Software Engineering Methods

Assignment 1 Part 1 - Scenario Homeowner's Association

Student name	Student number
Rafael Petouris	4776968
Jelt Jongsma	5496594
Vladimir Petkov	5447194
Bram Snelten	5519365
Alex Brown	5171709
Roland Bockholt	5534283

Figure 1: Image of a house model titled Homeowner's Association [1].

CSE2115

Group 19a

Delft University of Technology, The Netherlands

27 November 2022

Contents

1. Bounded Contexts & Microservices

1.1 Domain-Driven Design

Before we can start programming a Home Owners Association (HOA) System, we need to design our system so we will not run into errors and mistakes down the road. To start we looked at architecture patterns to get an idea for the high level architecture. We found that Domain-Driven Design (DDD) would fit best for this system. DDD maps functionality into bounded contexts which can then be used to determine how a system should be implemented.

By looking at the requirements, other examples and feedback from Sergey Datskiv, our teaching assistant (TA), and then determining the different functionalities and keywords of the HOA system, we were able to find the following contexts:

- Home Owners Association
- Activities
- Elections
- Proposals
- Reporting
- History
- Notifications
- Users
- Authentication

All of these contain functionality that can not be split any further while still being meaningful contexts. Elections, for example, could be split into applying and voting, but this would not make any sense from a DDD point of view.

1.1.1 Architectural choices

After figuring out which bounded contexts exist in the HOA System, we had to decide how we wanted to implement them as microservices, as this is a requirement from the client.

From other examples we noticed that the user and authentication contexts are generally their own microservices. We have chosen to merge these two services into one since the user microservice will have very little logic and few responsibilities. Our reasoning is that using very small

microservices will make the communications a lot more complex and the microservices do not become more scaleable or testable. Thus losing the advantage of microservices The combined user microservice is a generic service; we store the role of the users in the other services and kept the user service very simple. This means that the user service could be used in another application or that the application could be extended with other services that use the user service.

At first, we decided to group all other contexts into one HOA microservice, because all of their functionality is so closely related and system specific. After a meeting with our TA, we came to the conclusion that one single HOA microservice would have too much functionality and complexity for one microservice. Reexamining the contexts again we determined that the activities context could be separated from the HOA and become its own microservice. The activities service has enough complexity to become desirable to scale independently and activities could be a concept that could be used by other services should the application be extended.

We could also have created a voting service that handles elections and proposals, but since elections and proposals rely on the members and member roles from the HOA System, they make no sense outside of the HOA System because it would require a lot of communication with the HOA service and increase complexity. For this reason, we decided against it. The same logic applies to reporting, notifications, and history as reporting also relies on the rules from the HOA System, and notifications and history need to contain HOA System-specific information. This final split results in three microservices:

- Activities
- Home Owners Association
  - Elections
  - Proposals
  - Reporting
  - History
  - Notifications
- Users
  - Authentication

Of course, extracting other features from the HOA is still possible, like extracting elections, reports, or even history, however, we decided that if we extract the elections component as a separate microservice the HOA itself would lose some key functionality and serve as a largely redundant microservice. Therefore, we believe that all logic surrounding the voting and rules

of the HOA should be encapsulated.

The activities microservice, as well as users were extracted as they could be possibly integrated with other systems that way due to their functionality. For example, another platform might want to reuse our code for users or activities microservices. Having the a microservice for the reporting alone does not make sense in our opinion, because it would contain not much functionality and not be complex enough to be justified as a separate service.

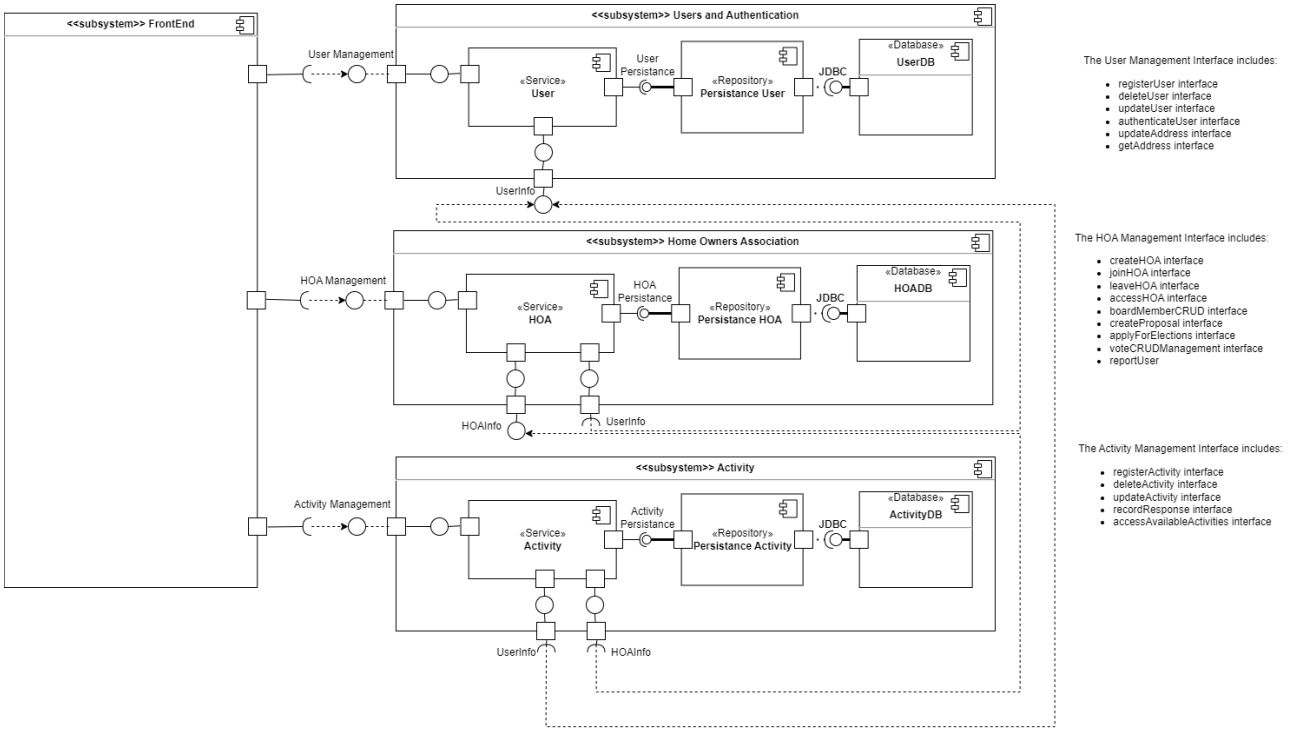

Figure 1.1: Component Diagram for the implementation of the Homeowner's Association

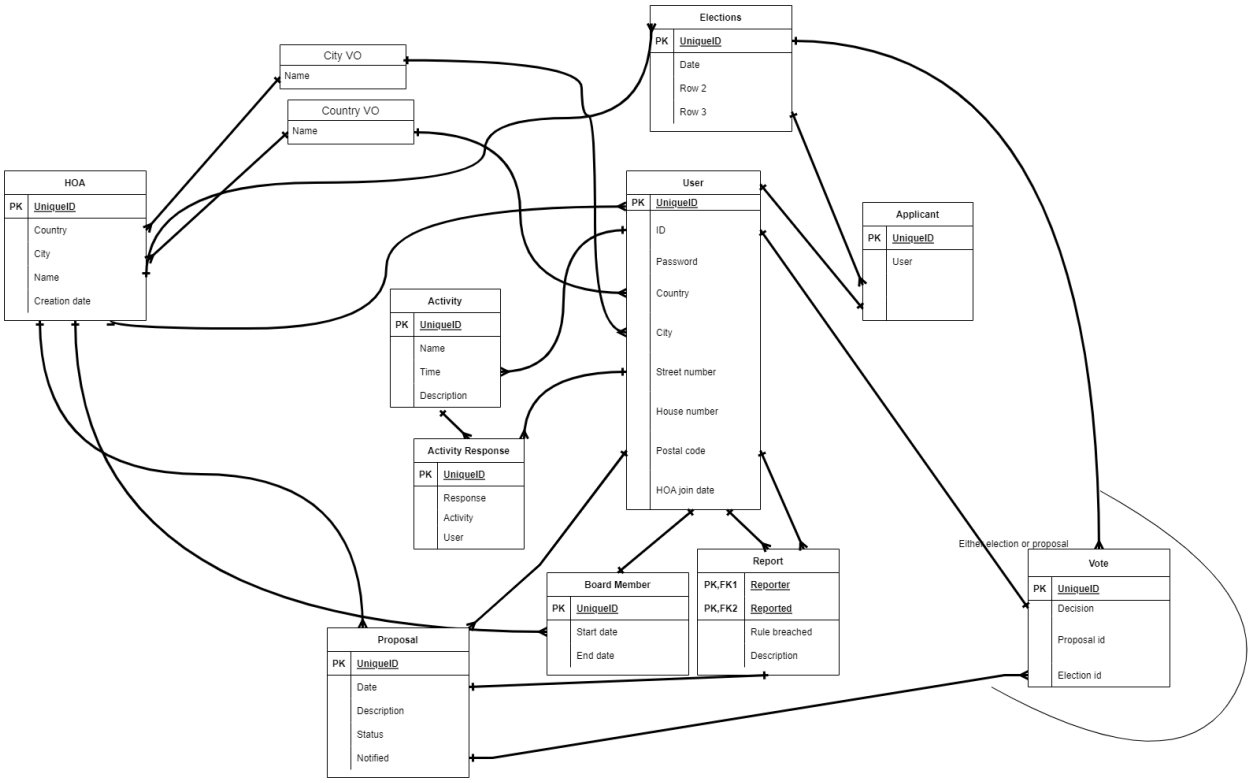

Figure 1.2: Relational schema of the Homeowner's Association

1.2 Microservices

1.2.1 Users

The User microservice is responsible for registering, authentication and security. It allows users to log-in to the system and edit/add their personal information. An account consists of the userId and a password; passwords are stored in the database as hashed strings. If a user is logged in and sends a request to any of the other microservices, Users sends the personal authentication token to the microservice, and then the user is authenticated. The User database stores the accounts of all of the users including those without HOA.

To edit the personal information of a user, a request is sent to the API updateUserInfo, the user must be authenticated and then the personal information can be updated in the database. In order to delete a User account, a request needs to be sent to the deleteUser API, the user is then authenticated and the user details in the database are deleted.

1.2.2 HOA

The HOA service implements most of the functionality. It allows users that have registered to join an HOA. When a user is a member of an HOA, they can see the names of all HOA members and board members, apply to be a member of the board, and vote for the board members. Members of the board can request a change in the rules for the HOA, and only board members can vote on that proposal. If a majority of board members vote for the proposal, the proposal is approved and the rules of the HOA are changed. A member of an HOA can report another member of the same HOA for a suspected rule violation.

In order for a user to join an HOA, the user sends a request to the join API with the authentication token in the header and the ID of the HOA in the body. The authentication manager then sends a request to the user micro service to authenticate the user. If that is successful, there is a check if the user is already in an HOA. If that is the case, the user cannot join, however, if the user is not part of an HOA yet, it is checked whether the address of the user makes him eligible to join or not. If he is eligible, he can join the HOA and his membership and the date are stored in the database.

Once a user wants to leave an HOA, he can do so by making a request to the leave API with the authentication token in the header. Again the authentication manager sends a request to authenticate the user to the user microservice. Then it is checked if the user is indeed in an HOA, if that is the case, the entry that the user is a member of the HOA is deleted in the database, and thus the user leaves the HOA.

In order to become a board member, one first needs to apply to be a candidate. This can be done by sending a request to the apply-board API. Again, the same authentication as previously described takes place. If the User is longer in the HOA than the minimum required time, the membership status is updated to Applicant. Once a vote takes place, a vote can be cast through the vote interface and later be changed. Once the vote is closed, the membership status of those elected is updated in the database and the election result is stored in the database for the HOA history. To access the history of the HOA, which contains all election results, rules, and activities, a request to the API is made, the user must be authenticated and then the history is provided.

To change the rules of the HOA, a board member needs to submit a request to the API, be authenticated, and check if the request indeed comes from a board member and is then stored in the database. After 14 days, a request becomes a proposal and a vote takes place. All board members can see all the votes by sending a request to the API. Through the voting interface, all board members can vote on the proposal. When the vote closes, the result of the vote is

stored in the database, the rules are changed in the database. To report another member of an HOA for rule violations, a request needs to be sent to the report interface and the report is stored in the database. Every member can get the reports they filed requesting them from the report interface, as well as all reports that were filed against them.

1.2.3 Activities

Activities allows users in an HOA to create activities. An activity is specific to an HOA and has a description and a date. While an activity is active it will be visible to all users in that HOA on the noticeboard, on the noticeboard other users can declare interest in the active activities by submitting their responses to it. Once the date on the activity has passed the activity becomes inactive and can only be seen in the history of the HOA.

In order to access all the current activities a request is sent to the getActivity API of the Activities microservice. First of all, the user has to be authenticated as described earlier. Then the Activities microservice makes a request to the HOA microservice to the findHOA API with the NetId of the user and receives the HoaId of the user and provides all the activities that are part of the user's HOA.

Bibliography

[1] Nick Youngson. *Homeowners association*. Retrieved: 27 November 2022. URL: https://pix4free.org/assets/library/2021-10-24/originals/homeowners-association.jpg.