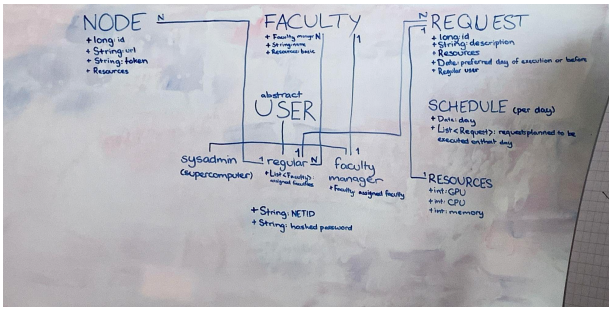
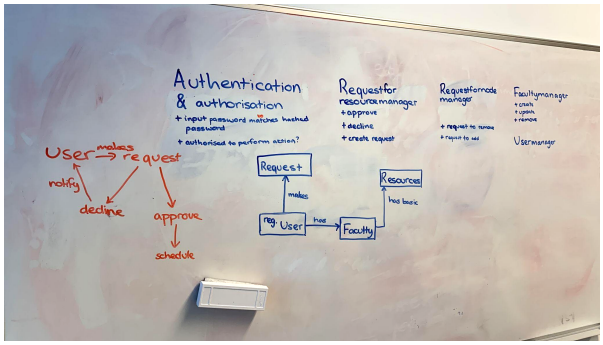
Software Architecture Report Group 14a

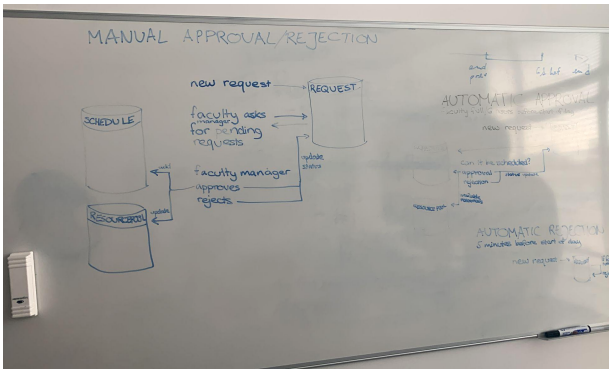
Separation and reasoning of bounded contexts

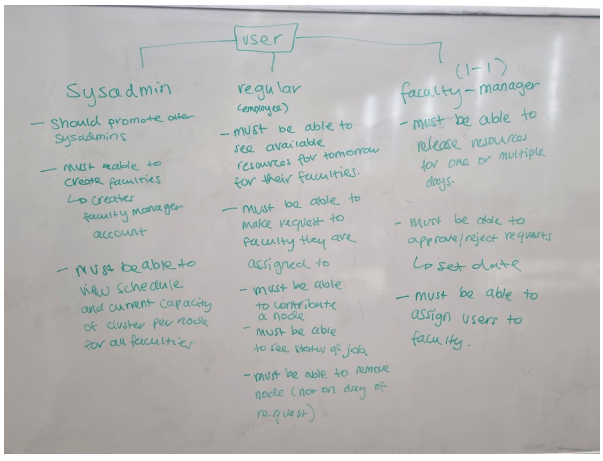
Our first goal was to create bounded contexts within our larger domain provided by the Delft Blue scenario using domain-driven design (DDD) to provide a clear separation between different parts of the business domain and to help ensure that the models used to represent those parts are kept consistent and coherent.

Below are examples of brainstorming sessions with our team to illustrate our DDD process:

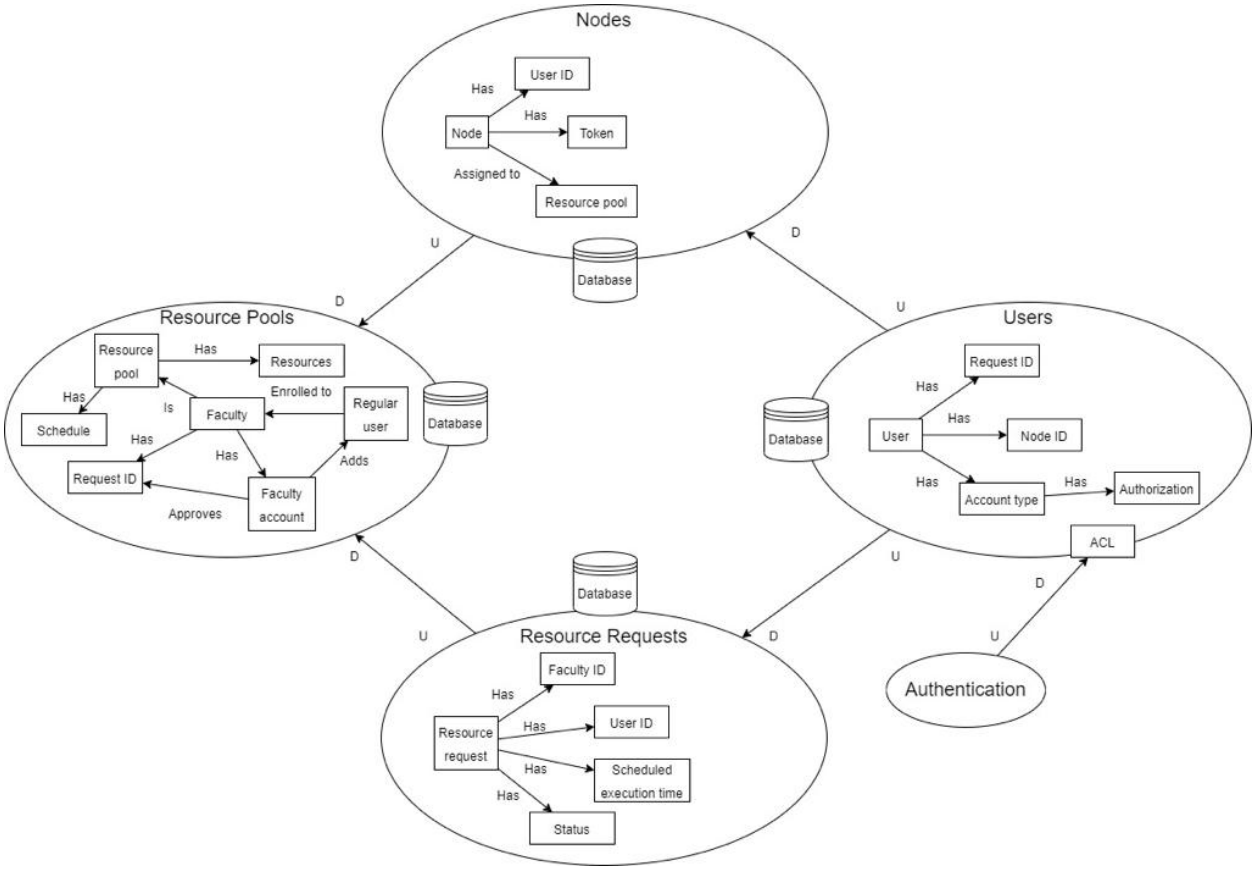








We spent a lot of time as a team brainstorming how to use ubiquitous language and clear boundaries to model the main domain given by the domain experts into bounded contexts. Below is provided the final context map of our application.



To define the bounding contexts of our system, we started by reasoning about the scenario of a user submitting any type of request to the system (not only for resources). We have 4 categories that any user of the system falls into - a sysadmin, a faculty manager which is the account a faculty uses, an employee which is an account of a faculty employee and a “floating” user which is an employee account that has not been employed to a faculty. For each one, we created a list of the type of requests they are authorized to submit. Then for every request, we thought about what the system would need to do to process that request. We recognized five bounded contexts that describe our system, that would cover the functionality of processing an arbitrary request.

Users

Finally, we have the users context which provides much of the fundamental logic surrounding users such as the user type (admin, employee, faculty manager, or a “floating” user), and keeps track of outstanding requests and nodes provided. As it has an overview of every request when it is submitted by a user, authorization of requests happens here before they propagate through the system and get executed.

Authentication

The authentication general bounded context is upstream to the user as it provides the token needed for a user to be authenticated within the system. All other contexts are logically downstream from the user context.

Resource Requests

The resource requests bounded context refers to the logic of resource requests within the domain. We also see that it is downstream from users which supplies the logic needed to create a user who would like to make a request. The resource request bounded context is upstream from the resource pool as it provides the logic for users to make resource requests to faculties.

Resource Pool

The resource pool is by far the most complicated bounded context, encapsulating the logic around faculties and the free resource pool. This includes how the faculty has users assigned to it, and a faculty manager who is able to approve requests from employees. We also store the schedule of jobs of every faculty here. Through our DDD process, we also realized that the faculty is just a free resource pool assigned to a particular faculty. This meant that the faculties and free resource pool concepts could be described in one bounded context.

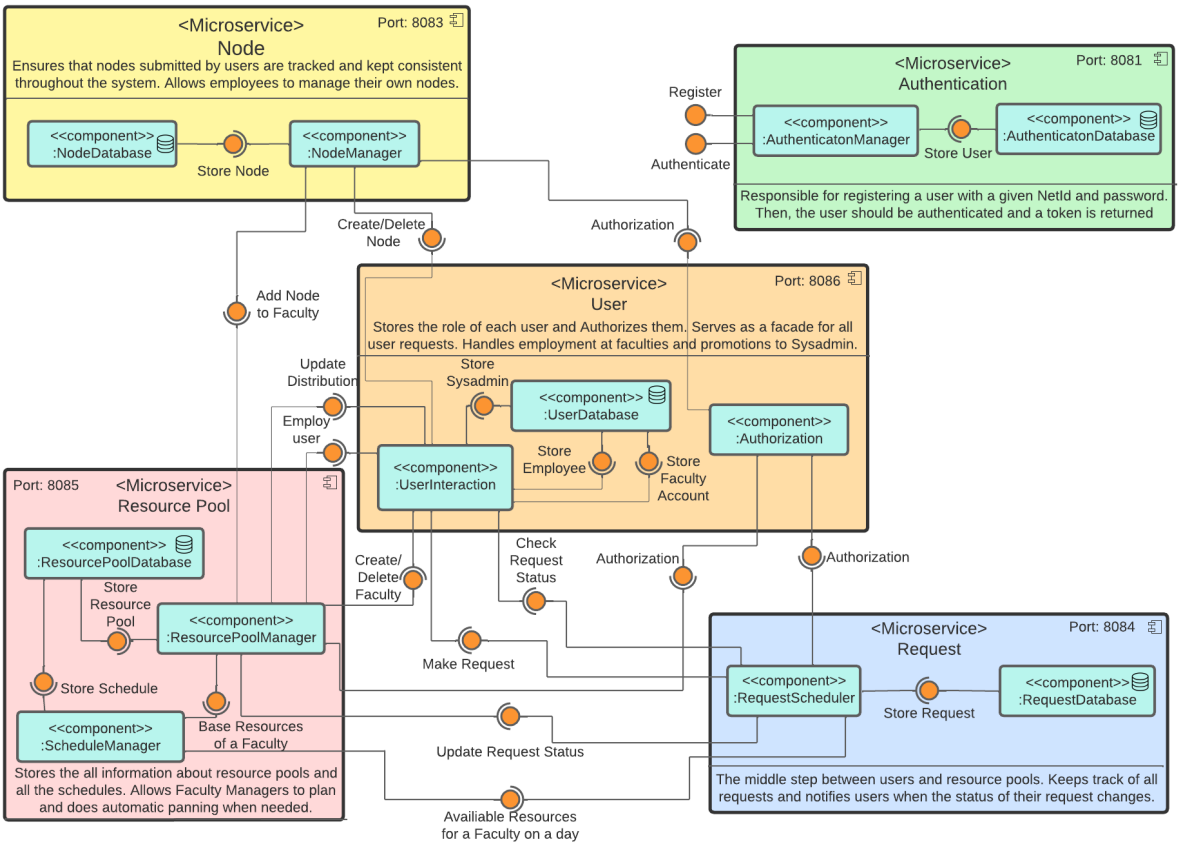
Node

The node's bounded context encapsulates the logic that allows users to contribute a node to a faculty or the free resource pool. Originally we thought about logically defining a node as a very small resource pool but the complexity that comes from users being able to add and remove nodes encouraged us to make it a separate bounded context. It is downstream from the users context because nodes strictly belong to a particular user and the node context is upstream from the resource pool context because the nodes provided are added to a particular resource pool.

How bounded contexts are mapped to microservices

We have chosen a one-to-one mapping between a bounded context and a microservice, thus we have five microservices, each corresponding to the provided bounded contexts - Users, Requests, Resource Pools, Nodes, and Authentication. Every microservice executes the functionality their bounded context has been assigned. It is also worthwhile to note that every microservice has a separate database to store its

domain data. The final architecture of our microservices can be seen in the following UML Component Diagram



Users

The Users microservice handles all functionality related to user accounts. It creates and deletes them, upgrades their permissions (promoting an employee to a sysadmin), etc. Since all accounts are stored here, we decided to implement authorization here. This is achieved by making all requests made by a user go through the Users microservice first and then, after getting authorized, be sent to their corresponding microservice. In order for a microservice to know whether a user has tried to circumvent this by sending a request directly to that microservice or it is the Users microservice that is sending the authorized request, requests sent between microservices include a password that is only known to them. This way the only microservice that accepts requests from users without a password is the Users microservice.

Authentication

The Authentication microservice is responsible for registering a user with a given NetId and password and should the user be authenticated, a token will be returned. The authentication microservice has two provided interfaces; one to register nodes and one to authenticate a user. The implementation of the authentication microservice is essentially identical to the implementation provided by the course staff, with the addition of saving a user in the User microservice database when they are registered for the first time in Authentication. Only the user microservice directly communicates with the authentication microservice. Any other downstream communication (as explained by our context map) assumes that the user microservice has ensured the user is properly authenticated.

Resource Requests

This microservice implements the process of handling a resource request, as well as all the business logic about scheduling requests. When we started designing the project architecture, at first we opted for six bounded contexts, and thus ended up with six microservices, the five we have now and an additional one - Schedule. When a user submits a resource request, it has to immediately be assigned into one of four categories: automatically accepted, automatically rejected, left pending for manual review, or left pending until the free resource pool gets more resources. We decided to put this functionality into a separate microservice as it is mostly business logic and is the core part of our whole project - scheduling the requests we get from users. However, after looking at the process of scheduling a request, we obtained a deeper understanding of how different microservices would interact and what they would provide to each other. We realized that it does not make sense to separate the logic of scheduling a request from the process of receiving it in two different microservices. Those processes are very relevant to one another and splitting them between

microservices would increase complexity, and would require a lot of redundant communication between them, which would decrease efficiency and make it harder to develop the functionality. To avoid this, we decided to remove the Schedule microservice and put its functionality into the Resource Pool and Request microservices. In the context map, you can see the components “RequestScheduler” in the Request microservice where we implement the business logic of scheduling a resource request, and “ScheduleManager” in the ResourcePool MS which handles the schedule of a resource pool.

Resource Pool

The Resource Pool microservice is responsible for the business logic behind faculties and the free resource pool. At first, we opted to design an abstract class for a resource pool and let Faculty and Free Resource Pool be child classes, but after discussing the need for an abstract class, we realized that it is not going to help with modularity, but will only increase the complexity of the application. Thus we decided to have a resource pool class that the free resource pool directly uses and let faculty extend from that. Creation and deletion of faculties and employing employees and floating users by a faculty are implemented here, as well as distributing the resources of the system between faculties. Every faculty has a faculty manager account that the faculty uses to employ and fire employees, approve or reject requests and release its resources to the free resource pool.

Node

The node microservice has two primary components - the node database and the node manager. The node manager is responsible for the creation, assignment, and deletion of nodes which are then persisted to the node database. The node manager communicates with two other microservices. Firstly, the user microservice which makes requests related to nodes and provides authentication. Secondly, the nodes microservice communicates with the faculty to assign nodes to faculties.