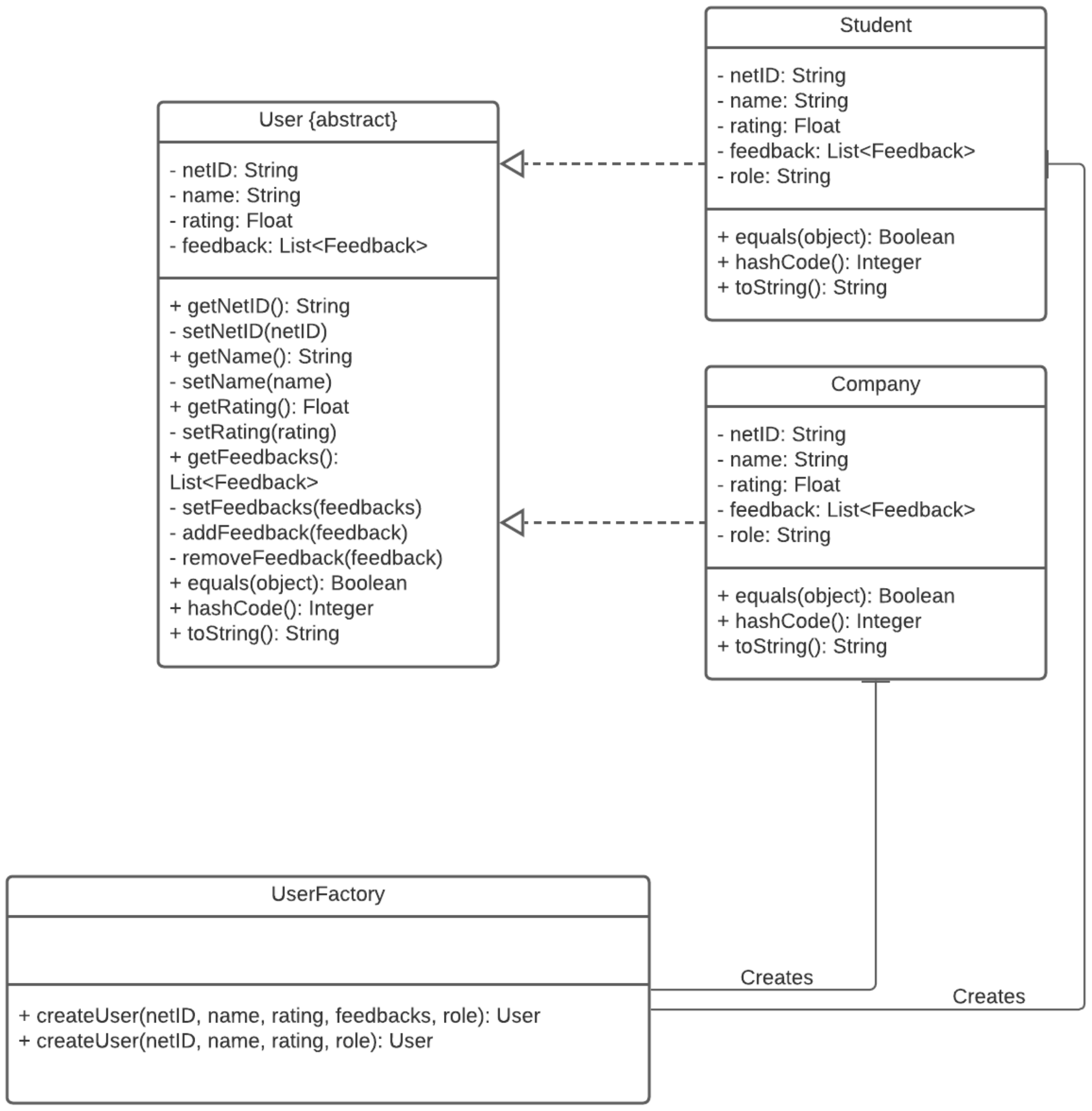


Figure 5: UML-class diagram of Factory Design pattern implemented in Users microservice