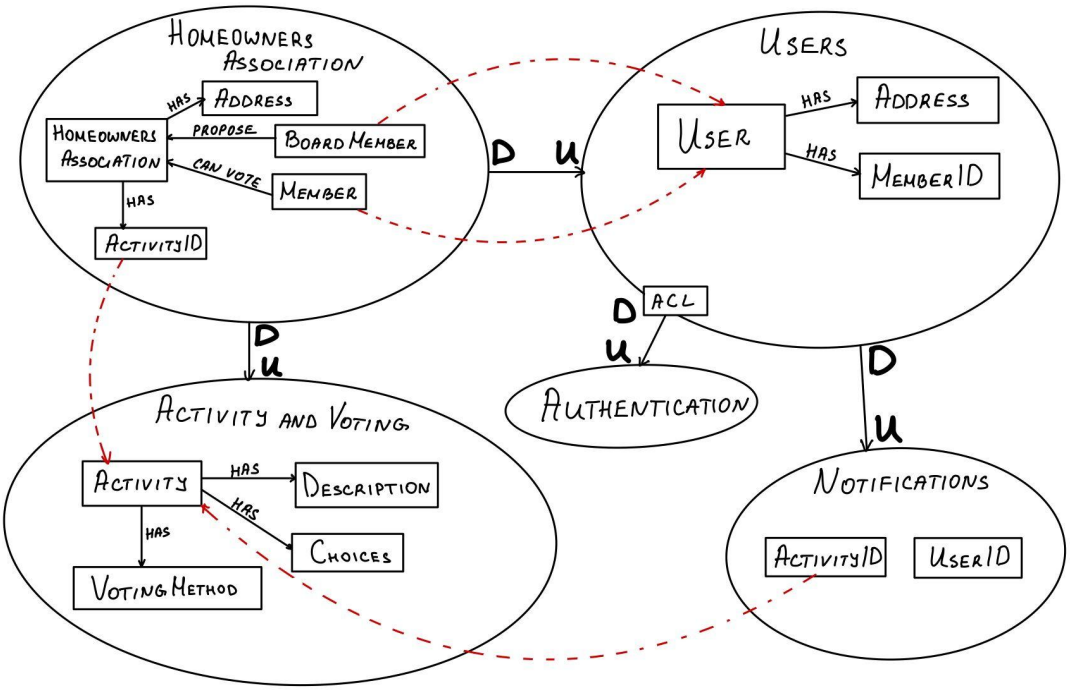
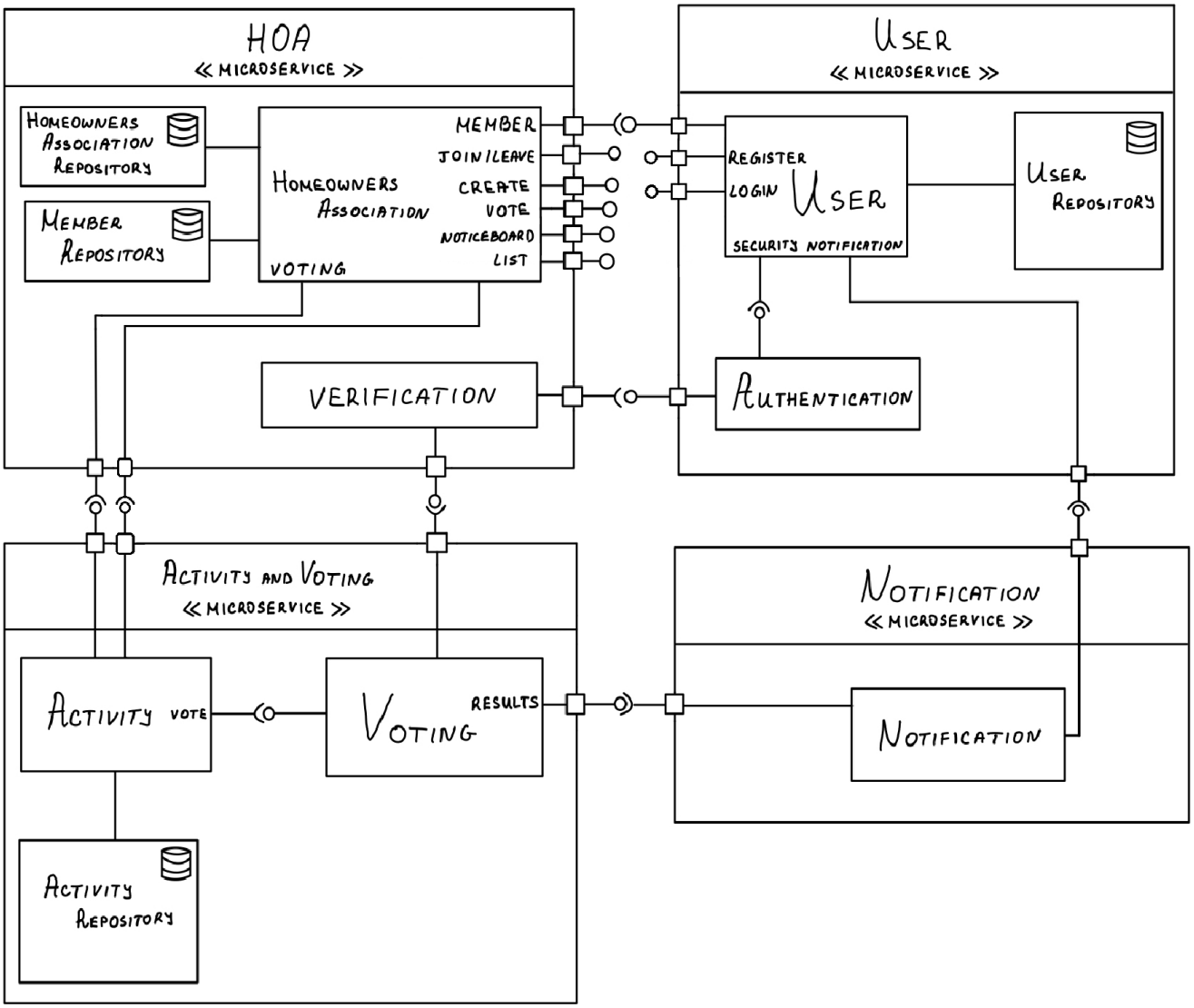
Assignment 1 23a

TASK 1



The first thing we did in order to determine the separate bounded contexts, was read the requirements carefully and identify the different subdomains in our application. After thoroughly discussing the possible separations we came to the conclusion that we will split things into four main bounded contexts. The first bounded context we identified is the users, as they need their separate logic for registration, login, etc. This bounded context will also include authentication, as it allows the user to verify themselves. We will be mapping it to a separate microservice that will hold a database and deal with the interactions between services. The second bounded context has to do with the homeowners' associations. This we are also mapping to a microservice, which will hold a database of the homeowners' associations, as well as deal with any logic between itself and other microservices. The third bounded context we identified was Activity and Voting. That is also mapped to a separate microservice, that will have a database holding all the possible activities, such as gatherings, elections, and proposals. The microservice will interact with others and will hold the logic for the vote counting. The notifications are the last bounded context we recognized. It will be its own microservice, that deals with notifying the users of events. They need to receive information about the various upcoming activities, such as proposals, voting, etc. This is possible through the use of the notification microservice which supplies the user with alerts when they are triggered.

Splitting each bounded context into a separate microservice will allow us to improve scalability and modulate our application to help for easier extensions in the future.



User microservice:

The Users microservice is responsible for storing the users and handling their authentication. In a way, it acts as a starting point, as the user must either create a new user account or login into an already existing one by providing credentials before interacting with the rest of the microservices. The service is responsible for producing JWT tokens, which would be further used by the user/client to authenticate itself in future HTTP requests to the microservices. This microservice will only manage essential user data such as a unique id and a hashed password.

We decided to include the authentication in the Users microservice as they are both relatively simple, without many requirements, and logically similar. Splitting them would create two small microservices, considering that we don't have to implement OAuth, SSO, or many domain-specific features for the users, which would make sense to separate things.

The microservice is needed as the rest of the microservices require existing authenticated users to operate. The service provides endpoints for internal usage such as getting a user email based on their id, which can be acquired from the JWT tokens.

Homeowners' association microservice:

The homeowners' association microservice is responsible for keeping track of the homeowners' associations, their members, boards, and rules, plus additional information such as reports and history. It would also provide a way to check whether a particular user is eligible to vote for a specific activity/votable event and handle the consequences of voting once notified by the activity and voting microservice. The microservice should provide endpoints for all homeowners' association-specific features apart from the notice board and the voting for elections, proposals, and reactions to activities. We decided to pack all these HOA features together, as they are all closely connected and would otherwise require a lot of inter-microservice communication. Also, considering the scale of our project is not too big, we didn't see splitting them as beneficial.

The users can use the microservice for various operations. They can view all the homeowners' associations, join homeowners' associations, see which one they are part of, and view the notice boards of the homeowners' associations they are part of. Whenever the user wants to do any of those actions, they have to send their authentication token and the homeowners' association microservice will confirm the validity of the token and get the user id from it, which will then be then used to verify their membership of a specific homeowners' association. After that, the user can complete all of the internal tasks concerning their membership. The user can also create homeowners' associations and they will have to provide their address information as well as a name for the homeowners' association. The user also has to provide their personal information such as an address when joining a homeowners' association as it is important to ensure that they are joining the correct association.

Activity and voting microservice:

When it comes to Activity, its role is to keep every event eligible for the notice board of the homeowners' association. This component will store the description of the activity and the way people have voted for it. Those events include the three main things that may happen - a social event organised by a member of the homeowners' association to which other members can respond, an election procedure for the next board members for which members of the homeowners' association can vote, proposals made by a board member of the homeowners' association for which only board members can vote. Those events seem different at first sight but they are actually pretty common in terms of characteristics. Every event has to be on the notice board when a member makes a request to see the notice board. Also, every event includes some kind of voting - the election requires voting for members, social events require voting for interest or disinterest, and the proposals require voting for or against a policy. This is why the design choice was to unite those concepts under the common component Activity, which will be responsible for the storage of the information about the activities while at the same time will provide the User Microservice with a voting procedure and the Notification Microservice with messages when a notification needs to be sent.

Other solutions have been considered as well when making design choices. One of them was to make voting a separate microservice. However, this way the number of microservices would increase which cannot be justified, as the overhead will increase, and

the pros of the scalability will be diminished, since every single activity includes voting under some form, and also there is no other component that requires voting. Another option was to leave the voting as a separate microservice and keep information about the activities within the homeowners' association component. However, this would unnecessarily bloat the information within the homeowners' association and transform it into a too-big component. Taking the above-mentioned reasons into consideration the decision came down to having the Activity component and it would provide the voting service and the storage of the information.

When a member of a homeowners' association makes a vote, verification is needed in order to check that such a person is a member of this particular homeowners' association. This is done by activity and voting microservice which requests information from the homeowners' association to conduct the aforementioned verification of the user. Apart from voting verification, homeowners' associations would also deliver additional verification information regarding other activities. In order to conduct the voting for the board members candidates need to apply for the spot on the board. However, a person applying to become a board member has to have been a member of a given homeowners' association for at least 3 years. This thing is also requested by the activity and voting microservice from the homeowners' association microservice.

The activity and voting microservice asks the homeowners' association about the verification each time instead of keeping e.g., a list of people who can vote, because it can change over time (even during the duration of a given activity), apart from that this solution would translate into bigger data transfers, which we can – and are to – avoid.

Notification microservice:

The structure of this notification microservice is quite simple, its implementation lean, it features only one singular port and component really, one for adding notifications. With this, the microservice will do nothing but make sure the notifications are then sent to the people that are to receive them. Lumping together all of these notifications into one big dedicated microservice has one chief advantage: its flexibility. This comes as now the functionality of actually disseminating the notifications is completely centralised at one microservice, making possible changes to it later on significantly easier and faster to implement than if the sending of notifications was to be handled separately by all microservices that would want to send them. Next to this, it could also aid with the scalability of the service as none of its resources are going to be used for anything but sending notifications and processing new notifications coming in, and the load of sending notifications scales pretty directly with the amount of users, which the load of the other functionalities in the microservices which use this microservice to send their notifications does not necessarily. This also meant that, with it also not being logically similar to any other tasks handled by the other microservices, it was more elegant to make it its own microservice instead of embedding it somewhere else.

The notification microservice interacts with two microservices, the user microservice and the activity and voting microservice. The interaction with the user microservice is to

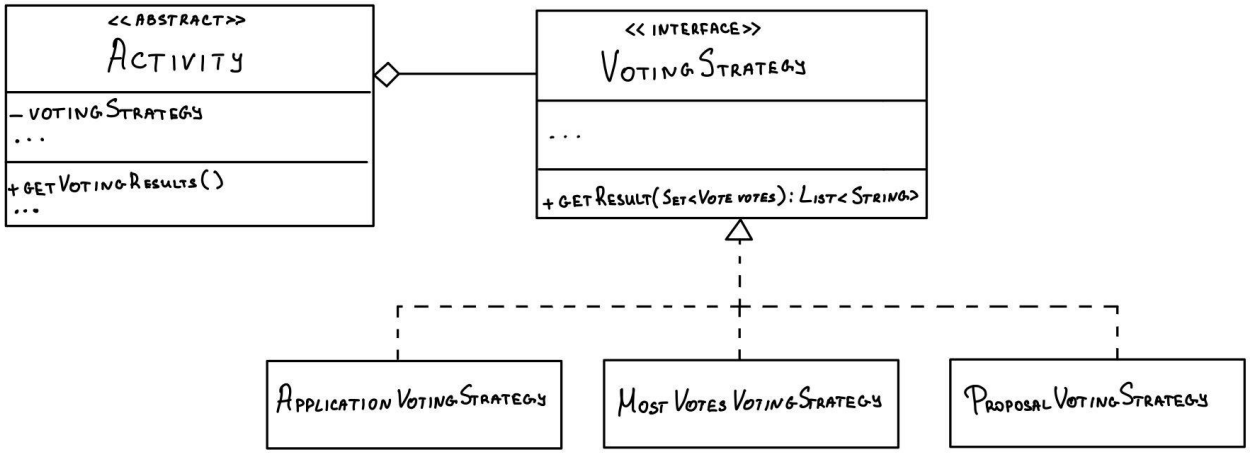
retrieve the users' email addresses to send the notifications to from any given userId string as that is what will be provided to the endpoint.

The interaction between the notification microservice on the one hand and the activities and voting microservice on the other is as one might well expect it to be. The voting microservice will upon the end of an election trigger the addition of a notification to the notification microservice, which will then notify the users that are to be notified about the election's outcome.

TASK 2

Strategy Design Pattern:

After a thorough analysis of the task given, it was immediately obvious that regardless of the kind of **activity** it would involve some type of feedback from the client- gatherings require the client to react based on their interest, elections require voting to choose the next board members, proposals require acceptance from the board members, applications require a person to apply for the board member role and so on. This commonality between the activities is what led to the decision to unite them under one **Abstract class Activity.** However, the different kinds of events require different kinds of voting. Some require a boolean **yes or no** answer (Proposals), others require **top n** choices (Elections) and so on. At the same time voting and making choices are yet still the same. This logically led to the conclusion to use the Strategy Design Pattern for the result produced from the voting procedure.



In the above-shown diagram it can be seen how the classes are structured. There is an **abstract class Activity** from which **Application, Election, Gathering, and Proposal** extend and implement the methods and fields of activity. Activity has a private field with the voting strategy. This is because some activities can reuse another voting strategy. For example, Gathering and Proposal both return a definitive yes/no answer. There is the **interface VotingStrategy** which has the public method **getResult**() which should be implemented in the classes implementing the interface. There are three different strategies currently used - **ApplicationVotingStrategy** which returns all users who gave a positive answer to the question whether they apply for the position, the **MostVotesVotingStrategy** which returns the top n choices and the **ProposalVotingStrategy** which returns a positive or negative result based on the votes given.

The code and the concrete implementation of the design pattern can be seen in activity-and-election microservice in domain/activities/voting where the interface and

implementations of the voting strategies are positioned and in domain/activities where the Activity classes are implemented

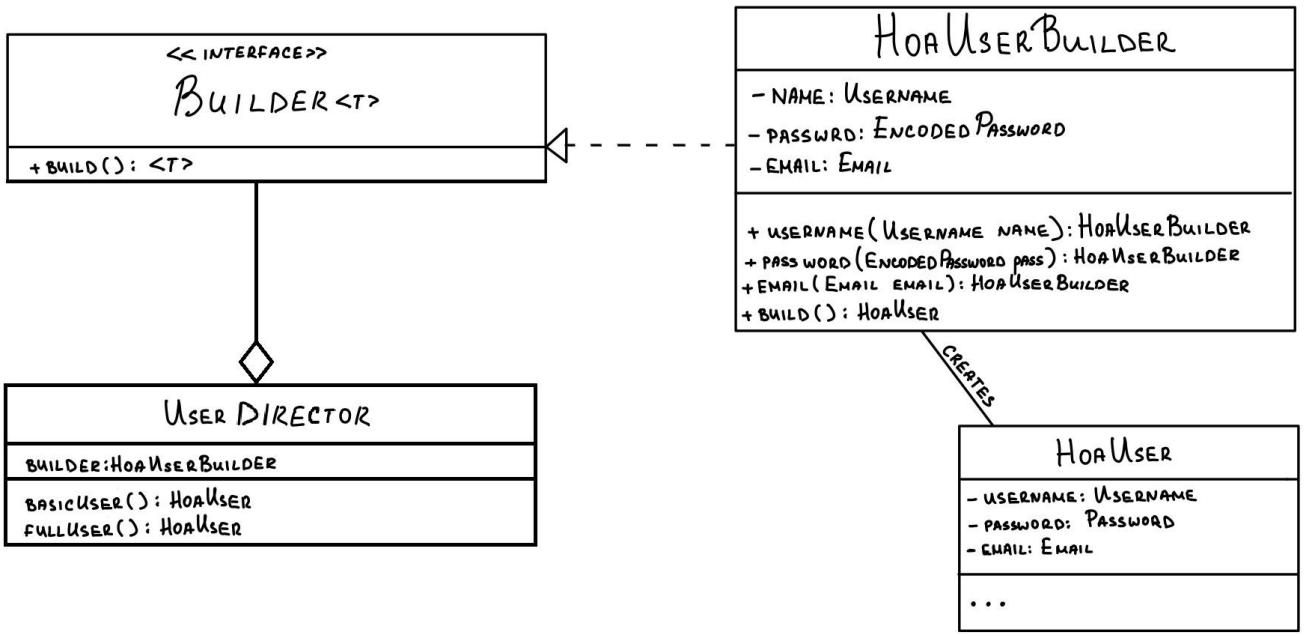
Builder Design Pattern:

As developers, we aim to create the best software possible. In order to do that we also need to improve our efficiency, and the way we do things by focusing on the important parts of the application. This is why we decided to use the Builder design pattern in our project.

The builder design pattern is a creational pattern that allows the construction of complex items, through a simple step-by-step process and it is a design pattern that improves the quality of life of the developer.

The builder design pattern allows for:

- Separation of concerns: It separates the construction process of complex objects from their representation, and allows for easier modification of the construction process.
- Improved readability: It helps the developer inspect the construction process easily and apply any changes needed to the small and more manageable parts of the construction process.
- Flexibility: A separate builder can be made from the already existing interface, which would help with the creation of new objects with varying configurations.
- Reusability of code: It allows the developers to reuse the builder for new objects that follow the same construction process, but have minor differences.
- Ease of use: Thanks to the fragmented and straightforward process the developer can efficiently create new complex objects.



Here is what the implementation of the builder design pattern for the HoaUser class looks like in our project. The ‘`Builder`’ interface is located inside the commons module, as every microservice should be able to create builders using this common interface, in order to have a similar skeleton and avoid code duplication. The concrete builders can be found in the corresponding microservices’ modules.

Builders are also useful when we have optional fields that are not necessary for the creation of an object. This helps by not having to create a separate constructor for each possible combination, or remember the order of the arguments in the constructor. This is all taken care of by the builder. Our ‘`Builder`’ interface throws a ‘`NotAllNecessaryFieldsAddedException`’ in case not all of the compulsory fields are set, saying which field is not set. This helps the developer quickly recognize when they have made a mistake.

We have also added a `Director` to our builder design pattern, which provides a higher-level abstraction that allows us to quickly build default users with either only the necessary information or along with the optional fields. This is in a way a Facade that aims to simplify the process when it comes to testing for example and we just need a default user.

Throughout the whole project, we’re making use of the Builder design pattern, but in order to optimize for time, as we are a team of five, we have decided to implement it for the `HoaUser` class and as for the others we settled for the convenient lombok annotation `@Builder`. This allows us to take advantage of the design pattern, while not having to write all of the boilerplate that comes with it.