SEM 13a - DelftBlue assignment 1

Michiel Bakker - Bink Boëtius - Ionut Ciobanu - Alican Ekşi - Ivans Kravcevs - Nils Mikk

Task 1 - Software Architecture

Bounded Contexts

The implementation of DelftBlue will be done using microservices. For this reason, a few relevant bounded contexts have been identified using Domain-Driven Design (DDD). The User, Waiting List, Resource Node Management and Schedule contexts are core domains, since they are the most important part of the system and dictate what the system does. Authentication is the only generic domain, since it does not belong to the main focus of the system, but the system still depends on it. All the identified bounded contexts will be explained below in more detail.

Authentication

Every user needs to authenticate, this is done by SSO and is provided by the university. Users receive a JWT token from the authentication service, which they can use to authenticate themselves to other microservices.

User

There are three separate types of users: Regular Employees, Faculty Admins and System Admins. Every type of user needs to interact with the system in some way, and this context facilitates that interaction. All of them also require a NetID and a password.

Waiting List

There will be multiple pending requests that need to be approved or denied in the system. This is done by the faculty account through the waiting list context. The waiting list also handles request creation; after a user requests something, it gets placed in the waiting list.

Resource Node Management

The system administrator needs to have access to all resource nodes at any point in time. This is handled through this bounded context, along with faculty resource distribution.

Schedule

The schedule holds all approved requests per day and also deals with allocating released resources.

Microservices

Using the bounded contexts explained before, microservices were devised. These microservices and why they were chosen is elaborated upon below, also indicating to which bounded context they map.

User

Bounded context: User

The User service holds all external APIs to communicate with users, so the User microservice serves as a gateway microservice. As mentioned in the bounded context of User, there are three types of users, regular employees, faculty admins and sysadmins.

User microservice will have all the functionalities to communicate with external tools(e.g. client side), all the other microservices will be internal and will not communicate with external tools. It will be done to make our system easier to use and more secure.

There will be an API for receiving a request and forwarding it to the Waiting List microservice. This API will be usable by regular employees (role) because they are the ones that want to make use of the DelftBlue supercomputer for their research tasks. Because a user has to have access to this API to send a request, this API will be implemented in the User microservice.

There will be an API for retrieving pending requests from the Waiting List microservice, approving requests and retrieving available resources from the Resources microservice. This API will be usable by faculty admins to approve pending requests for their faculty depending on the available resources per day per faculty.

Furthermore, we will have a special API for sysadmins that will give the sysadmins the capability to manage all microservices in the system. Since this is also an external microservice, it will be implemented in the User microservice.

This microservice also stores the status of all requests, so users of this app can access information about their requests. Because the status of requests is dependent on two microservices, Waiting List and Schedule, it is decided to put the status of the request in the User microservice, a central, easy-for-user-accessible place.

Authentication

Bounded context: Authentication

The authentication microservice provides JWT tokens for a user's provided login information. This JWT token can then be used to authenticate the user in all other microservices. The JWT token also includes the role of the user because every role has a different access level.

It is decided to have the Authentication microservice separate (from for example User) to make the application more scalable. This way other authentication services could more easily substitute the existing one if needed.

This microservice also stores the users together with their roles and their passwords. This is decided because the Authentication microservice needs this information to check the validity of the credentials of the user and create the JWT token.

Waiting List

Bounded context: Waiting List

This microservice holds all pending requests indexed by faculty because there needs to be a place where the requested requests are stored while waiting to be approved and/or scheduled. The Waiting List is also there for interaction with the list of pending requests.

It allows a faculty account to retrieve pending request and forwards the request approved by the faculty admin in the User microservice to the Schedule microservice. All created requests are added directly to the waiting list. This microservice also handles the time-based logic, for example for the six hours before the day starts, when requests should be automatically accepted on a first come first serve basis. This is because the Waiting List microservice stores the information of the pending request and can thus automatically forward the right requests to the Schedule microservice.

Schedule

Bounded context: Schedule

The schedule holds the list of approved requests and the date of each request's execution, so it would store the final schedule for the DelftBlue supercomputer. Moreover, it contains logic that handles checking the available resources and, if needed, updating the available resources from the Resource microservice when trying to schedule a request.

We are assuming this microservice would handle the communication to the DelftBlue supercomputer, which is also good for scalability reasons. Considering if, in future, the university decides to share the supercomputer with other universities, the only microservice that other universities would need to access would be the Schedule microservice. Because of this, it is also a good idea to have the logic that checks and updates the available resources in this microservice, so that it is not possible for the microservice to store an invalid schedule.

Resources

Bounded context: Resource Node Management

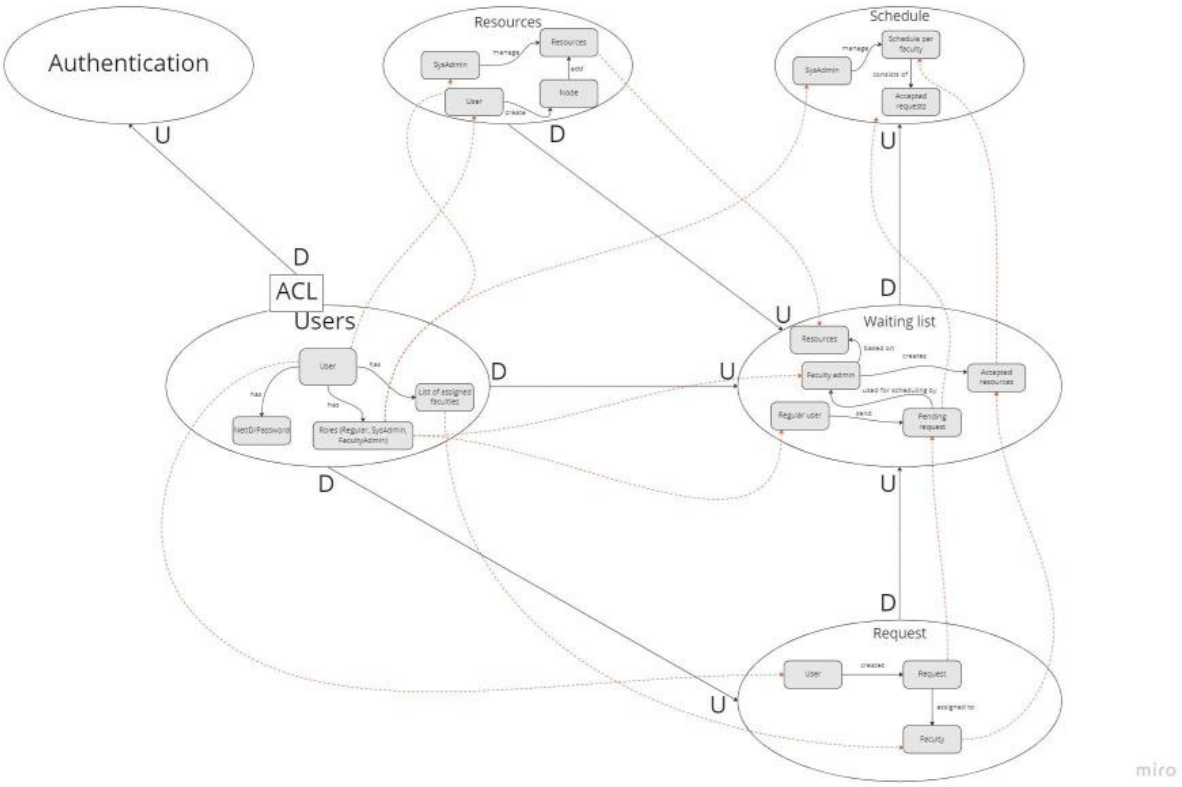
This microservice allows system admins to redistribute resources between faculties and to see the capacity of the cluster by the node via the User microservice. It also provides this information to the Schedule microservice, to check if there are enough available resources on a specific date, and to faculty admins (only showing the resources of their faculty) for when they choose which day to schedule a pending request on. This microservice also allows users to include their resources as nodes in the system and faculty admins can choose to release their resources for one or multiple days so that others can make use of them.

There is a need for a place where all the available resources are stored, which is why the Resources microservice is necessary. The added nodes from users and the released resources are also stored in the Resources microservice. This is decided because it will then be easier for other microservice to quickly get an overview of the total amount of available resources.

Diagrams

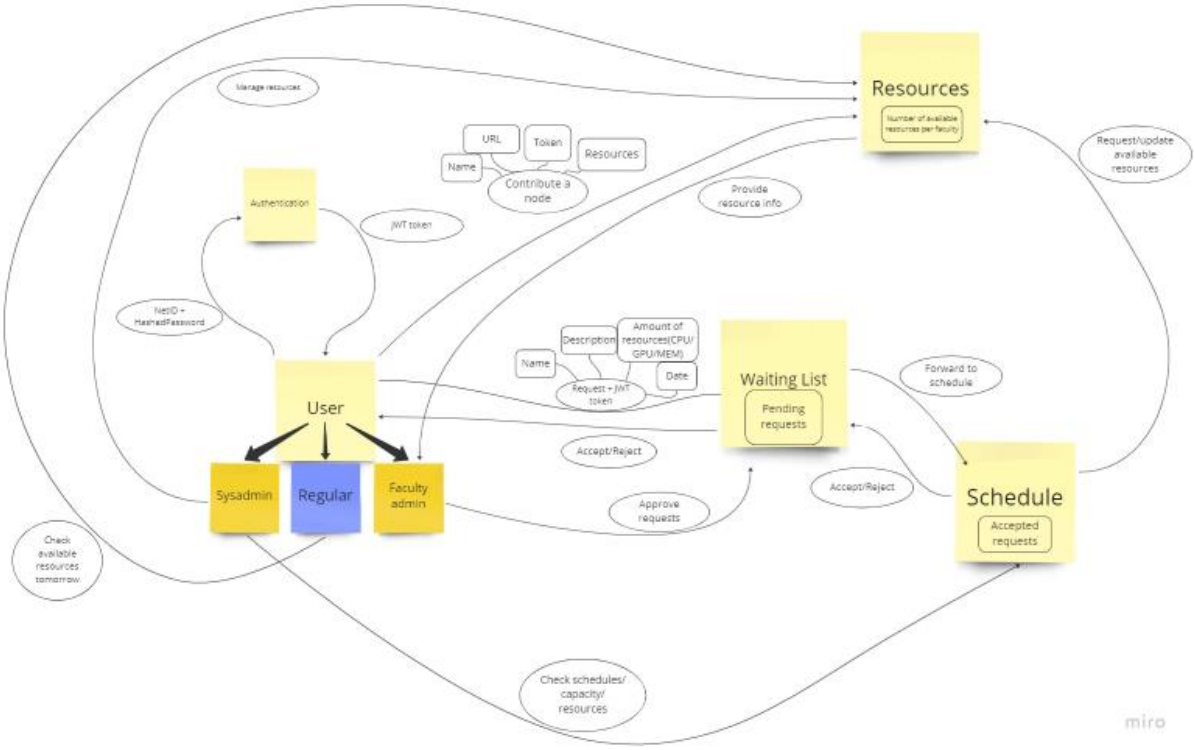
Context map

https://miro.com/app/board/uXjVP-MyAiI=/?share_link_id=236337672676

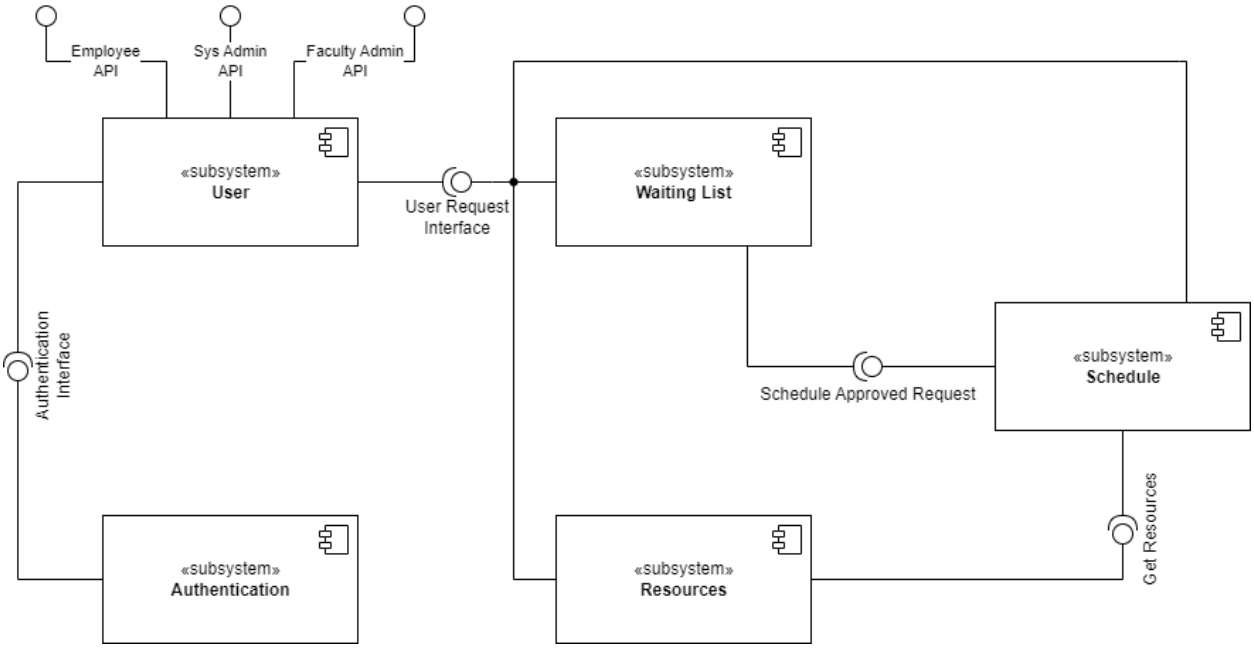


Microservices diagram

https://miro.com/app/board/uXjVP-MpMbU=/?share_link_id=953641747794



UML component diagram



Task 2 – Design Patterns

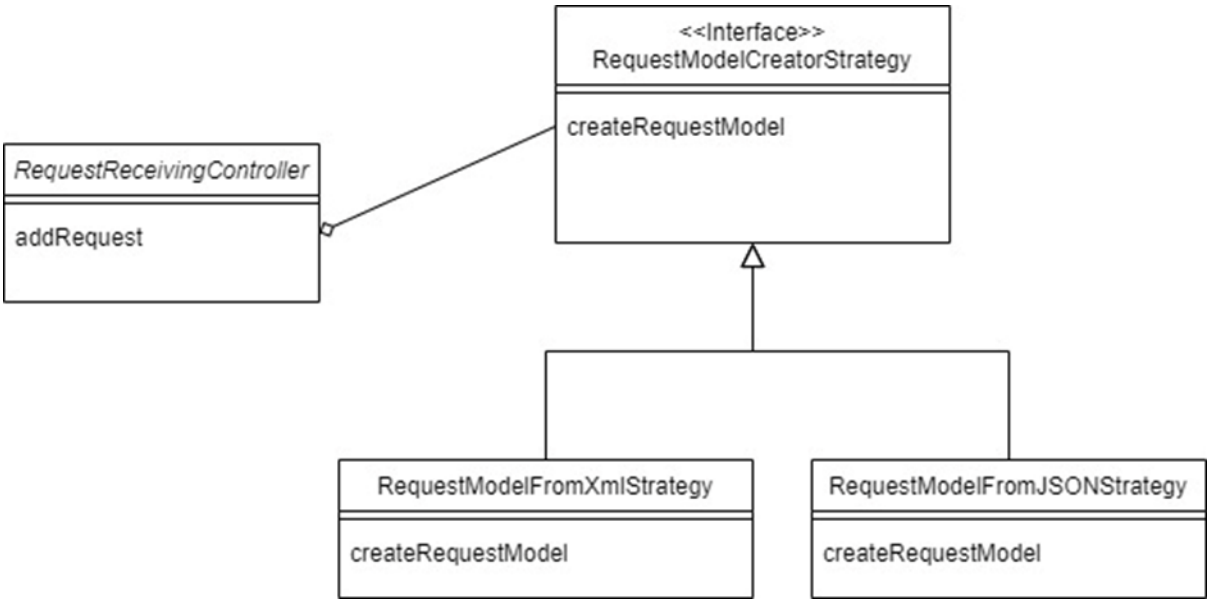
Strategy pattern

Why and how is it implemented?

Our software application should be compatible with different client applications. Because of this, it was decided to use a strategy design pattern to accept requests through the User microservice API in form of different semi-structured data formats. Therefore, we are using 2 different strategies to receive requests. This API supports now both JSON and XML formatted requests. We had the plain text strategy before as well, but it was decided to remove this strategy during the final discussion because plain text is practically not used for object transfer nowadays.

We are using a Strategy design pattern with one common interface and 2 different strategies to receive requests. A client can choose what strategy will be used by specifying the headers of HTTP requests. The controller will create a specific strategy class based on it.
All created classes are in the requestmodelget folder in the user-microservice module. The API that is using them is defined in the addRequest() method in the RequestReceivingController in the user-microservice module.

Class diagram



Adapter pattern

Why and how is it implemented?

There was a problem in the application that 2 different microservices were using different data transfer objects to communicate with one another. Therefore, we decided to use an adapter design pattern to make one microservice use the output of another microservice. We have used an adapter pattern for accepting requests in the FacultyAdminController class, which should communicate with the WaitingListController class. We are using the AcceptRequest interface and Adapter class that inherits from it and takes the WaitingListInterface as the parameter.
All created classes are in the acceptrequestadapter folder in the user-microservice module and the API that is using them is defined in the acceptRequest() method in the FacultyAdminController class in the user-microservice module.

Class diagram