# Assignment 1

## Bounded Contexts

We have identified the following bounded contexts:

**Core Domains:**
- Student
- Company
- Job Request
- Contract

**Generic Domains:**
- Gateway
- Security

**Supporting Domains:**
- Feedback

Since bounded contexts are sub-concepts within the whole domain, the ones above are sub-concepts of our **Student Freelancing** domain. We decided to make each one of these bounded contexts a microservice by using the **decomposition by subdomain** strategy since we are using **Domain-Driven Design (DDD)**. This allows for easier maintenance, scalability and reliability of data in the future. For example, if we have an issue with the contract generation, we can be sure that it's probably an issue in the contract microservice and nowhere else. Furthermore, if we have an increase in the number of students in our app but not companies, we can just scale our student microservice independently, allowing for a more efficient growth of the system.

## Microservices

The architecture of our project is described below. We describe the reasoning behind our choices per microservice. After these explanations, we move on to the design patterns.

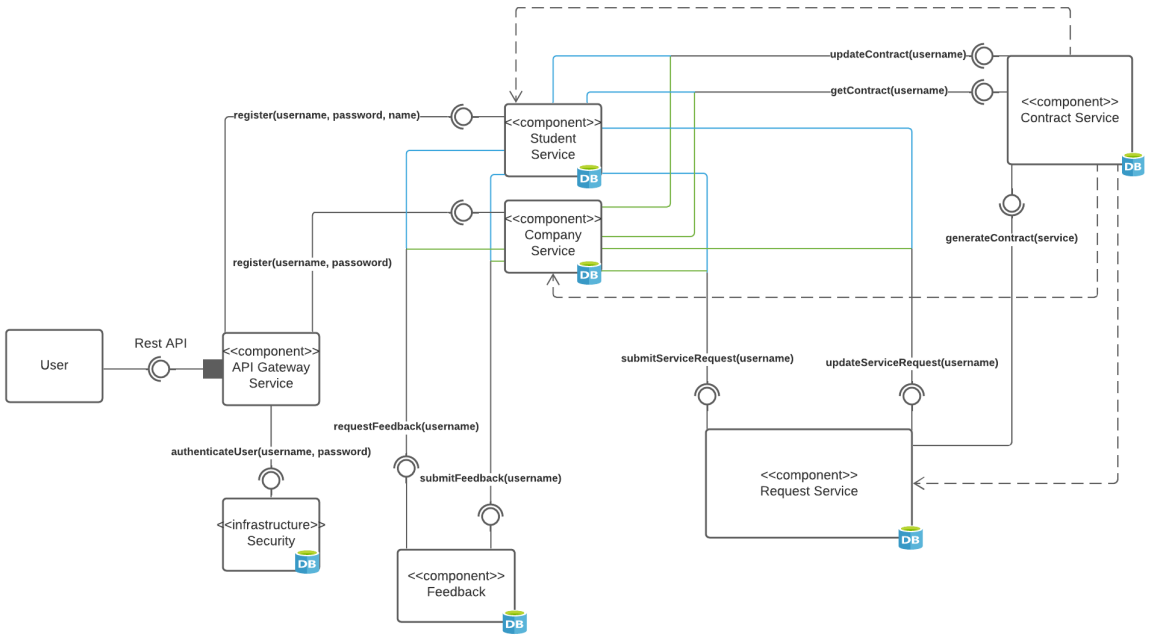



*Figure 1: UML Component Diagram of our Student Freelancing System*

## Contract Microservice

The responsibility of the Contract Microservice is to make sure that a contract is correctly generated once both the student and the company agree on business terms. The reason why we decided to create a separate Contract Microservice was because it would be more convenient for the generation of the Contract. As soon as both parties agree to the terms via the Contract microservice, the Contract Microservice will generate the Contract and store it in its database, allowing both the Student and the Company to access it via an API call. Furthermore, the contract would be substantially more consistent as we would have a database to rely on the correct form of the contract. Moreover, modifying contract terms becomes easier as the company can modify a term of the contract, forward that to the microservice, then the microservice forwards it to student, student accepts and the contract is stored in the Contract Microservice database.

## Main Gateway

The purpose of this service is to load balance requests and route them to different microservice. It can direct requests to 3 microservices - the authentication, student, and company ones. However, in order for a request to be forwarded to the student and company microservices a valid jwt token should be provided in the authorization header. This is checked with a predicate that sends a validation request with the token to the authentication service. If the token is valid **and** that user has the correct role (either student or company) the request is forwarded to one of the 2

microservices. Moreover, if the token is valid before the request is forwarded, its path is rewritten to also include the username of the user.

## Feedback Microservice

The responsibility of this microservice is to keep in a database all of the reviews that students and companies have received. Furthermore, a student or a company can request to see the reviews of a certain company or student, respectively, and also post new feedback for them. Since we wanted our microservices to be as simple as possible and the feedback required their own database we decided that it was best if we had a dedicated microservice for them (even though it only has 2 endpoints) plus it is required by two other microservices (the student one and the company one) so having it in both of them would just mean we would have more code duplication.

## Authentication Microservice

The responsibility of this microservice is to allow users to login or register using their username and password and to validate jwt tokens. When a user sends their credentials, the system will check whether they are valid (they are in the database) and if they are it will generate a signed jwt token (using a random 64 byte string as the secret key) which stores the username, role of the user (student or company), and the duration of the token which is then returned to the user. If the credentials are invalid a response with status 401 is returned. The second endpoint that the microservice provides is registration- the user again needs to send their credentials and role and the system will check whether the username is already taken (in which case it will return a response with status 400) and also whether the role is a valid one. If the username is free, the new user will be saved to the database (the password is also going to be encrypted) and a newly generated jwt token will be returned. The final endpoint provides validation for jwt tokens by decoding them and checking whether they are expired or the username is invalid. This endpoint is used only by the gateway before forwarding an incoming request to the other microservices. The reason this microservice was split from the main gateway is because the spring cloud security library used was incompatible with the spring cloud gateway one.

## Student Microservice

The responsibility of the student microservice is to handle all actions surrounding a big user base within the system: all the students. It handles storing their preferences in a database, and allows for interaction with companies. Since students are a big part of the user base, we decided it would be best to have this entity inside a separate microservice, since a student would have a lot of features next to a lot of communication. Although the student and company microservice share few similar pieces of functionality, they are way too distinct to warrant just one single

microservice, or to use an inheritance structure; especially with the distinction be made between a student's service request and a company's service request. This together made it so that we chose to keep the student separate altogether. One could create a new student and store it, but also get a list of all the students, or get a specific student. The student itself can communicate with other microservices, such as the request microservice, where the student could see a list of all the job offers companies have made. Another feature that the student microservice has is that it can post a job service. This job service could be seen by companies, who could then reply to that job service.

## Request Microservice

The responsibility of the Request Microservice is to keep track of all the requests and communication between students and companies. This communication is used to inform both parties (students and companies) of their requests regarding wage, hours needed for a specific job, requirements (from the company side) and also expertise (from the student side). In order to have all these fields together, the idea of creating a separate Request Microservice was good. Each job request made by a specific company can be easily found by the students who are looking for jobs for that specific company. In reply, the students can send requests with their expertise and their requirements regarding wage and the amount of hours, in order to apply for that specific job. Both parties can modify their specific requests, helping them to negotiate to agreed terms regarding the job before signing the contract. Since there are types of requests, there are also two types of entities (one for Company Requests and one for Student Requests) which are stored in the database. A company can easily accept a student's request by sending a post request to the Request Microservice, which sets the companyId field of the Student Request to be the company's ID. Afterwards, if the student wants to work with that company, they can also accept their own request, setting the acceptedByStudent field in the student request to be true, generating a contract. The exact same procedure occurs with the Student accepting a company request. On the other hand, if the student wants to reject the company, they can send a post request to reject the company and the companyId will be set to an empty string and the boolean isAcceptedByStudent will be set to false. A similar procedure is done when the company wants to reject a student, just instead of setting the companyId to an empty string, the studentId is set to an empty string and the field isAcceptedByCompany of the CompanyRequest class is set to false.

## Company Microservice

The responsibility of the company microservice is to handle all actions that concern the companies. Once registered a company is able to post job requests, but also search for services provided by students. As far as communication goes between

other microservices, a company can search for specific students based on their expertise or provided services, this goes via the request microservice. When a company wants to create a job request, it will do so via the request microservice. Once a request has been accepted the contract microservice will generate and send a contract and when both student and company accept, the contract starts. When the contract has ended the company can submit feedback about the student, this goes via the feedback microservice. The students and company are both users of the system. We choose to separate them both into microservices because of the differences in functionality. This separation also makes the software more modular and thus easier to maintain in the future.

# Design Patterns

## Bridge

The Bridge Pattern was used in the Request microservice since it decouples an abstraction from its implementation so that the two can vary independently. In the Request Microservice, we have two types of requests: Student Request (helping students to communicate with companies by sending the expertise and the demands regarding the salary and the workload) and Company Request (helping companies to announce their jobs opened for students).

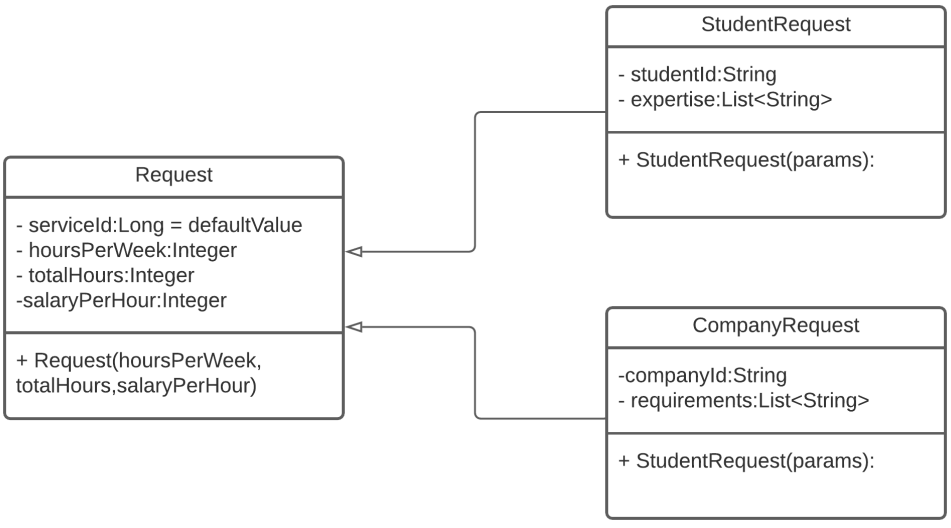


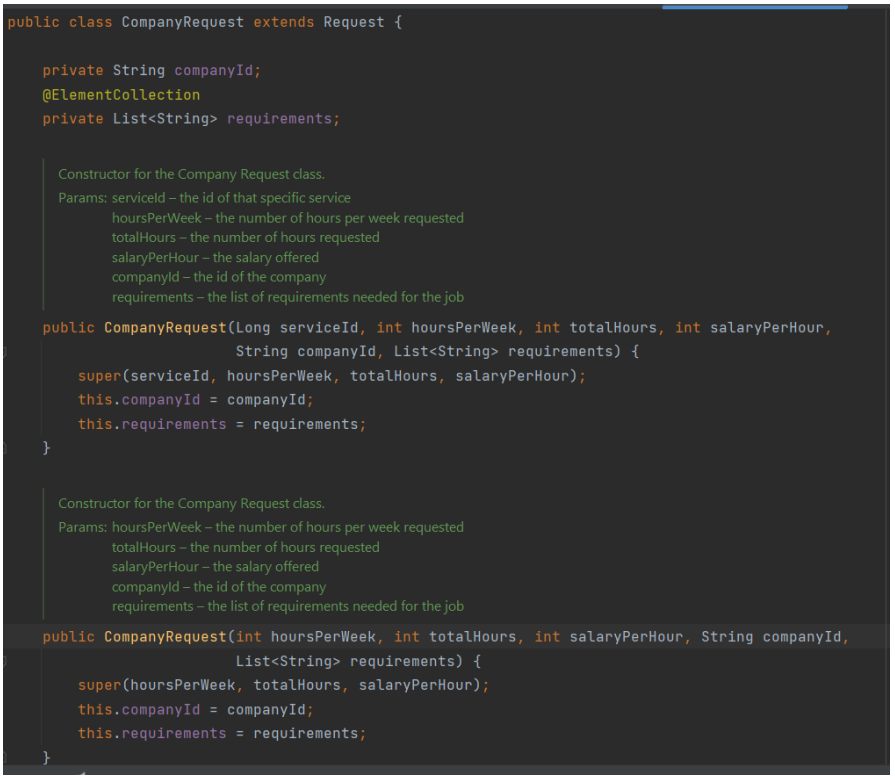

*Figure 2: The UML diagram of the Inheritance*

```
StudentRequest studentRequest1 =
    new StudentRequest( serviceId: 1L,  hoursPerWeek: 10,  totalHours: 60,  salaryPerHour: 10,  studentId: "Bogdan", List.of("C" + "++", "Java"));
StudentRequest studentRequest2 =
    new StudentRequest( serviceId: 2L,  hoursPerWeek: 8,  totalHours: 256,  salaryPerHour: 15,  studentId: "Mihai", List.of("C"));
CompanyRequest companyRequest1 =
    new CompanyRequest( serviceId: 3L,  hoursPerWeek: 20,  totalHours: 400,  salaryPerHour: 4,  companyId: "amazon94", Arrays.asList("Python", "Scala"));
CompanyRequest companyRequest2 =
    new CompanyRequest( serviceId: 4L,  hoursPerWeek: 20,  totalHours: 400,  salaryPerHour: 4,  companyId: "tud1842", Arrays.asList("C++", "Java"));
```

*Figure 3: Example of initiating multiple studentRequests and companyRequest*

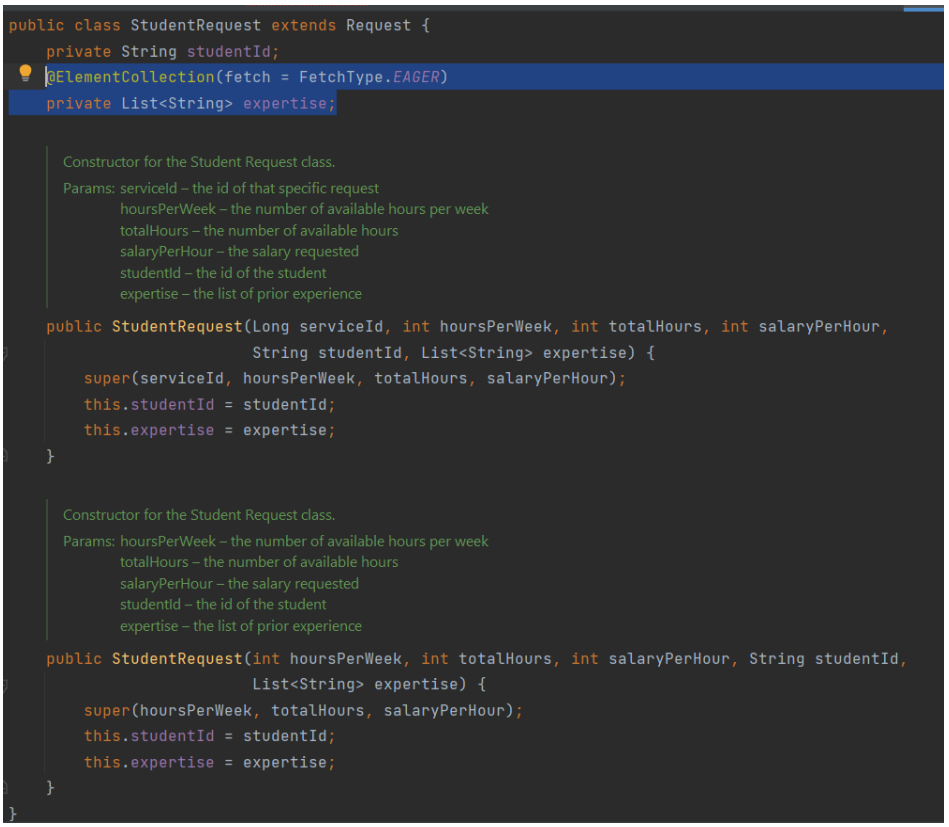
```
package nl.tudelft.sem.request.entity;

import ...

@Entity
@Data
@AllArgsConstructor
@NoArgsConstructor
public abstract class Request {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long serviceId;
    private int hoursPerWeek;
    private int totalHours;
    private int salaryPerHour;

    Constructor for the request class without ID for testing purposes.
    Params: hoursPerWeek – the number of hours per week
            totalHours – the number of hours
            salaryPerHour – the salary
    public Request(int hoursPerWeek, int totalHours, int salaryPerHour) {
        this.hoursPerWeek = hoursPerWeek;
        this.totalHours = totalHours;
        this.salaryPerHour = salaryPerHour;
    }
}
```

*Figure 4: The code of the abstract class of Request*



```java
public class CompanyRequest extends Request {

    private String companyId;
    @ElementCollection
    private List<String> requirements;

    Constructor for the Company Request class.
    Params: serviceId – the id of that specific service
            hoursPerWeek – the number of hours per week requested
            totalHours – the number of hours requested
            salaryPerHour – the salary offered
            companyId – the id of the company
            requirements – the list of requirements needed for the job
    public CompanyRequest(Long serviceId, int hoursPerWeek, int totalHours, int salaryPerHour,
                          String companyId, List<String> requirements) {
        super(serviceId, hoursPerWeek, totalHours, salaryPerHour);
        this.companyId = companyId;
        this.requirements = requirements;
    }

    Constructor for the Company Request class.
    Params: hoursPerWeek – the number of hours per week requested
            totalHours – the number of hours requested
            salaryPerHour – the salary offered
            companyId – the id of the company
            requirements – the list of requirements needed for the job
    public CompanyRequest(int hoursPerWeek, int totalHours, int salaryPerHour, String companyId,
                          List<String> requirements) {
        super(hoursPerWeek, totalHours, salaryPerHour);
        this.companyId = companyId;
        this.requirements = requirements;
    }
```

*Figure 5: The code of the class of CompanyRequest*



```java
public class StudentRequest extends Request {
    private String studentId;
    @ElementCollection(fetch = FetchType.EAGER)
    private List<String> expertise;

    Constructor for the Student Request class.
    Params: serviceId – the id of that specific request
            hoursPerWeek – the number of available hours per week
            totalHours – the number of available hours
            salaryPerHour – the salary requested
            studentId – the id of the student
            expertise – the list of prior experience
    public StudentRequest(Long serviceId, int hoursPerWeek, int totalHours, int salaryPerHour,
                          String studentId, List<String> expertise) {
        super(serviceId, hoursPerWeek, totalHours, salaryPerHour);
        this.studentId = studentId;
        this.expertise = expertise;
    }

    Constructor for the Student Request class.
    Params: hoursPerWeek – the number of available hours per week
            totalHours – the number of available hours
            salaryPerHour – the salary requested
            studentId – the id of the student
            expertise – the list of prior experience
    public StudentRequest(int hoursPerWeek, int totalHours, int salaryPerHour, String studentId,
                          List<String> expertise) {
        super(hoursPerWeek, totalHours, salaryPerHour);
        this.studentId = studentId;
        this.expertise = expertise;
    }
}
```

*Figure 6: The code of the class of StudentRequest*

# Facade

This design pattern was used in our gateway (micro-)service whose main responsibility is to check the requests for authorization and then forward them to either the student or company microservices. It also provides a simplified interface to the end user as it technically contains only the endpoints by the “callable” microservices. This makes using the API easier, because there are only 2 endpoints one needs to be acquainted with to be able to use the system. Since these two endpoints are tied into one interface, this interface resembles a simplified interface of what is happening under the hood, ultimately making the API user friendly.
Moreover, this way we are also able to isolate the microservices from the clients. The only disadvantage to our design choice is that this way the gateway is the bottleneck of the system especially when there is a large number of incoming requests and that is because by using facade we are making all of the requests go through one service thus it can become overloaded.

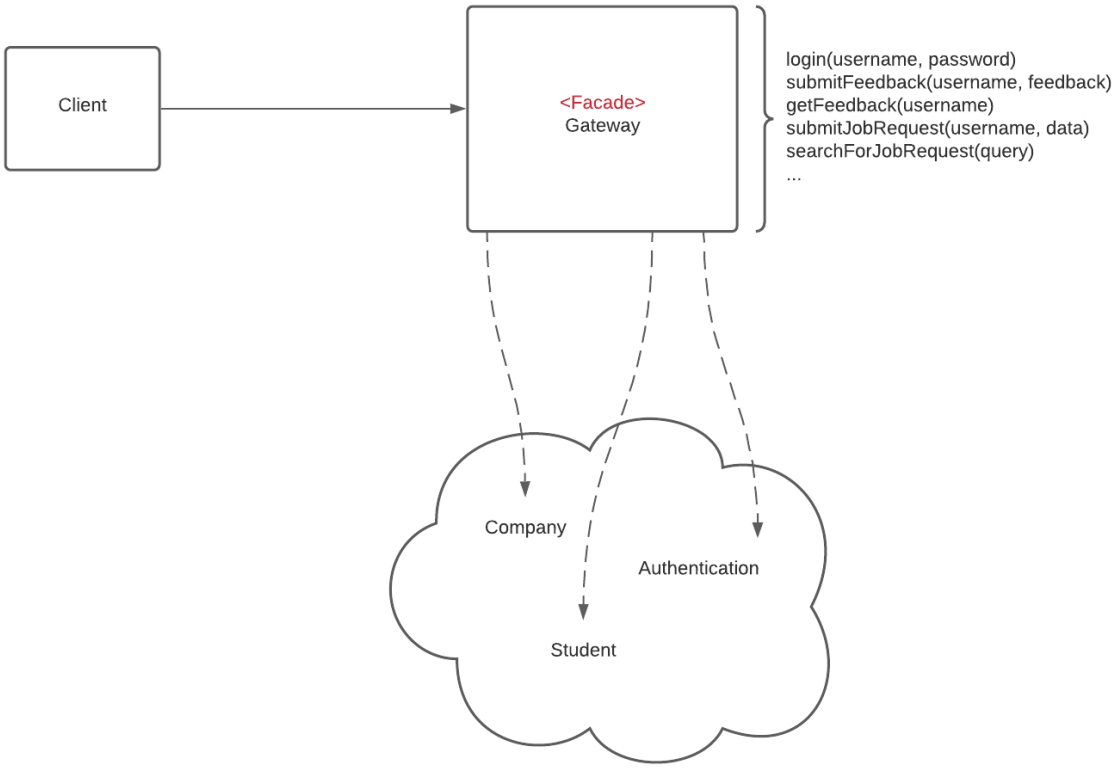



*Figure 7: Class diagram of the Facade implementation*