Assignment 1

Task 1

Index

General goal of the system

The goal of our system is to provide TU Delft employees with the ability to interface with a management system for a supercomputer called DelftBlue. The employees must be able to submit requests for computing resources, which need to be approved by a faculty reviewer. From such a scenario, three main bounded contexts can be derived: User, Faculty and some form of manager for the supercomputer's resources, which from hereon we will call Resource Manager. Additionally, each user must be notified in case one of their reservations is canceled. Therefore, we must add one more bounded context to address that: the Notification Manager. We decided to implement a microservice from each of these bounded contexts, as they fulfill the system's requirements in a simple and effective way. Whenever other bounded contexts came to mind, an excessive communication link would appear. And whenever we tried to merge those above-mentioned, an awkward and unfitting design appeared inside the microservice.

Explanation of the microservices - (function and internals)

Each of the abovementioned microservices has its own functionality and role in the bigger system. Below we explain why and how we will implement each of the microservices

1. User

The *User* microservice is a core part of this system. In it, we store all user-related information, such as the *netID* (a unique universal identifier for the user), the corresponding password, the list of faculties they belong to, and their role in the system (employee, faculty reviewer, or sysadmin). This microservice is the user's first contact point, through which the system will be able to establish their identity and ensure that they are a legitimate user. This will be done by comparing the user's input of the netID and password to that stored in the system. This microservice will also be responsible for generating tokens upon authentication of valid credentials, through which legitimacy can be confirmed in other microservices. It will also give the user access to different functionalities throughout the rest of the system based on their role, such as allowing employees to only generate requests, or enabling faculty reviewers to approve/deny pending requests to their faculties. Lastly, this microservice will even have the ability to manipulate user accounts, be that creating, deleting, or updating.

2. Faculty

The *Faculty* microservice is the most important one, as it is through it that requests for computing resources pass. Although it is named *Faculty*, it represents all faculties in the system. This decision was made with simplicity in mind. In a real case scenario, this could be easily adapted, so as to allow for X-axis scaling. Requests sent by users arrive to the faculty handler, where the appropriate scheduler is chosen. Then, they are sent to the scheduler, where they are checked to see if they can be scheduled. This is done by sending the resources, date, and faculty to the Resource Manager, which in turn will return the latest possible date, until the specified date, when there are enough resources available, or no date is sent back. If a date is received, then the requests are forwarded to the respective faculty. There they are stored in a pending requests queue, from where they can be retrieved by the faculty reviewer, who can accept or reject them. The approved requests are sent back to the faculty handler, where they follow the same path as before, but, now, they will be stored in the schedule for the respective faculty, alongside the date that was given by the Resource Manager.

3. Resource Manager

The *Resource Manager* microservice is responsible for the allocation of computing resources. These can vary in type, being CPU, GPU, or memory intensive. For these, it will have to keep track of the amount of each resource available each day, and how many of these are still available for a respective faculty. It will also allow the addition and deletion of cluster nodes, which will essentially increase/decrease the capacity of the system. This fluctuating capacity will not be reserved to any faculty, but instead it will be assigned to what is called the “free pool” - the non-faculty-reserved resources. In addition, it will also be able to receive “release” requests, in which a Faculty decides to release their reserved resources into the free pool.

4. Notification Manager

The *Notification Manager* microservice is responsible for keeping the notifications for each user. After some action is executed, it may originate a notification, such as *Request Dropped*. These will be sent by the system to this manager, which will store it for the user. The user can then poll this microservice for their notifications, which will then be sent to the user and deleted from the queue (but it will keep them persisted).

5. Communication

We are going to use asynchronous, event-driven communication between the microservices. For this, we will use the help of a tool called Kafka. Kafka is

essentially a topic-based log server, which connects producers and consumers of events in a scalable, efficient, and fault-tolerant way. For the initial contact with the system, the user will make an HTTP request to the gateway, which will have many available endpoints for the many features the system provides. After this, all communication between the microservices will be done through Kafka. In essence, a microservice publishes an event in a topic, and then, at the consumer's discretion, the event will be consumed. We chose this model as we wanted asynchronicity (as it simply fits the request flow better), in a fast, and reliable way.

6. Persistence

In the system, there will be information that will persist throughout each of the microservices. The first one would be the list of user details, which will include the user's unique netID, corresponding password, role (employee, faculty reviewer, or system admin). The requests generated by all the users will be saved too, regardless of whether they are pending, or have been accepted, denied, or dropped, as well as the user-added nodes that contribute to the free pool. Other than these three, it is vital that resource usage persists. This would mean resources for individual faculties per day, as well as that day's free pool availability. The persistence of the resource usage and list of requests is critical, as it is needed to prevent conflicts in the core functionality of the system: scheduling resources from nodes for jobs. Besides that, the notifications for each user are also stored, both seen and unseen ones.

Explanation of our DDD choices

We have considered many different design possibilities for this system, but we finally decided on the ones that best fit our requirements. There are several decisions that led to our current model. Here we will discuss the important ones in greater detail.

1. The scope of functionality of the user microservice

We debated upon whether we should have one microservice for authenticating the user and generating tokens, and another through which the user could send requests for resource usage to the faculty. As two separate microservices, almost all communication done with the authentication context passes through the *User*. In the end, we decided that it made the most sense for the system to have the two integrated into one microservice, as without the authentication the *User* did not have enough functionality to be its own microservice.

2. Letting the notification manager be a separate subdomain

The question we had for authentication - is it justifiable for it to have its own subdomain? - was also applicable to the notification manager. Just like with authentication, most communication would be carried through the user's context. However, we decided to keep them independent, as they have distinct roles. We thought that it would be better if all notifications came through a distinct notification manager, so as to prioritize the maintainability and clarity of our system.

3. Where we store the resources and the specific requests/jobs

Another major decision on our system was the choice of where to store the available resources, used resources, jobs, and whether these jobs are already accepted, or still have to be reviewed. We decided on a resource manager quite early in the process, and having faculty as its own context was also decided on rather swiftly. However, it took more time to decide which bounded-context would keep track of which information. Eventually, we decided that the resource manager would keep track of all resources, as its name suggested. The faculty context will keep track of all jobs, accepted or not, as they have to be reviewed by a faculty member.

4. Relationships throughout the system

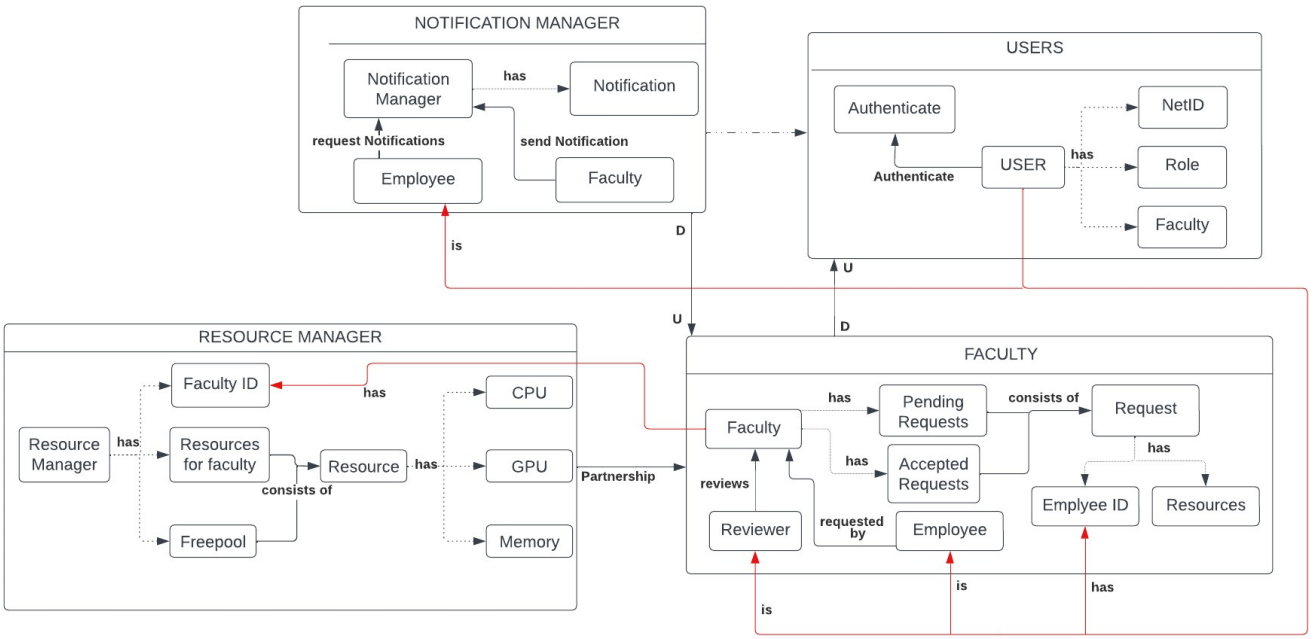
Most of the relationships in our system are basic up- and downstream ones. In these cases, the upstream (U) subdomain only influences the downstream (D) subdomain. However, one relationship stands out, namely the one between faculty and resource manager. This is a partnership. The faculty and resource manager both influence each other, while having their own dependent set of goals. This means that they are in a cooperating relationship, or in other words, a partnership.

5. Not modeling the gateway

There is a gateway in our system, which is the first link of interaction with the system. All requests are routed by the gateway, which sends each request to its appropriate microservice. This allows us to only have to implement authentication functionality in the gateway, since only requests from authenticated users may go through. However, this gateway is not a bounded context in of itself, the reason why it is not shown in the diagrams.

Diagrams

The DDD Diagram



The component diagram