Assignment 1

Group 26b: Alan Styslavski, Attila Birke, Nicolas Zink, Javier Perez, Berken Tekin, Jort Boon

Task 1: Software Architecture

Domain-Driven Design:

We began our DDD process by listing keywords relevant to the project description. The main keywords we found were:

- Authentication, login, security, contracts, employees, HR, candidates, contract details, messages, promotions, contract negotiations

From these keywords, we decided to split them into four domains:

- Authentication
    - Authentication, login, security
- Users
    - Employees, HR, candidates
- Contracts
    - Contracts, contract details, contract negotiations
- Messages
    - Messages, promotions, contract negotiations

Microservices, Domains, and Bounded Contexts:

Initially, we decided to use four microservices for the four domains we defined; however, after further discussions, we noticed that the user and authentication services would essentially serve the same purpose and authentication would mainly depend on the user service. As a result, we decided to combine these two microservices into one, reducing the total number of microservices to three.

The boundaries between the three components help us establish a clear language and models for all of the different domains. By using these bounded contexts, the system can be divided into three smaller, more manageable microservices. This has several benefits. First, it makes it easier to maintain the system as a whole, because each microservice can be managed and updated separately. Second, it allows us to assign tasks and divide responsibilities among team members with minimal overlap and confusion.

Authentication/Employee Microservice:

The authentication/employee microservice handles the creation and management of users and the authentication of these users. This microservice is also responsible for storing the users.

The authentication part of the employee microservice contains three API endpoints: register, registerAdmin and authenticate. The register endpoint creates a new user. A NetId and password is needed in the request; the user's role will be EMPLOYEE. If registerAdmin is called again, a NetId and password are required for the request, but now the user’s role will be HR. In both cases, the passwords will be hashed before they are stored in the database.

The authentication would work based on Spring Security, which means each user would have a NetId (username) and a password. After the user is logged in, they can authenticate themselves for every request with a unique JWT access token generated for each user.

The authenticate endpoint will generate a JWT access token for the given user. Again, in the request, NetId and password is required to get the token. If the username or the password is incorrect, then the user will not be authenticated, and no token will be provided.

There are additional endpoints for adding, updating, getting, and deleting users. All these endpoints can only be executed by HR, and the way to check if that’s the case is to include the JWT tokens in the header. If the role in the JWT token is HR, then the user is authenticated and can execute the API calls. Otherwise “Unauthorised” message will be thrown.

The users are stored in a database with the following attributes: an ID which is an integer and would be the primary key, a unique NetId, a string (which is the username), and a hashed password as a string. The contracts of each user would be stored in a database alongside the NetId, so a simple query would be enough to get the users’ contracts.

Contract Microservice:

The contract microservice handles managing and manipulating the contracts for employees. It provides tools for suggesting new contracts, modifying existing contracts, and changing their status within the system.

New contracts can be proposed by the HR department and negotiated with candidate employees, going from a draft to a final accepted version. In the negotiation process, both parties can modify the contract to fit their needs until reaching an agreement. Furthermore, employees can request to view their contracts and proposed changes.

Furthermore, the service ensures that only authorized users can access and manipulate contracts by verifying that they have the necessary permissions. Only users with the "HR" role are allowed to create an initial draft proposal, while all users can propose modifications to their drafts.

Additionally, this microservice interacts closely with the message service. Whenever a contract is proposed, accepted, or terminated, the opposite party is notified about the changes via this service. This ensures that all parties involved in a contract are aware of any updates in real-time.

Three primary endpoint functions support the contract negotiation process: ‘propose’, ‘accept’ and ‘terminate’. The ‘propose’ endpoint takes care of the negotiation process. A draft contract includes a reviewer field, which shows the party currently responsible for reviewing the contract and prevents unauthorized changes from being made once the contract has been proposed. The ‘accept’ and ‘terminate’ endpoints are used to update the contract status.

Message Microservice:

Users and other microservices call the Message microservice to send and receive messages or to get inboxes and outboxes associated with a specific user or the HR employees.

Employees and HR representatives can send messages to a specific Employee or a shared HR mailbox.

Users can retrieve messages they have access to. A User is authorized to retrieve a message if it is directed to them, sent by them, or the User is an HR representative. Messages are marked as unread by default; once the recipient retrieves a message, the message is marked as read.

Employees can view their inboxes and outboxes in a paginated form (100 messages per page, sorted by the date they were sent and grouped by whether they were read or not), as well as shared HR in/outboxes. Retrieving an inbox does not mark messages it contains as read.

Messages can be up to 2 thousand characters long and have a payload of references to other objects (for example, we can send a message that references several Contracts). References are stored in the database in a table with a 3-part composite primary key: *MessageId* referencing the Message the payload references, *type* indicating the type of payload (e.g. Contract), and *payloadId* referencing the actual payload [refactor this]

The microservice makes calls to the User microservice and Contract microservice to validate whether a particular user or Contract exists.

The messages have multiple distinct types, each with its own rule set enforced by the microservice (for example, a request for a copy of a document needs to be sent from an employee to an HR representative).

An overview of the architecture is visualized in Figure 1.

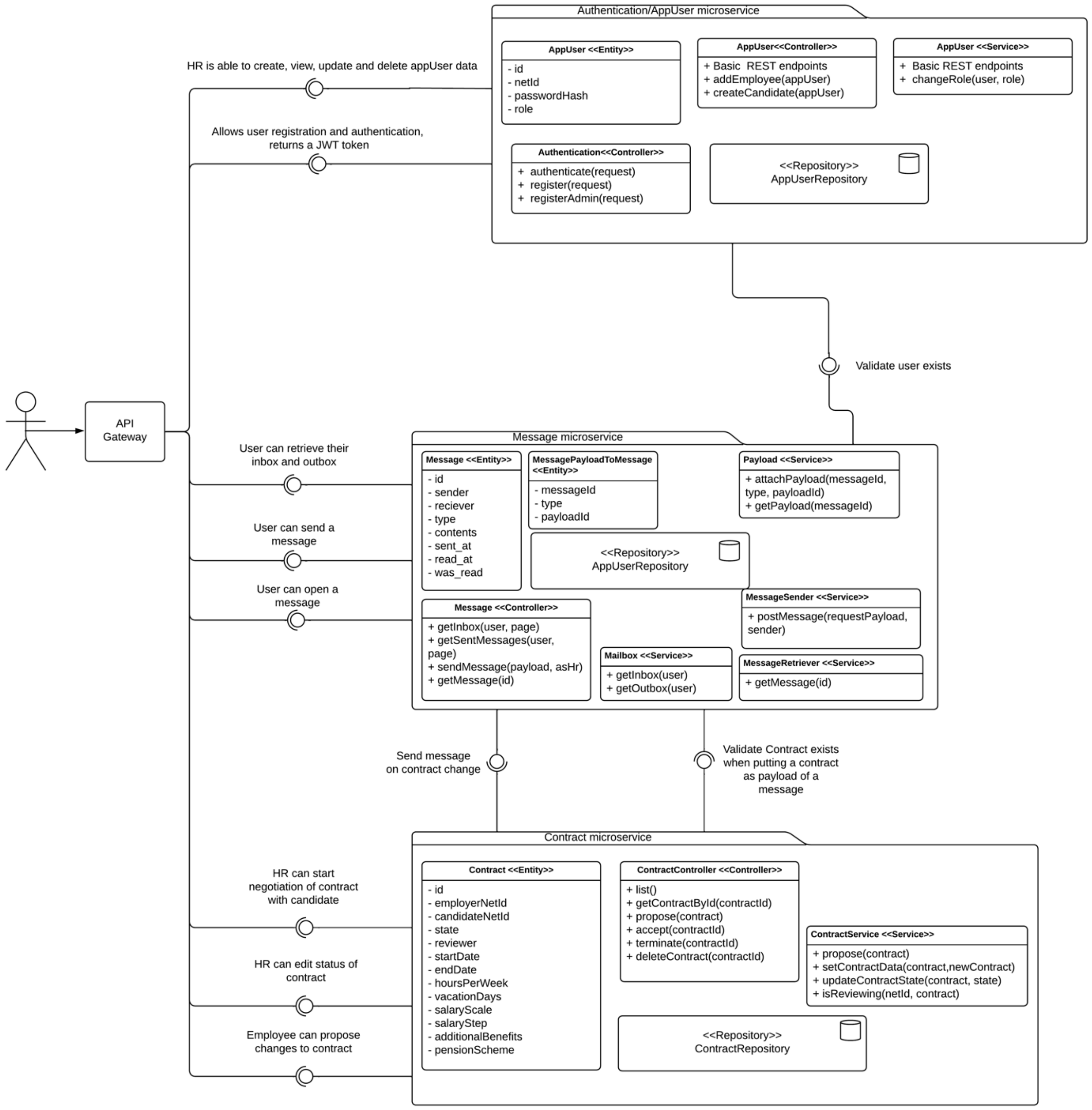


Figure 1: Software Architecture diagram for the Human Resources management system

Task 2: Design Patterns

Pattern 1: Chain of Responsibility

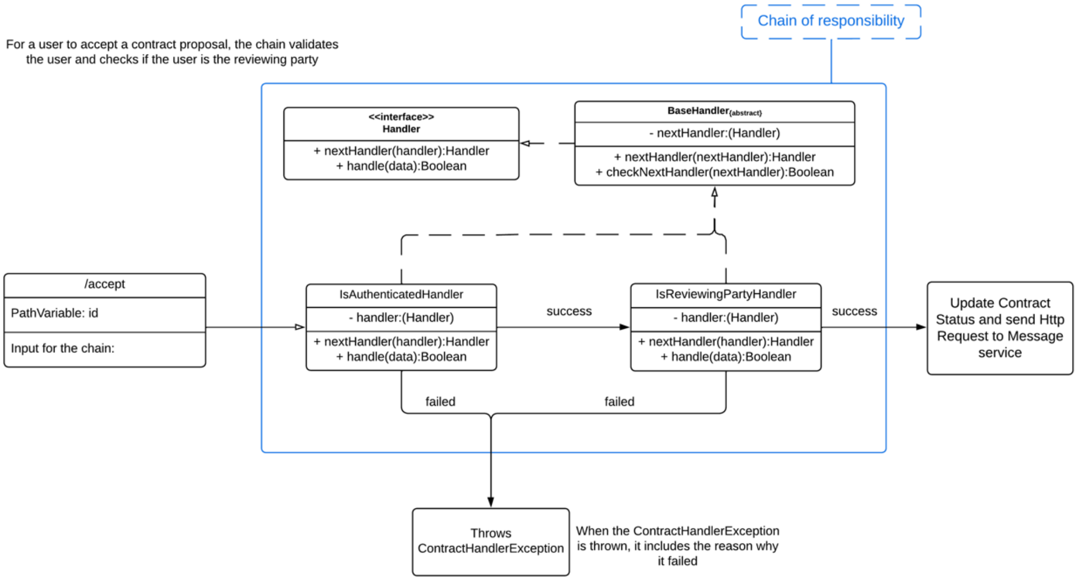
2.1:

The chain of responsibility is implemented in the contract service to handle tasks related to contracts in an organized way, since multiple handlers can be chained together to perform a task. This pattern is used for creating new candidates, proposing, accepting and terminating contracts. Each task has a specific chain of handlers responsible for checking the user's authorization and details, validating contracts and performing necessary actions. This decoupling enables a flexible and maintainable approach, since each handler is independent and can be added or removed from the chain without affecting the rest of the system. Thus, making it easy to add new functionality by modifying the handlers. This is suitable for the contract service because the tasks require many authorization checks, which can be done independently. If one part of the chain fails, the system will throw an exception.

The chains are created in the contract controller. For example, to create a new candidate, the chain consists of 4 parts before saving the final contract and user details to the database. Firstly, the chain checks if the user is authenticated, then if he belongs to the HR department and if the user details are correct, and finally if his net id is unique. All these tasks in the chain are implemented in their respective handlers. If one of them fails, thus throwing an exception, the system will not perform the final task of saving the contract to the database.

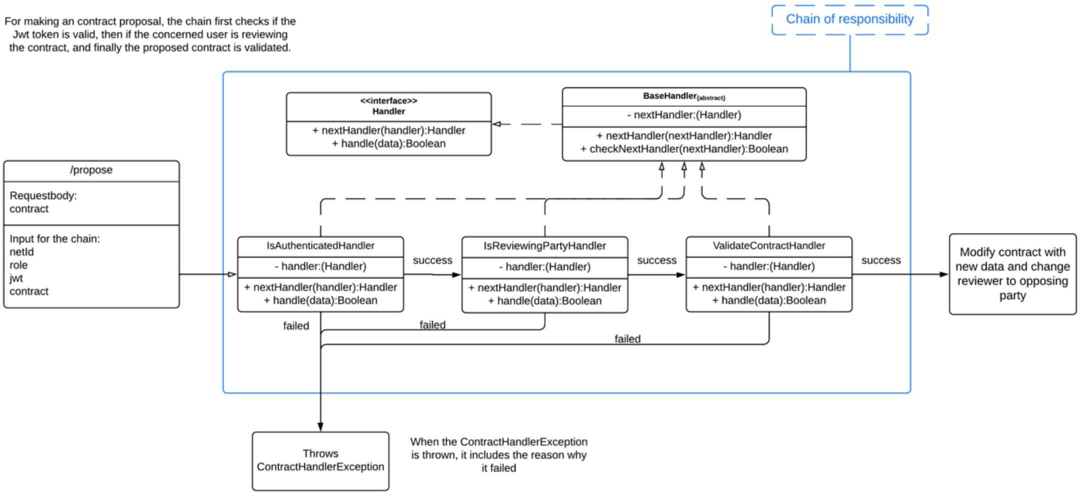
2.2: Diagram

For a user to accept a contract proposal, the chain validates the user and checks if the user is the reviewing party



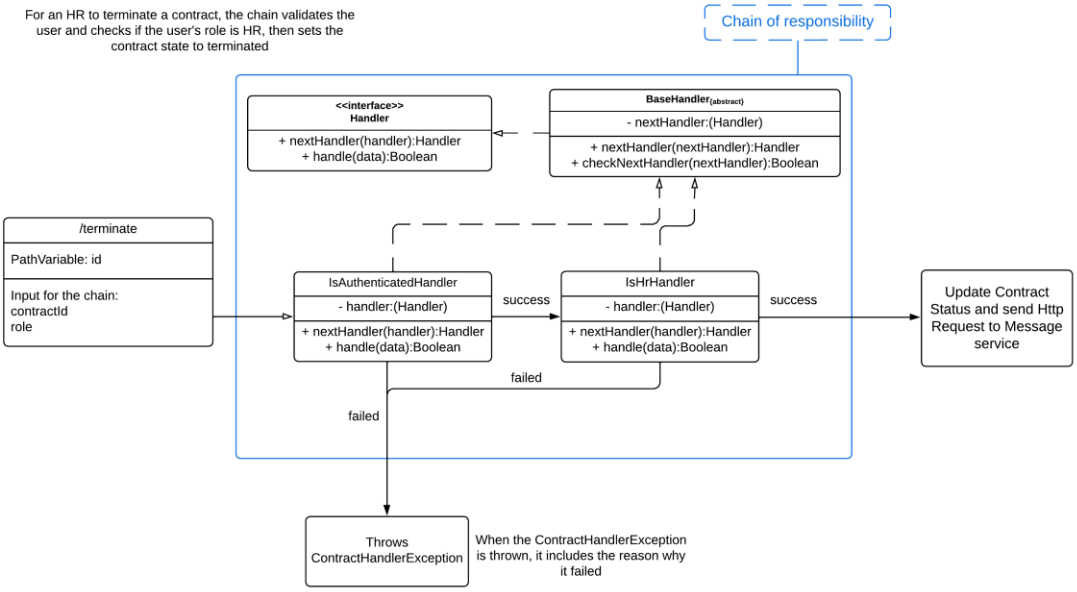
When the ContractHandlerException is thrown, it includes the reason why it failed

For making an contract proposal, the chain first checks if the Jwt token is valid, then if the concerned user is reviewing the contract, and finally the proposed contract is validated.

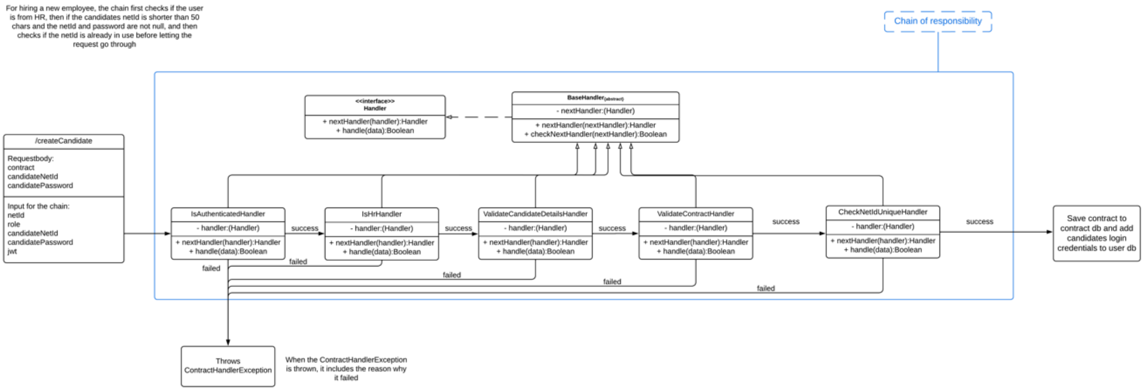


When the ContractHandlerException is thrown, it includes the reason why it failed

For an HR to terminate a contract, the chain validates the user and checks if the user's role is HR, then sets the contract state to terminated



When the ContractHandlerException is thrown, it includes the reason why it failed



For hiring a new employee, the chain first checks if the user is from HR, then if the candidates netId is shorter than 50 chars and the netId and password are not null, and then checks if the netId is already in use before letting the request go through

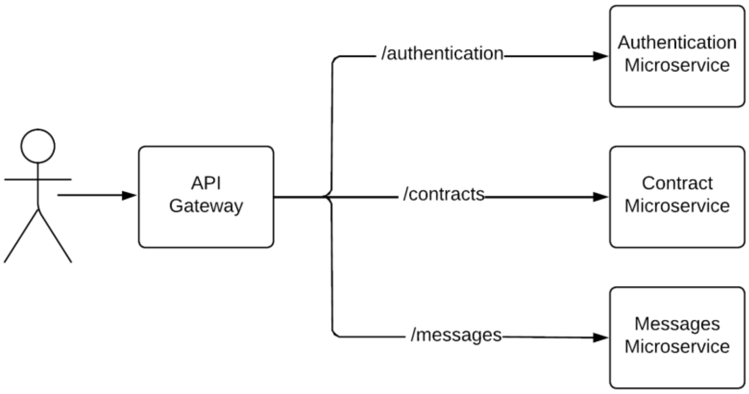
When the ContractHandlerException is thrown, it includes the reason why it failed

Figure 2: Chain Responsibility Pattern Diagram

Pattern 2: Façade

2.1:
The chain of responsibility is implemented in the contract service to handle tasks related to contracts in an organized way, since multiple handlers can be chained together to perform a task. This pattern is used for creating new candidates, proposing, accepting and terminating contracts. Each task has a specific chain of handlers responsible for checking the user's authorization and details, validating contracts and performing necessary actions. This decoupling enables a flexible and maintainable approach, since each handler is independent and can be added or removed from the chain without affecting the rest of the system. Thus, making it easy to add new functionality by modifying the handlers. This is suitable for the contract service because the tasks require many authorization checks, which can be done independently. If one part of the chain fails, the system will throw an exception.
The chains are created in the contract controller. For example, to create a new candidate, the chain consists of 4 parts before saving the final contract and user details to the database. Firstly, the chain checks if the user is authenticated, then if he belongs to the HR department and if the user details are correct, and finally if his net id is unique. All these tasks in the chain are implemented in their respective handlers. If one of them fails, thus throwing an exception, the system will not perform the final task of saving the contract to the database.

2.2: Diagram


/authentication
Authentication Microservice
API Gateway
/contracts
Contract Microservice
/messages
Messages Microservice