Advanced Software Engineering

DESIGN REPORT

Team number:	0508

Team member 1	
Name:	Klavio Tarka
Student ID:	12146253
E-mail address:	a12146253@unet.univie.ac.at

Team member 2	
Name:	Michael Phan
Student ID:	11839153
E-mail address:	a11839153@unet.univie.ac.at

Team member 3	
Name:	Kevin Richardo Grote
Student ID:	12143222
E-mail address:	a12143222@unet.univie.ac.at

1 DESIGN DRAFT

1.1 DESIGN APPROACH AND OVERVIEW

1.1.1 Assumptions

Assumptions:

- The maintainer will only have access to the health check system, he will check if the service is alive or dead.
- There can be multiple devices in the house that can control rooms and the devices in each room.
- The user can only interact with their devices once he's logged in.
- Each service shall have it's own database, unrestricted by other databases.
- Each service shall have it's own API endpoints and can only communicate with other classes via these points.
- Each action will be followed by a notification via E-Mail.

1.1.2 Design Decisions

Design Decisions on the logical level:

- Decision for using an n-to-m mapping between devices and rooms and not simply devices being contained in rooms. This would allow us to also support devices that span multiple rooms like a heating system.
- Decision to have time-based actions service only store the operations and the time events to then address the controller. This reduces the need to have any too tight coordination and we can mostly just use conformist patterns (with ACL) and not worry about any shared kernels, etc.

Decision to use sequence diagrams because they allow the detailed description of communication flows.

1.1.3 Design Overview

We have finished developing all the architectural views for the 4 + 1 model, and beforehand we also worked with the context map which gave us a basic idea of how the system should work and communicate with each other.

To document our team's progression we have included the team's evolving ubiquitous language in the following context map iterations.

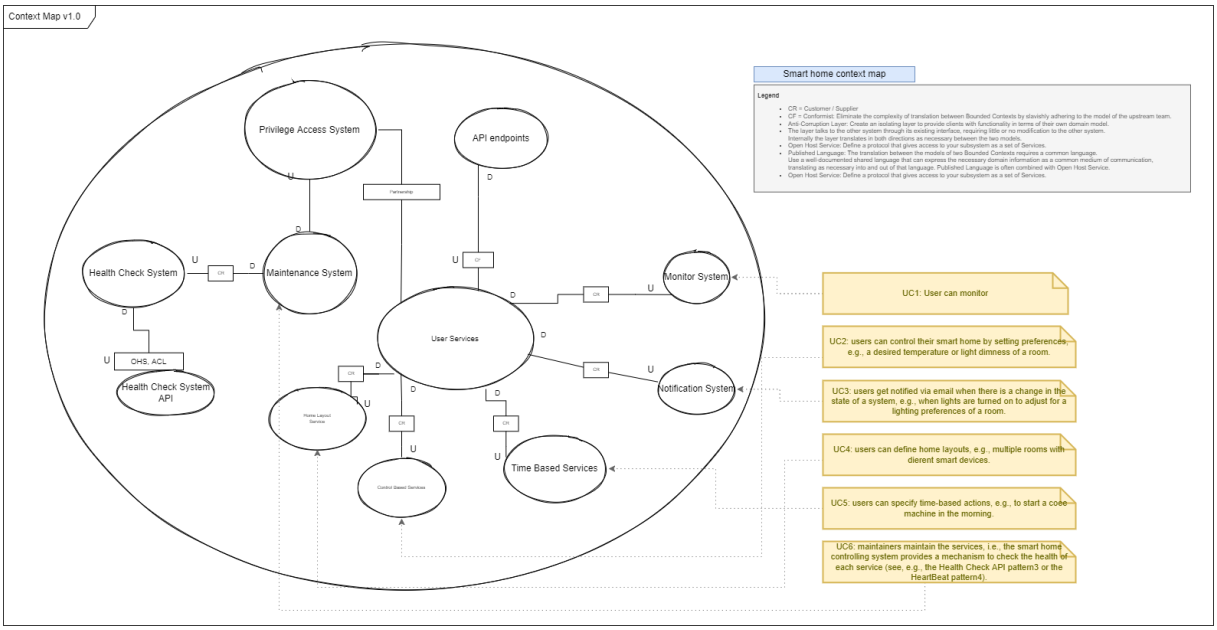


Figure 1: Context Map 1.0

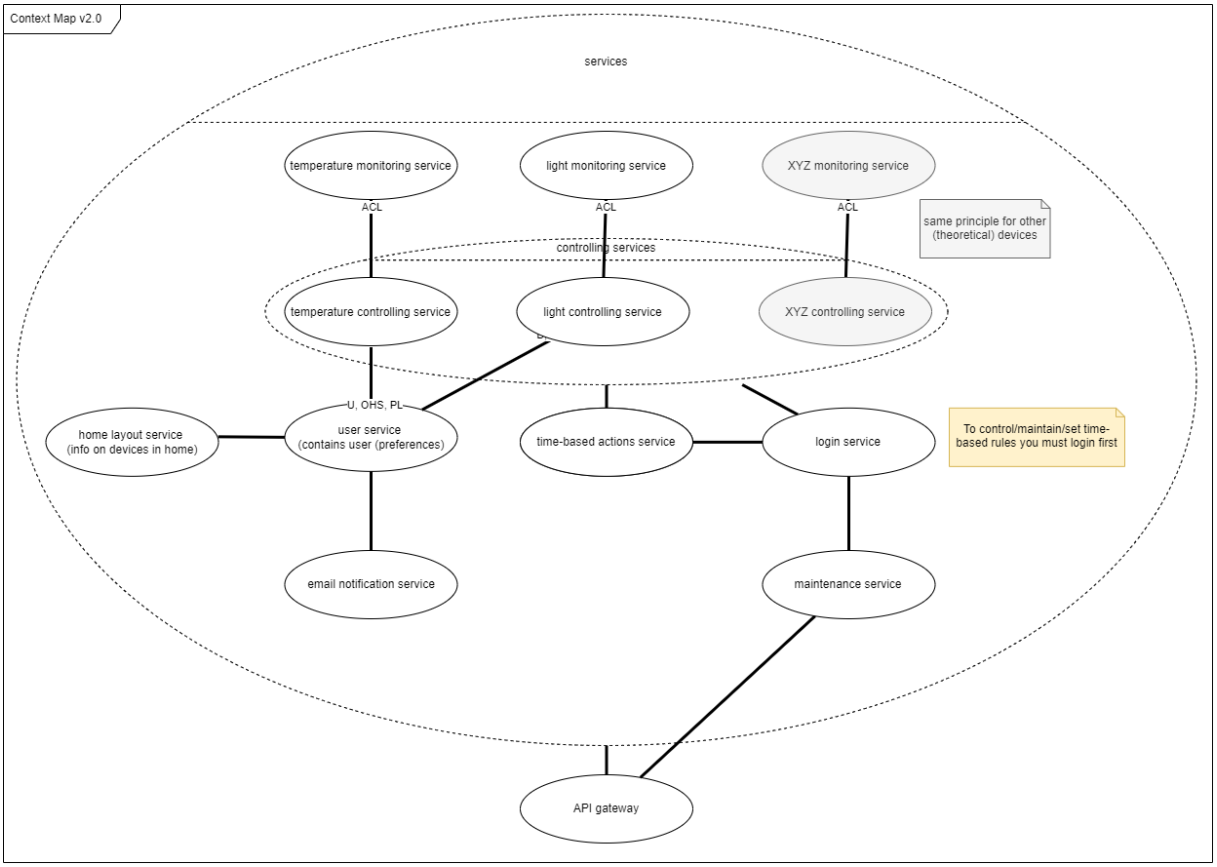


Figure 2: Context Map 2.0

1.2 Development Stack and Technology Stack

We decided to use C# as our coding language as it is being developed by Microsoft and thus it has well written documents, a large amount of resources both in their official website and spread on the internet. As for our architecture builds, Microsoft provides official documentation for many elements. In particular we we read information about the publish/subscribe event system, additional information on DDD as explained on the official docs.microsoft website, the

microservice system which is thoroughly explained using an example GitHub project about online shopping, and lastly, a few books which are offered for free, again by the official Microsoft website (we are not sponsored by Microsoft).

.NET 6.0 was used for most simple microservice examples that were created by the team, because it is the latest version of .NET and the one that holds the most up to date resources.

dotnet will most likely be the nuget providing system we are going to use as it is easy to implement external libraries in our actual system, and it also helps the developer with creating the UI part of the microservices.

Links:

- https://docs.microsoft.com/en-us/
- https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/microservices-architecture
- https://docs.microsoft.com/en-us/azure/architecture/patterns/publisher-subscriber
- https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/ddd-oriented-microservice
- https://docs.microsoft.com/en-us/microsoft-365/solutions/cloud-architecture-models?view=o365-worldwide
- https://www.youtube.com/watch?v=r8ucofiI8vY&t=2536s

1.2.1 Project Management and Development Stack

GitLab, Draw.io, Google Drive, OneDrive

1.2.2 Implementation Technology Stack

C#, Draw.io, dotnet, .NET 6.0, GitHub, Google drive to share our files and work on them at the same time, Discord for communication.

2 System Requirements

The specified system requirements have not been explicitly split up among the team members, but as we decided to go for a domain driven approach, all the system requirements are relevant and should be considered by all the team members wherever their consideration is required to design a particular facet of the system or implement it. This being said, we will ensure that every person implement their part of the system requirements. We have also not yet defined test cases, however, based on the interaction of the services/components as we have defined and discussed in our internal discussions, the required inputs and outputs can already be anticipated.

3 4+1 Views Model

3.1 Scenarios / Use Case View

In the following sections we present use case diagrams that depict the main features / scenarios covered and provided by our solution. Whenever we refer to *UC#*, then the main use cases that are required in the original assignment sheet are meant. For each of these *UC#*s we have defined a separate use case diagram, as well as for a few additional ones, we added to illustrate additional key functionality. In the use case description section following the diagrams we then describe in depth the key use cases from the use case diagrams.

For example, when we refer to use case ***3.1*** then this means the description refers to the sub-use-case 3.1 in the “main use case” ***UC3***.

To keep the balance between readable and helpful, we have decided to abstract from some very basic or obvious use cases, which don’t require further explanation.

3.1.1 Use Case Diagram(s)

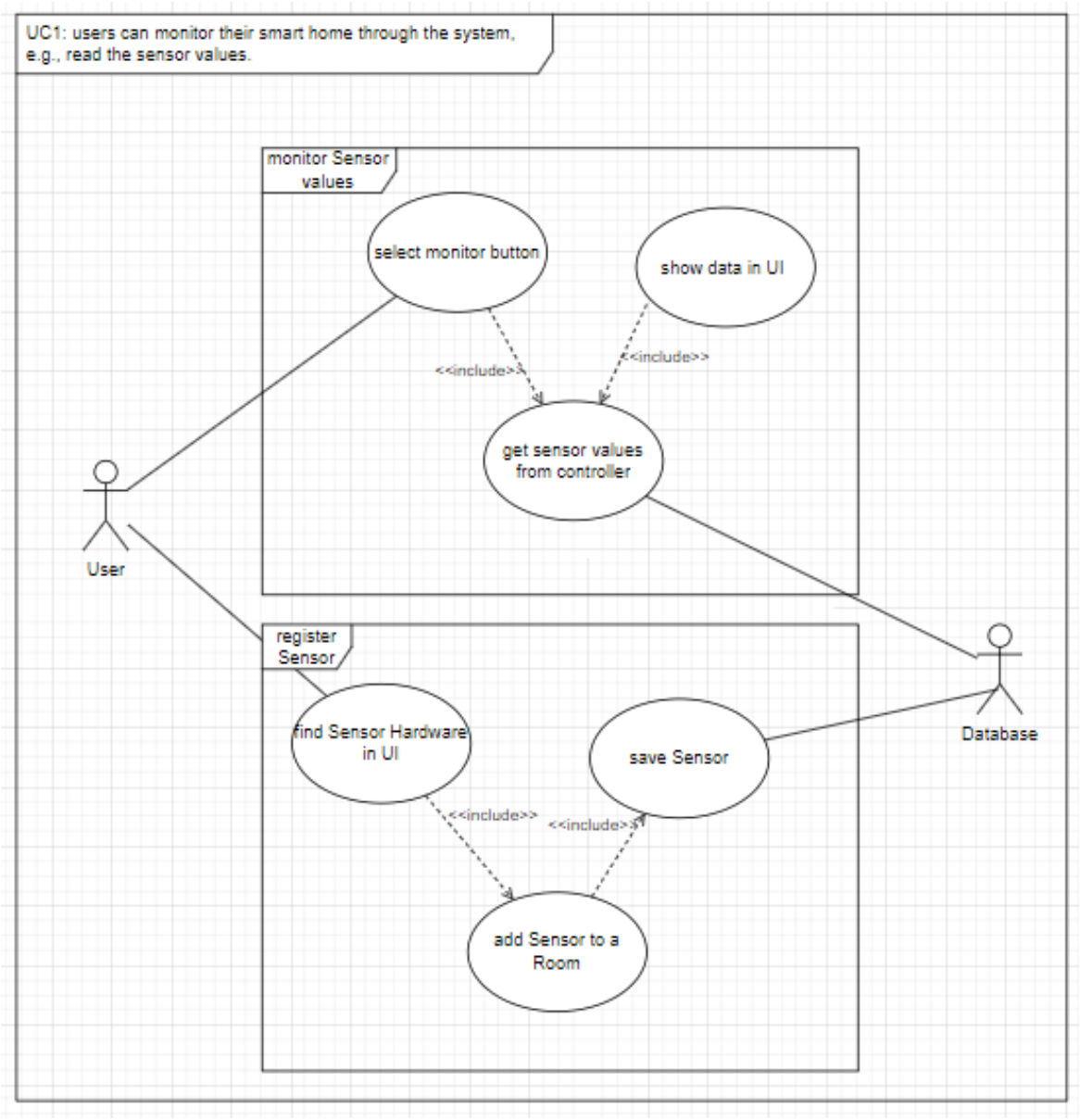


Figure 3: UC1, users can monitor their smart home through the system

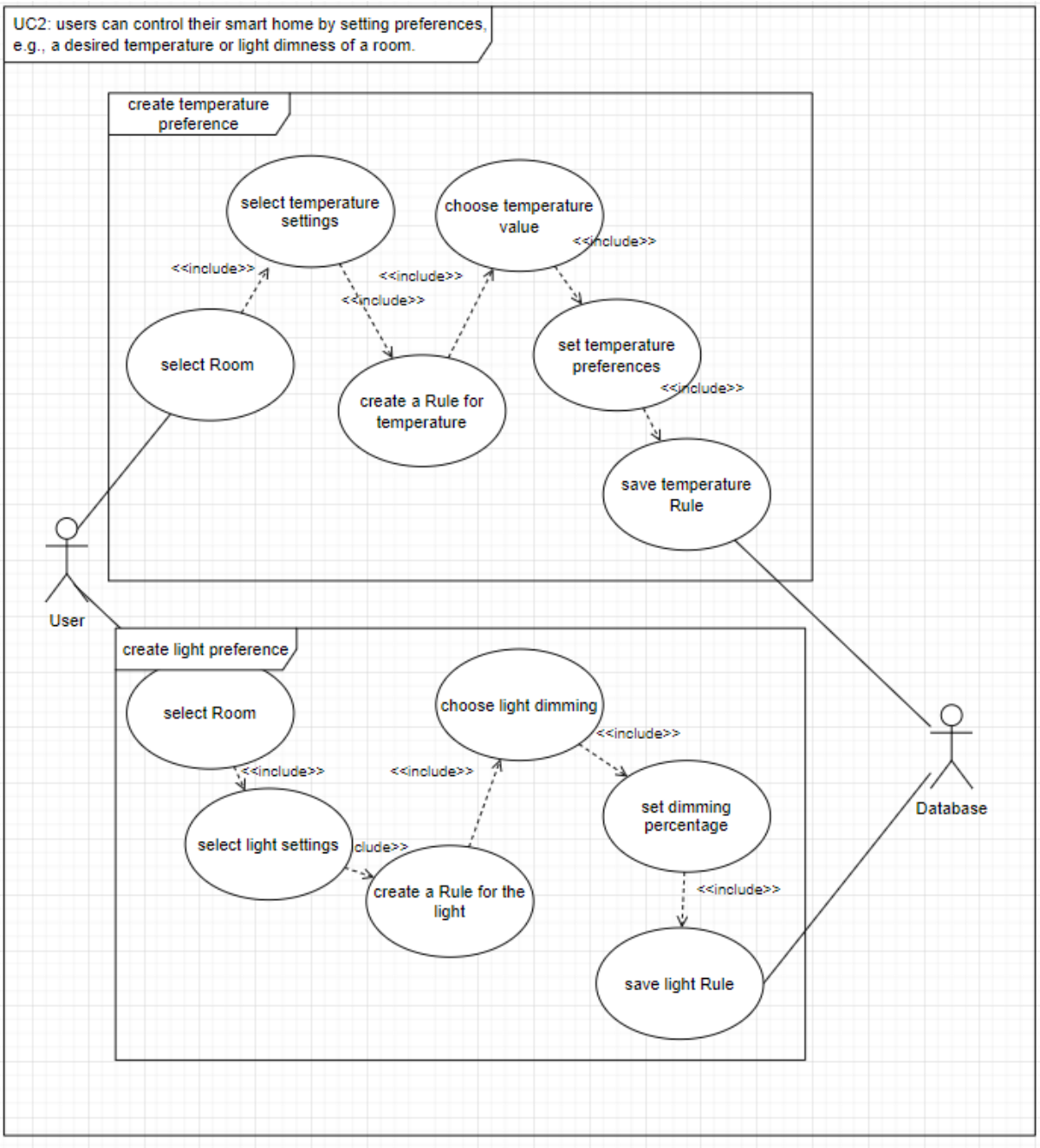


Figure 4: UC2, user can control their smart home by setting preferences

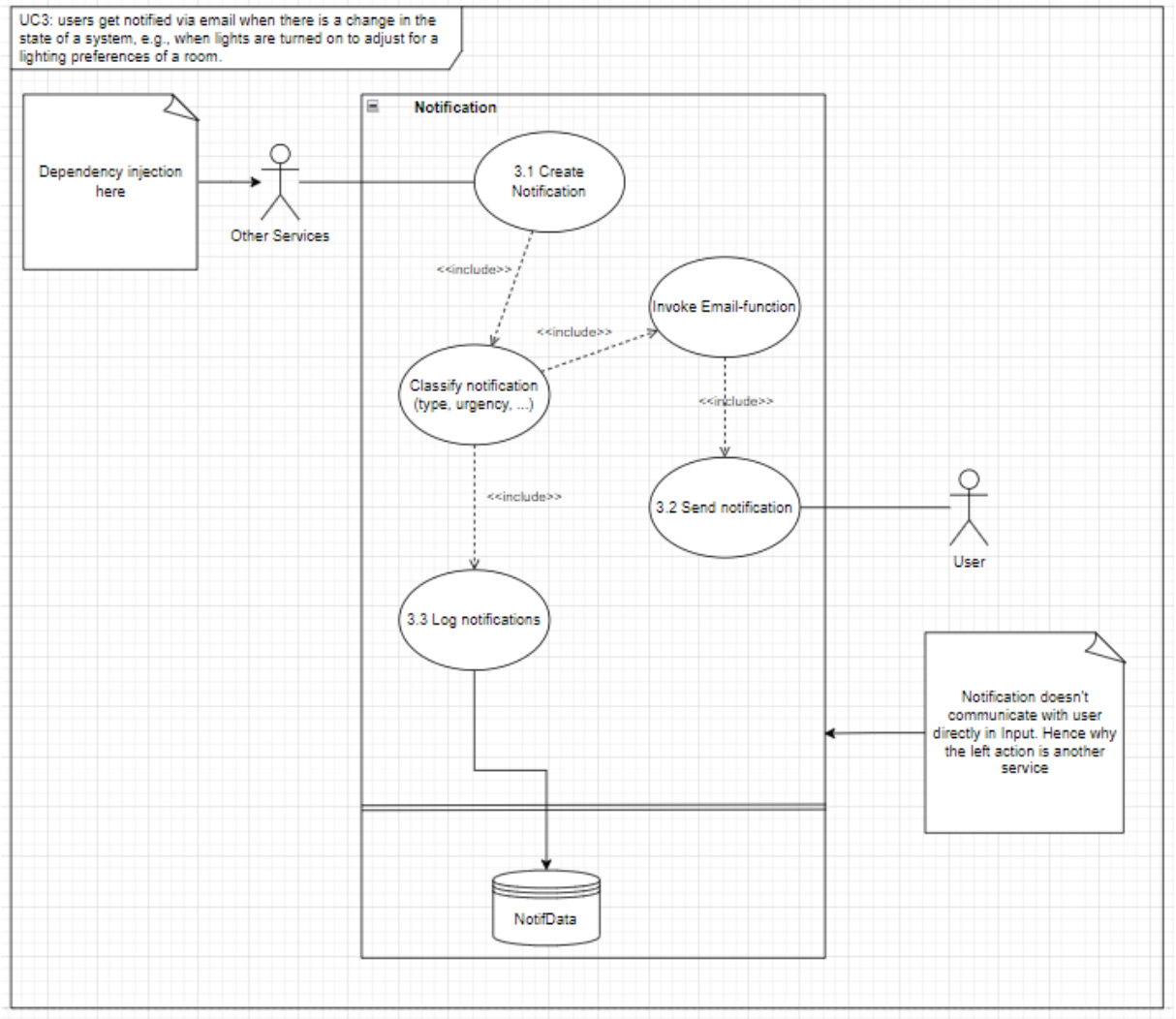


Figure 5: UC3, user get notified via email when there is a change in the state of a system

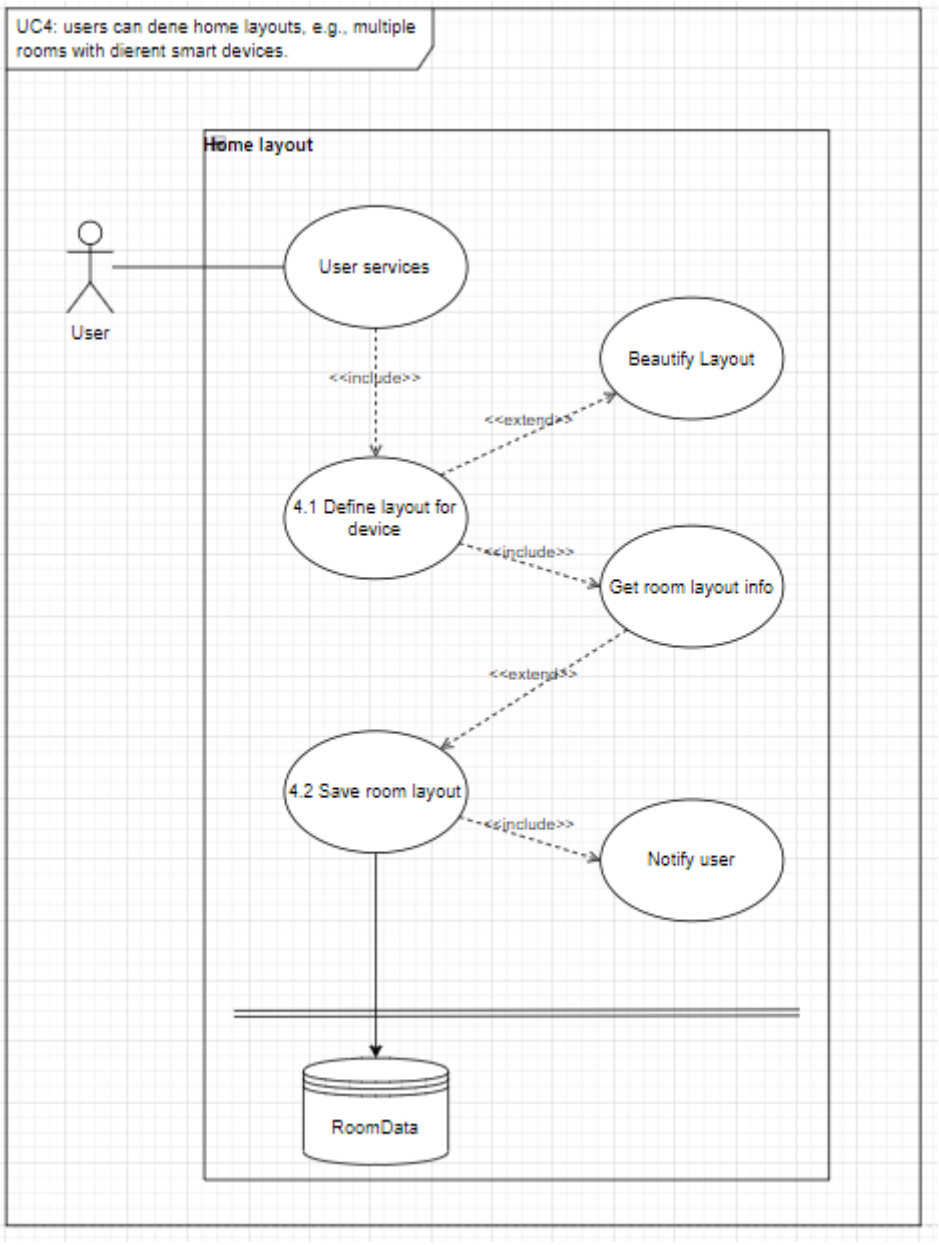


Figure 6: UC4, user can define home layouts

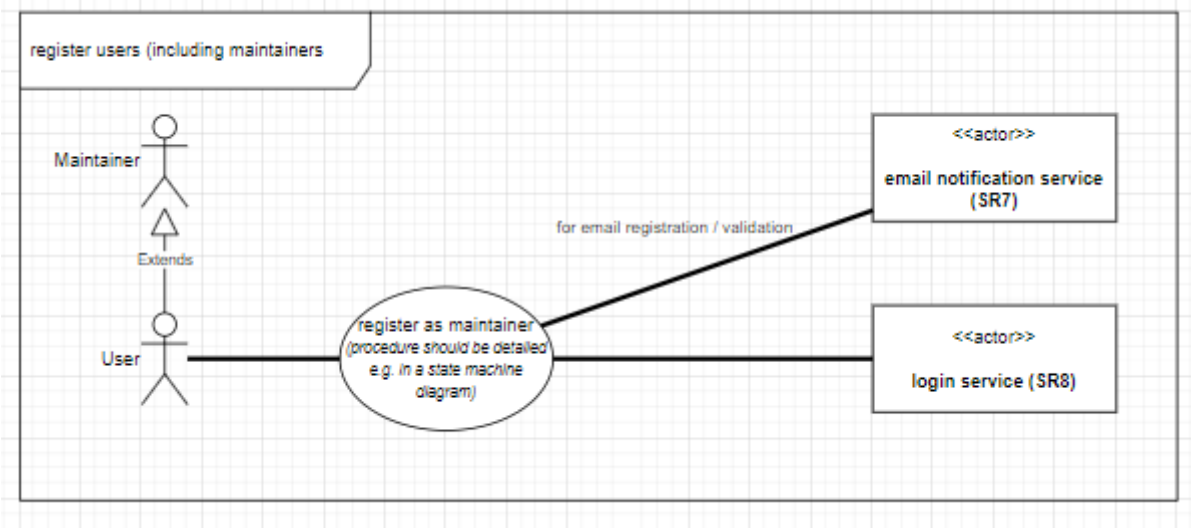


Figure 7: use case, user can register in the System

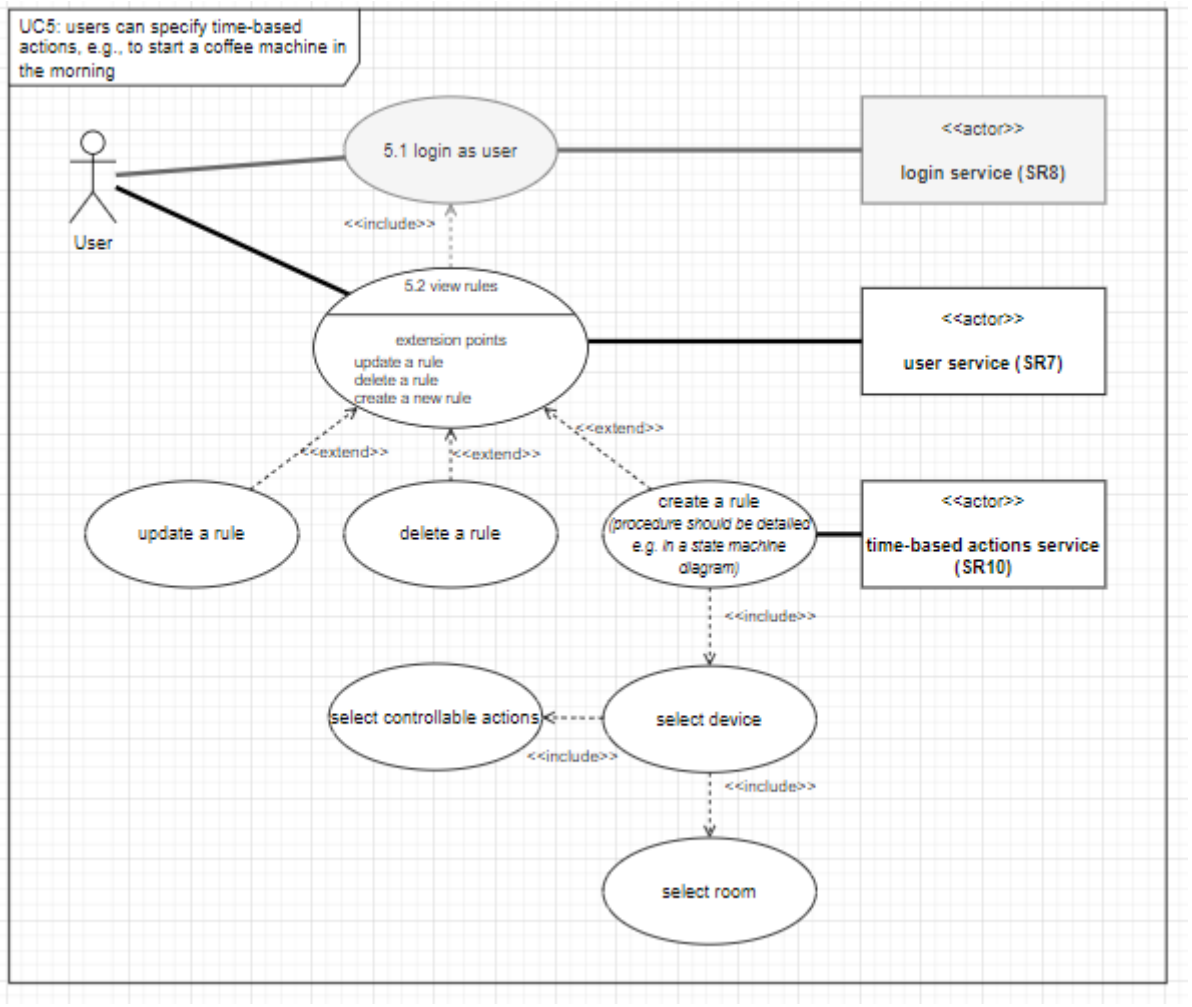


Figure 8: UC5, user can specify time-based actions

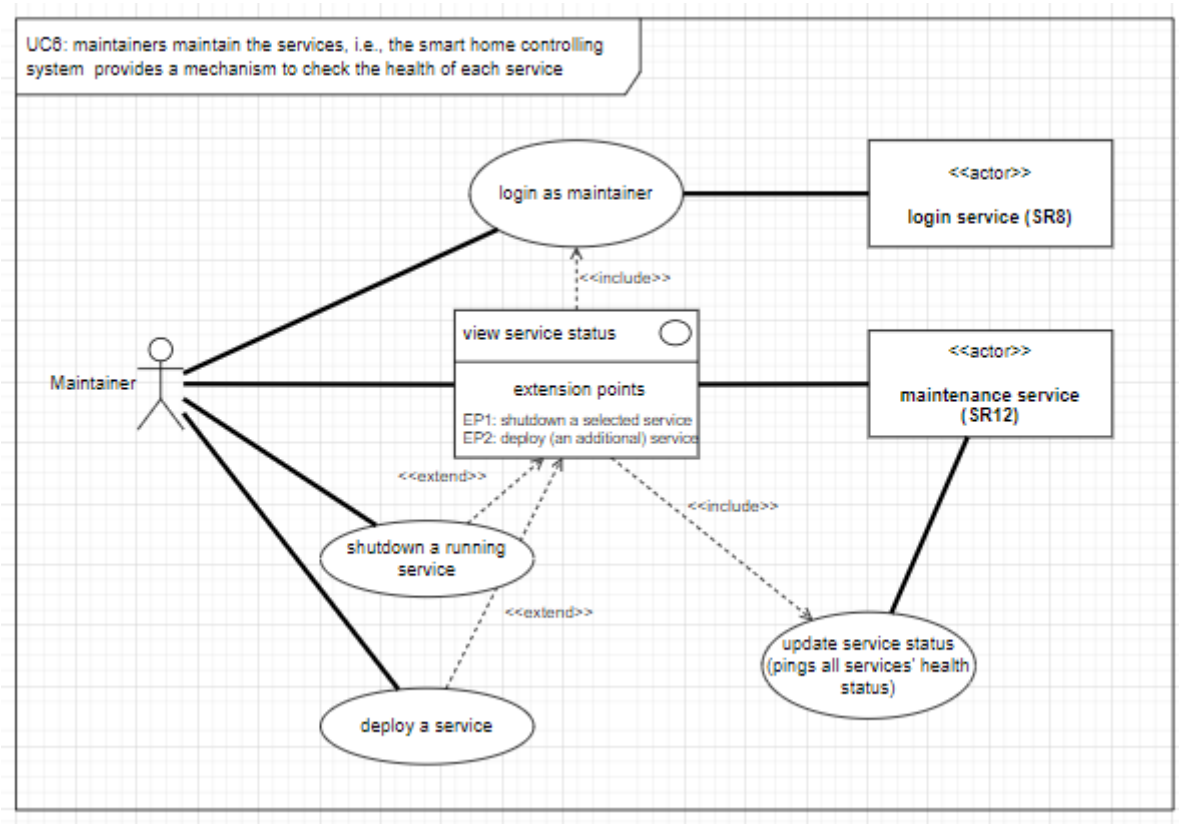


Figure 9: UC6, maintainers maintain the services

3.1.2 Use Case Descriptions

Use Case:	**User can monitor their smart home through a system**
Use Case ID:	Monitor system (UC1)
Actor(s):	User, Database
Brief Description:	User can create, save, and monitor sensor data
Pre-Conditions:	User must be logged in; Sensor must be installed
Post-Conditions:	User can successfully monitor the data
Main Success Scenario:	1. select monitor button 2. get sensor values from controller 3. show data in UI
Extensions:	
Priority:	high
Performance Target:	The System should be able to register at least 100 sensors
Issues:	Is there any limit of sensors?
Use Case:	User can create Rules and Preferences for temperature
Use Case ID:	tempControl System (UC2)
Actor(s):	User, Database
Brief Description:	User can create and save rules and preferences for temperature
Pre-Conditions:	User must be logged in; Smart Home Devices must be installed; Rooms must be declared
Post-Conditions:	User can successfully create his preferences
Main Success Scenario:	1. select room 2. select temperature settings 3. create a rule for temperature 4. choose temperature value

	5. set temperature preferences 6. save temperature rule
Extensions:	
Priority:	high
Performance Target:	The System should be able to set at least one Rule per heating element
Issues:	
Use Case:	User can create Rules and Preferences for lights
Use Case ID:	lightControl System (UC2)
Actor(s):	User, Database
Brief Description:	User can create and save rules and preferences for lights
Pre-Conditions:	User must be logged in; Smart Home Devices must be installed; Rooms must be declared
Post-Conditions:	User can successfully create his preferences
Main Success Scenario:	1. select room 2. select lights settings 3. create a rule for lights 4. choose light to dimm 5. set dimming percentage 6. save rule for light
Extensions:	
Priority:	high
Performance Target:	The System should be able to set at least one Rule per light element
Issues:	

	Create Notification
Use Case ID:	Notification 3.1
Actor(s):	Other Services
Brief Description:	Creates the notification for each user activity
Pre-Conditions:	User has to interact with the system and there needs to be an email defined
Post-Conditions:	None
Main Success Scenario:	Notification is created
Extensions:	None
Priority:	High
Performance Target:	Creation of notification
Issues:	None

	Send email notification
Use Case ID:	Email Notification 3.2
Actor(s):	Notification creation
Brief Description:	Sends the notification via email
Pre-Conditions:	None
Post-Conditions:	Make sure the email is sent
Main Success Scenario:	Email is sent to user
Extensions:	None
Priority:	High
Performance Target:	The email is sent to notify the user
Issues:	Make sure there's internet connection when the email is trying to be sent.

	Store notifications
Use Case ID:	Store notifications 3.3
Actor(s):	Created notifications
Brief Description:	Saves the notification time, which device is taking the notification in the database.
Pre-Conditions:	Notification needs to be created successfully
Post-Conditions:	None
Main Success Scenario:	Notification is created
Extensions:	None
Priority:	High
Performance Target:	Storing the notification in the database
Issues:	Database connection is done successfully

	Define layout for each device
Use Case ID:	Room Layout 4.1
Actor(s):	User
Brief Description:	Defines the room layout for each device
Pre-Conditions:	Device needs to know the home layout
Post-Conditions:	Get the new room layout information
Main Success Scenario:	Room layout is changed
Extensions:	Beautify Layout
Priority:	High
Performance Target:	Allows the user to determine the room layout
Issues:	Make sure the room is defined by 1 device.

	Save layout
Use Case ID:	Save Layout 4.2
Actor(s):	Room layout creation
Brief Description:	Saves the room layout in the database
Pre-Conditions:	Get the information about the new room layout from 4.1
Post-Conditions:	Make sure to notify the user once the layout is saved
Main Success Scenario:	Email is sent to user
Extensions:	None
Priority:	Low-Medium (I don't think this should be mandatory)
Performance Target:	Create a new layout for your house
Issues:	Make sure the layout is saved in the database.

	Login as user
Use Case ID:	Login as user 5.1
Actor(s):	User, login service
Brief Description:	the pre-condition for many user-actions is being logged in successfully. Therefore, this demonstrates exemplatorily how the login use case works.
Pre-Conditions:	None
Post-Conditions:	None
Main Success Scenario:	1. The user is per default presented with a login screen. 2. The user inserts their unique username and passphrase. 3. The user service receives the request and checks it with its repository/database. 4a. On match the login service will return a token corresponding with the user role registered and enable actions for that role.

	4b. On mismatch the login service will return a error notification to the user.
Extensions:	None
Priority:	Critical
Performance Target:	Login using a common interfacing pattern. Backend service work should not exceed usual login time (a couple of seconds)
Issues:	Wrong password, or username not registered. Token not returned successfully

	View rules
Use Case ID:	View rules 5.2
Actor(s):	User, User service, time-based action service
Brief Description:	The User can view their time-based rules in place. Extensions of the use case allows the deletion, change or creation of a new time-based rule.
Pre-Conditions:	User has to be logged in as a user.
Post-Conditions:	None
Main Success Scenario:	1. The user accesses the in-place rules from their dashboard (clicks a button). 2. The user service then provides that view by pulling the data from the time-based action service and displaying it. Extension 1 - create a new rule: 1. On the extension for creation of a rule the user needs to select one of their registered devices and select the option for creating a new time-based action. 2. They then have to choose the operation the device should perform (the allowed operations are provided by the time-based action service, which received that information from the appropriate controller service) and the time that operation should begin and optionally when it should end. 3. This information is then transmitted to the time-based action service who stores that information in its database. 4. When that start time is triggered by the time-based actions database, it then sends a request to the relevant controlling service. Optionally: 5. It sends an email notification to the user informing about the success of the operation.
Extensions:	1 - Create a rule (explicated in main success scenario) 2 - Update a rule 3 - Delete a rule
Priority:	Normal
Performance Target:	As some database operations are required this might not be the fastest of operations but this should propably be ok, as these rules have to be set just once and then they are just passively reacting to events without direct user intervention.
Issues:	Authorization for the execution of actions needs to be considered as the user might not be logged in at the time.

	Maintainer maintains service
Use Case ID:	View service status (UC6)
Actor(s):	Maintainer, Maintenance service
Brief Description:	Maintainer can view the services' status

Pre-Conditions:	Maintainer has to be logged in as a maintainer (role).
Post-Conditions:	None
Main Success Scenario:	1. The maintainer accesses the service status in the dashboard after logging in. 2. The maintenance service pings all the registered services' health API and updates the health status in the dashboard. 3. The maintainer is now informed about the service situation. Extensions include the shutting down or deployment of services. These steps are done however in a different system that is not in the scope of this smart home system.
Extensions:	1 - shutdown a running service 2 - (re)deploy a service
Priority:	Important
Performance Target:	Normal
Issues:	Service can be down

3.2 LOGICAL VIEW

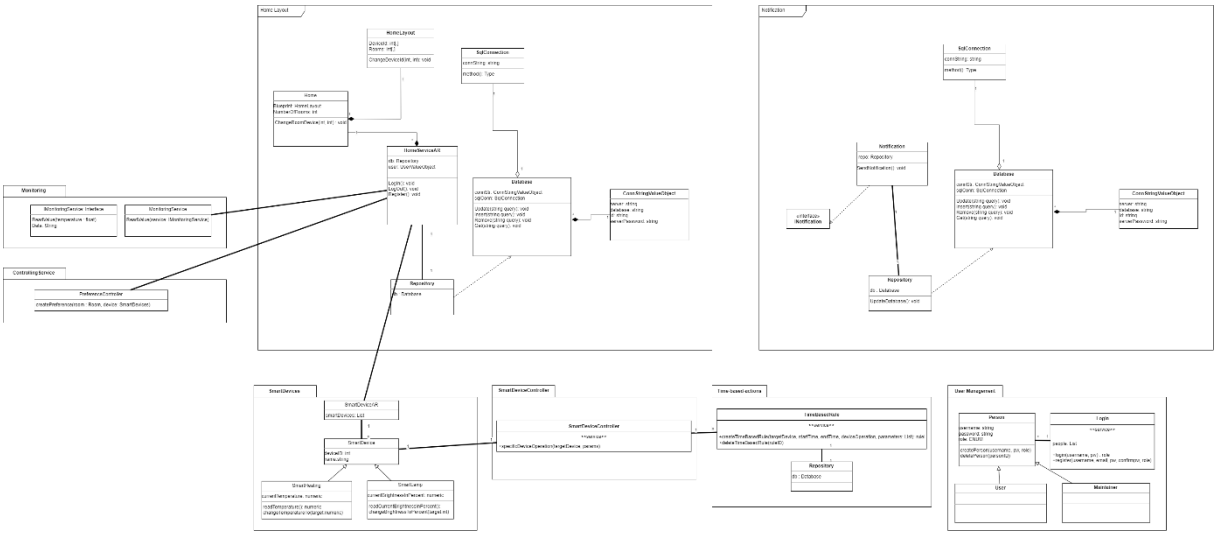


Figure 10: Class Diagram of our Smart Home Controlling System, the complete .svg file is provided in the Gitlab master branch.

3.3 PROCESS VIEW

The following sequence diagrams provide more information on the dynamic aspects of the system. They provide insight into the sequence of components that are being invoked.

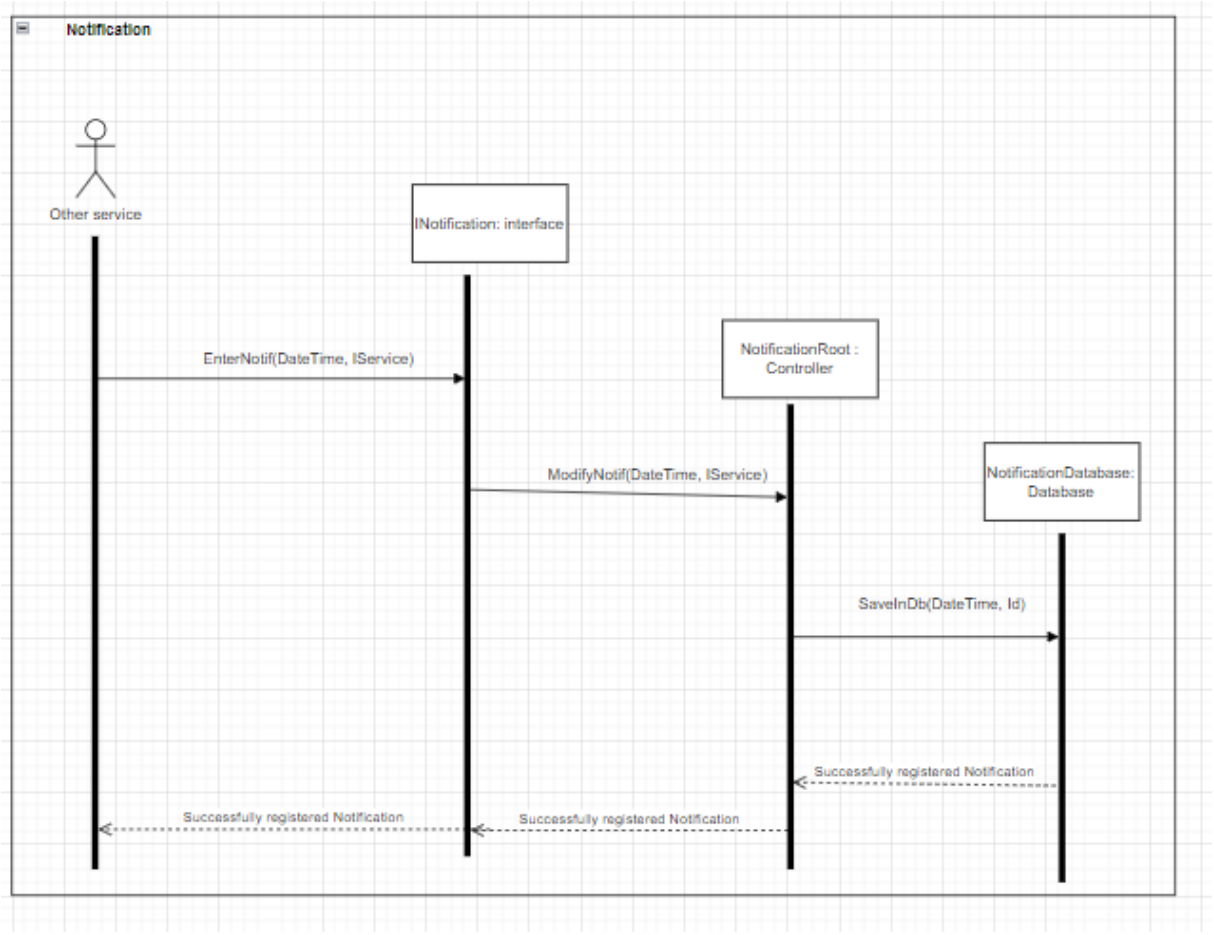


Figure 11: Notification Sequence Diagram

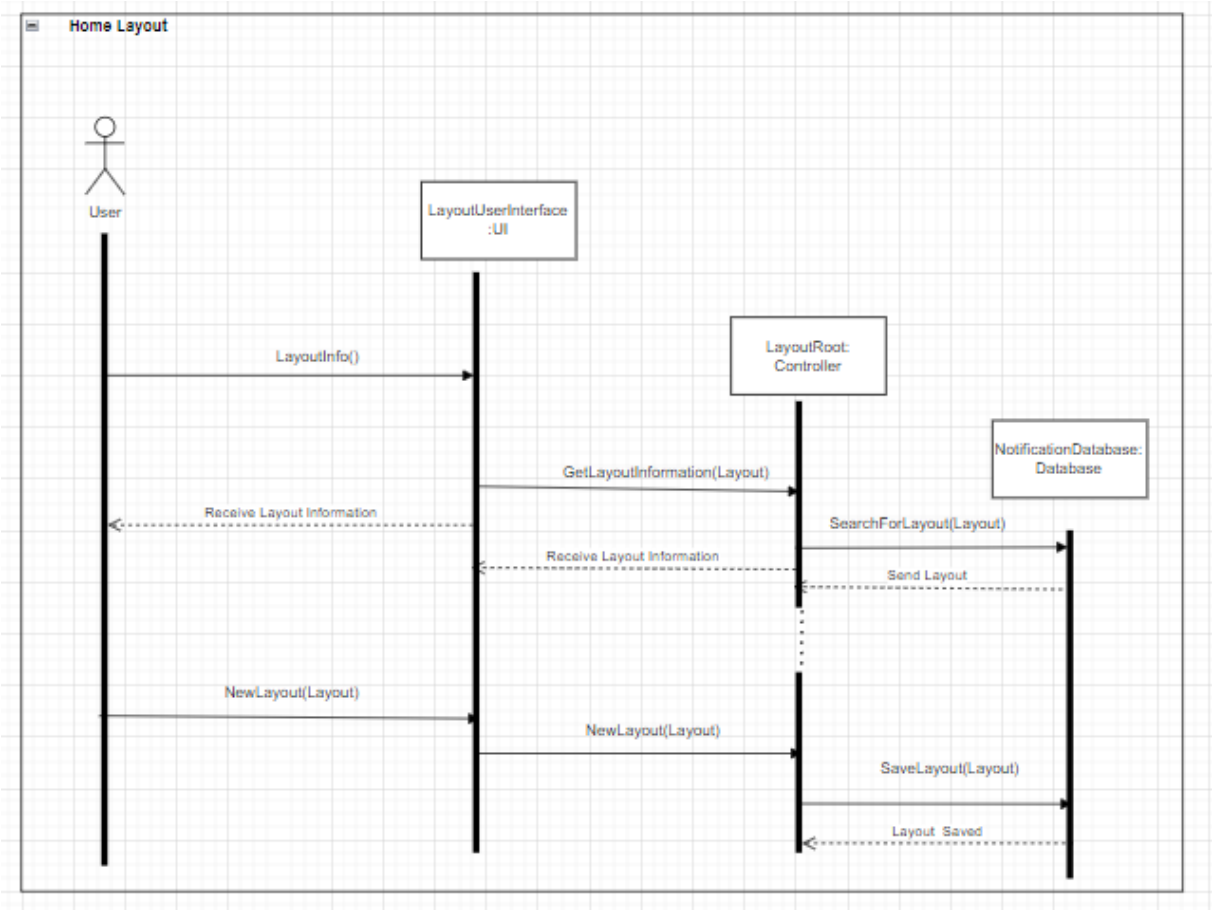


Figure 12: Home Layout Sequence Diagram

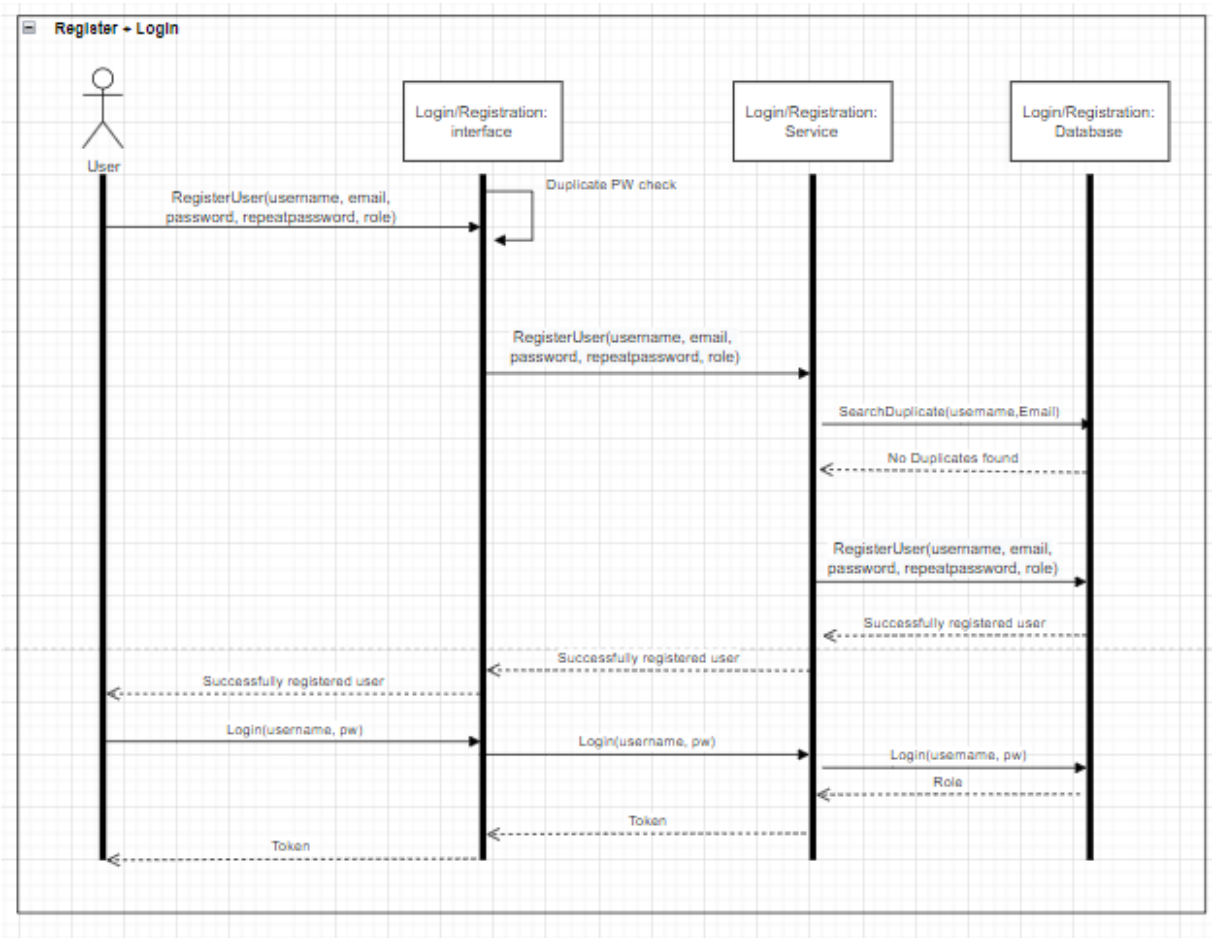


Figure 13: Register + Login Sequence Diagram

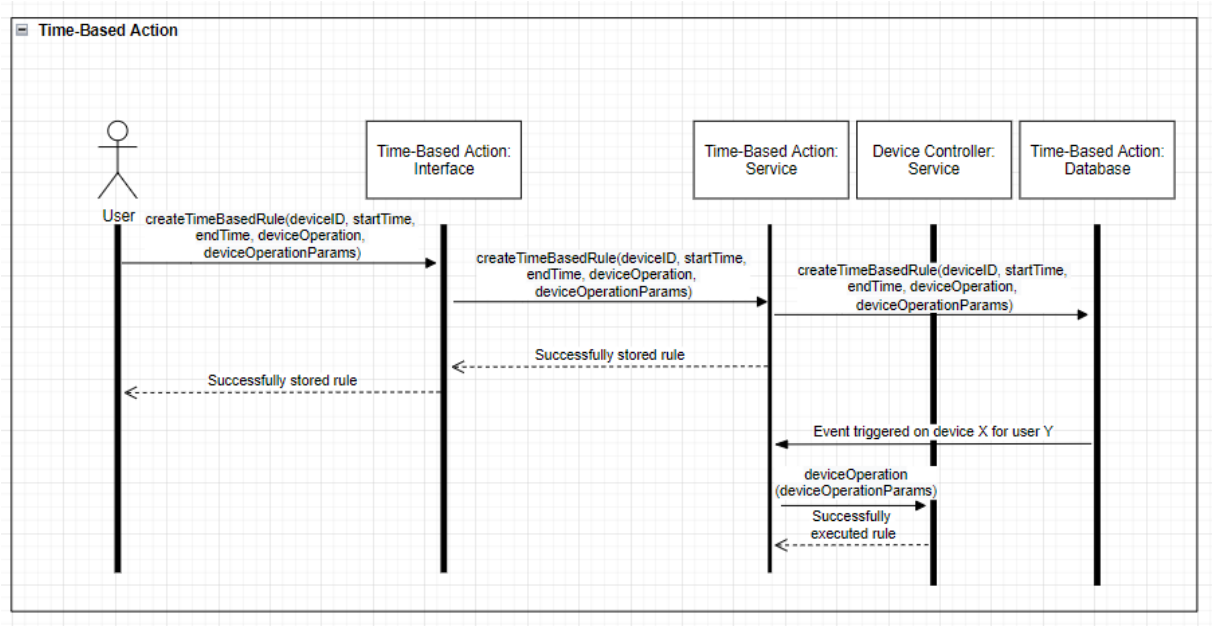


Figure 14: Time-Base-Action Sequence Diagram

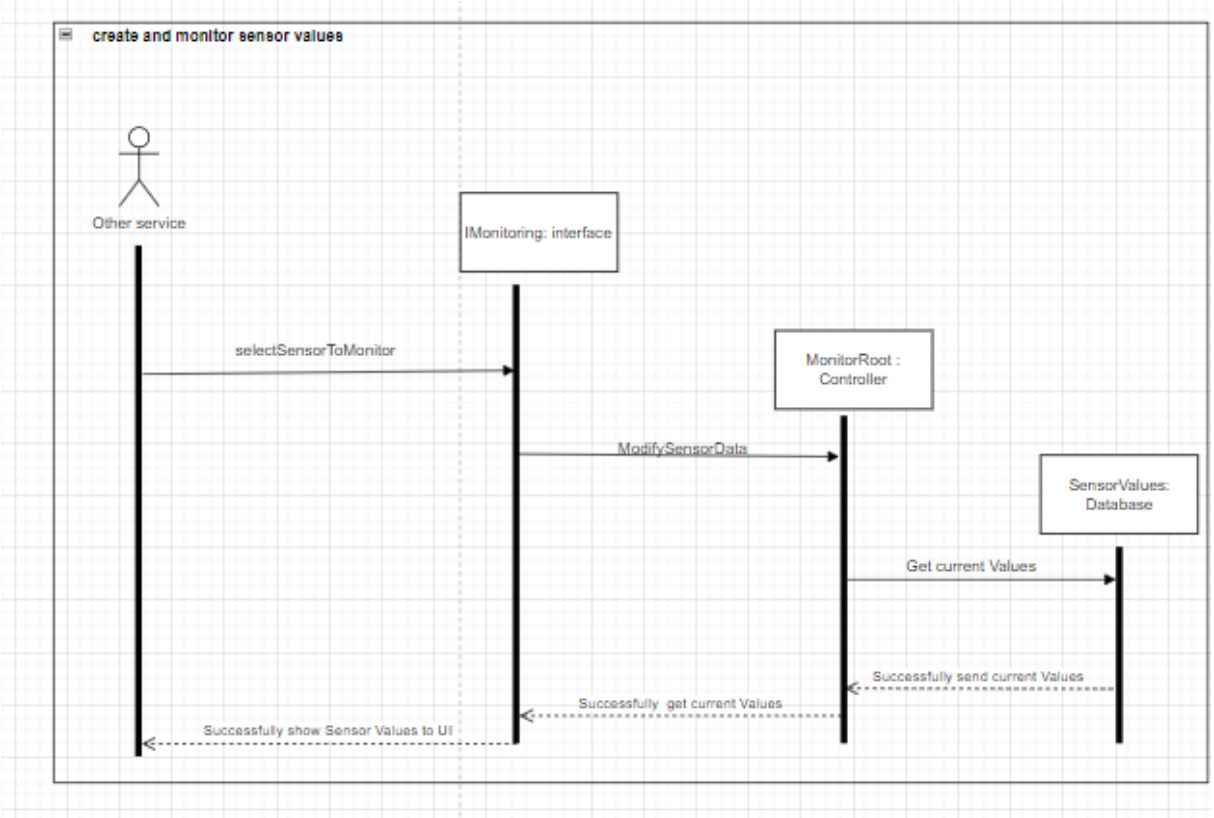


Figure 15: create and monitor sensor values Sequence Diagram

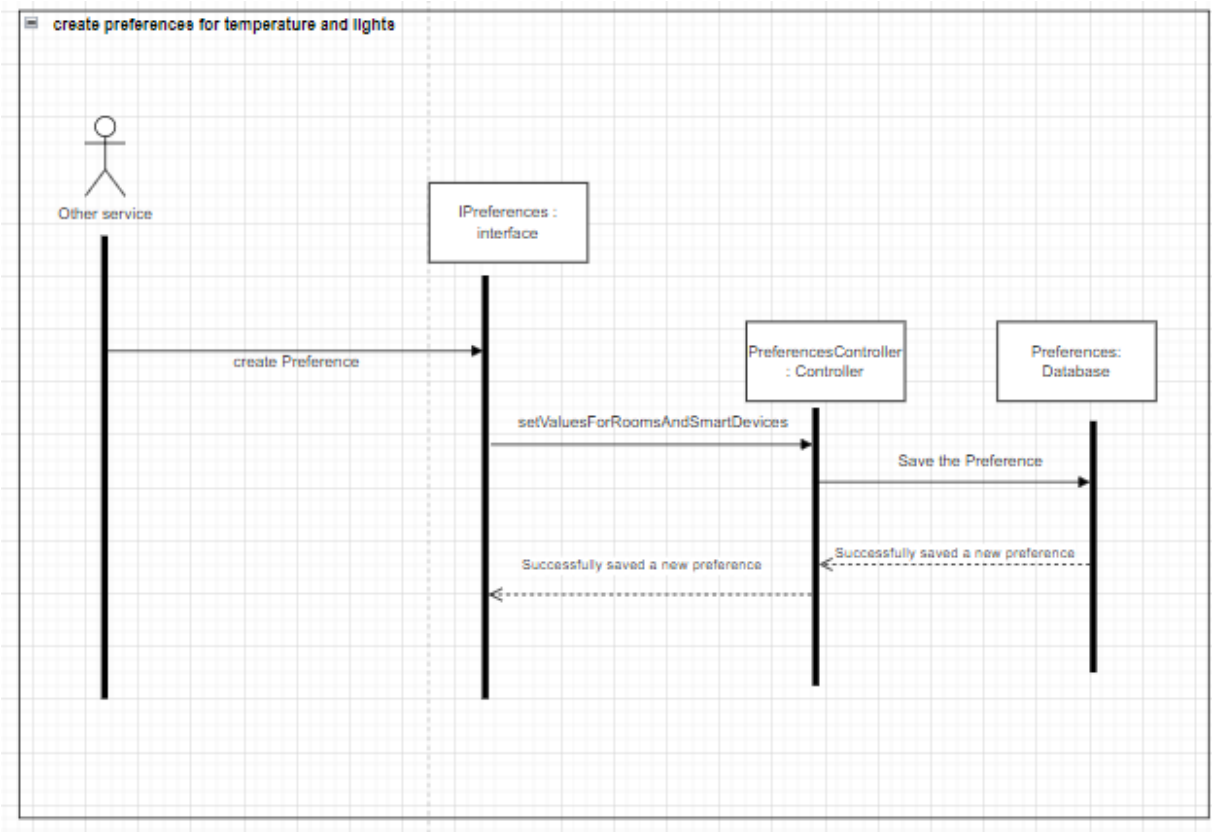


Figure 16: create preferences for temperature and lights

3.4 DEVELOPMENT VIEW

The following component diagram provides insight about the interaction between the components of the system on a technical level.

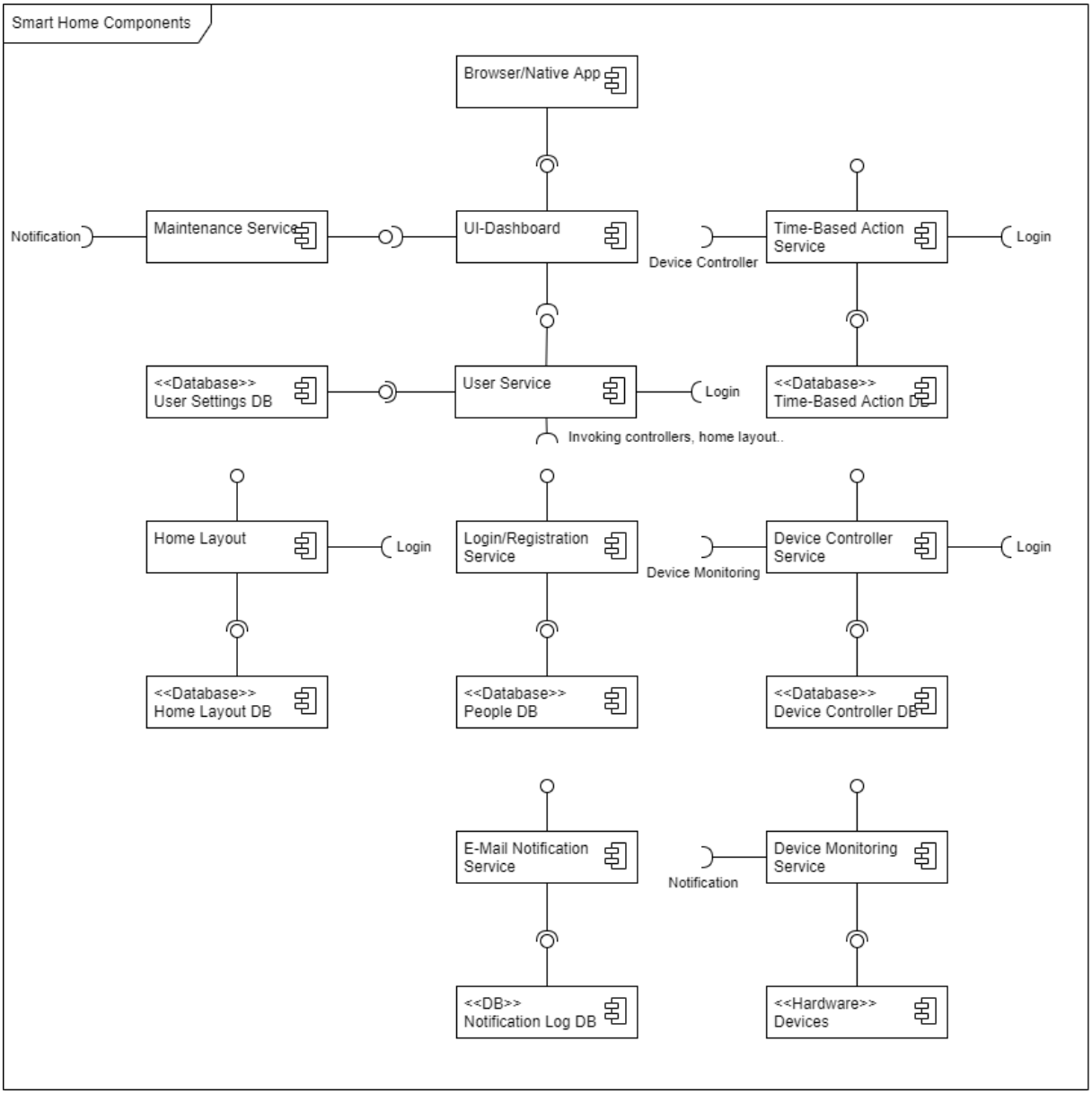


Figure 17: Components and their interaction

The central components which feed directly into the UI are the Maintenance Service and the User Service Components. They do invoke the Login service and save a token. Furthermore, the User service can invoke the components of home layouts and device controlling services or time-based actions. Device Monitoring services encapsulate hardware and are invoked by controlling services. E-mail notification service provides functionality to be invoked by other services. Finally, as this reflects a microservice architecture, the different components are served by their own databases which retain information on specific things e.g., stored user preferences or the allowed device operations for a specific device.

3.5 PHYSICAL VIEW

The Physical View includes the architectural design of how our system will be connected in the real world. We have decided to follow a simple approach for the longevity of our system, and how it will run.

First, the application is composed out of microservices, each providing different services, such as:

- User services (monitoring system, home layout service, logging in, time-based service, notification system etc)
- UI which the user will interact with (simple website, potentially built using ASPNET)
- Database system which again will most likely be SQL

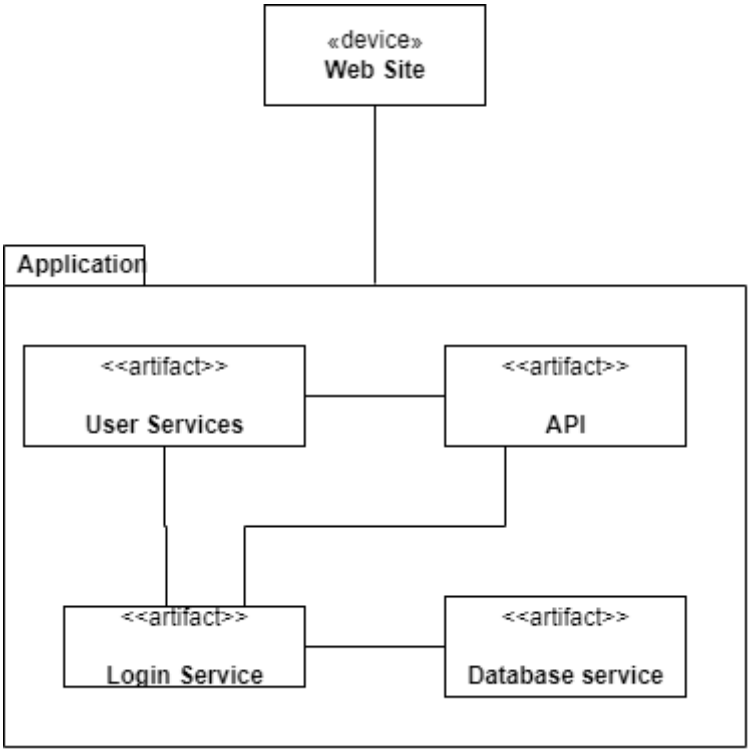


Figure 18: Physical view of the system

4 TEAM CONTRIBUTION AND CONTINUOUS DEVELOPMENT METHOD

4.1 PROJECT TASKS AND SCHEDULE

As a comparatively compact design and development team of three people, we decided to go for a flexible project structure and team coordination approach. We organized our communication using a Discord server and utilized shared Google Drive integrated draw.io to design our models and have collaborative discussions on the design decisions. The report was written using Microsoft Word with its OneDrive share functionality for simultaneous writing and editing.

As we were tasked to approach the project using the 4+1 architectural design tools, we followed the process accordingly. First, we split the 6 main use-cases as specified in the assignment sheet into “2-2-2”, meaning each of us was tasked with the main responsibility for two use-cases. The first step here was to design the use-case diagrams that represented the vision

each of us had for our use cases, and then in the logical view to imagine the required structural basis for the realization of such defined use cases. An additional tool we utilized in imagining the scenario view were sketches or “wireframes” of the user interaction to discover the required elements and components. In the second step we had to discuss the proposed ideas and had to integrate them especially on the logical level. We noticed that the very same starting point (info from the assignment sheet) had led to some different imaginations of how the processes would work. Examples for this are the conception of the home layout, or the interaction between devices and rooms. To synchronize these ideas, we held discussions on the benefits and drawbacks of the different proposals and consequently developed a common vision.

After having gained structural and conceptual clarity we then approached the remaining 4+1 views – process, development, and deployment – more efficiently and built on a common base.

Before writing this summarizing report we also attempted various options of making meeting notes. We, however, found that it was a rather time-consuming task with limited benefit, as the project management side of it was rather transparent to us due to the compact team size. We found the complexity to be rather of a technical or conceptual nature than of a project management one.

Figure 19: project schedule towards the “Design” submission

4.2 Continuous Integration, Delivery and Deployment Plan

The continuous integration, delivery and deployment for our project works by being based on the common understanding of the domain and the defined interfaces of the different parts. This allows continuously delivering and incrementally updating the system as its implementation is being completed. The defined 4+1 models provide the ubiquitous language to the team and ensure a consistent implementation.

4.3 Distribution of Work and Efforts

4.3.1 Contribution of Member 1:

Contributed on all the views for the 4 + 1 model mainly on Home layout service, Notification Service and Privilege Access Service. Developed the initial context map for an idea of how the system should function as a whole. Special responsibility for UC3 and UC4.

Home layout Service

Designed the home layout system as below

For the home layout design I decided to create a matrix like home access class. - Each room will have a place in the matrix of the entire house, for example

0 0 1 1 2 2 2

0 0 1 1 0 2 2

3 3 1 1 0 2 2

0 -> Represents empty spaces (walls) which aren't part of the house The rest of the numbers represent rooms in the house - There will be a second matrix which handles devices controlling rooms. For example

0 0 1 1 2 2 2

0 0 1 1 0 2 2

1 1 1 1 0 2 2

Each 0 again represents no devices - Each other number represents the device ID that is controlling room. This design allows us to create a map like interface for the user of the smart device and easy interactions with the design. Each room can have their own devices which will have their own ID and can be accessed from the device that is controlling the room.

Notification Service

Defined the interfaces for notifications and the notification creation. A notification will be created for every single action that happens within the system and each notification will be saved in the database.
Even if the system is restarted, the notifications will be safe inside the database and the system can progress where it left off by checking the database for any ongoing notification.

Privilege Access System

Concepted the log in system for the system. Users can login using their email/username and password. After logging in the user gets their privilege which is either a maintainer or a normal user. The maintainer gets access to the Health Check System, and the normal user gets access to the User Services such as Home layout service, Monitoring system etc.

4.3.2 Contribution of Member 2:

Contributed to all views for the 4+1 model. Special responsibilities were the time-based action service and the maintenance service and related use cases (UC5 and UC6). Created the updated context map v2.0.

Initially the time-based action service was quite complex and quite closely coupled with some of the other services. In later iterations it was simplified so we can have a more pure microservice architecture without fewer links towards other services and therefore fewer risks for the successful operation of the service.

Designing the maintenance service (especially the structural view) was quite challenging at the start, as we all had different ideas on what exactly the role of the maintainer would be. Is it more of remote server admin role that just redeploys updated versions of the service or is it a person that installs the system on the user's premises and maybe even a kind of power-user that maintains their own system. In the end we decided to abstract the maintainer and just integrate them with the login service and the dashboard, where its role is mainly just checking the service status. As we are a small team we did decide to not add additional features to the scope of the system on this end and therefore shutting down or deploying a service is left to its own system and not integrated here.

4.3.3 Contribution of Member 3:

Contributed on all the views for the 4+1 model, mainly on the creation, monitoring, and preferences Use Cases. Special responsibility for UC1 and UC2.

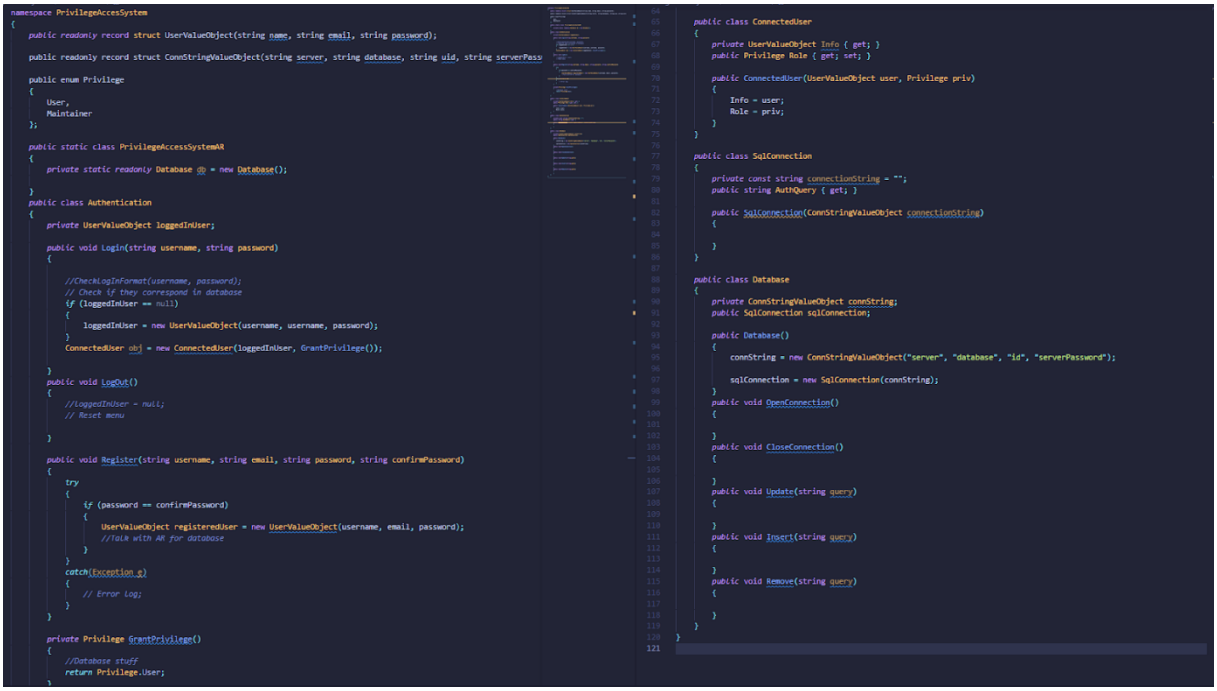
To begin with my journey, I started with a brainstorm session. I thought about what smart home system includes, especially what devices can be included. After a quick overview I started

writing use cases to the requirements about how I monitor sensor values. In conclusion we decided to use more microservices and to encapsulate the logic to monitor sensor values.

After the monitoring part I thought about creating and setting preferences for smart devices. We will create rules and save them up in the Database by using interfaces and microservices to encapsulate our logic. It ends up being easier to lay the focus on microservices than on difficult relationships, because of the overall complexity of the system.

5 HOW-TO / MOCK-UP DOCUMENTATION

Privilege access system mockup:



```
namespace PrivilegeAccesSystem
{
    public readonly record struct UserValueObject(string name, string email, string password);

    public readonly record struct ConnStringValueObject(string server, string database, string uid, string serverPass

    public enum Privilege
    {
        User,
        Maintainer
    };

    public static class PrivilegeAccessSystemAR
    {
        private static readonly Database db = new Database();

    }
    public class Authentication
    {
        private UserValueObject loggedInUser;

        public void Login(string username, string password)
        {

            //CheckLogInFormat(username, password);
            // Check if they correspond in database
            if (loggedInUser == null)
            {
                loggedInUser = new UserValueObject(username, username, password);
            }
            ConnectedUser obj = new ConnectedUser(loggedInUser, GrantPrivilege());

        }
        public void LogOut()
        {
            //LoggedInUser = null;
            // Reset menu

        }

        public void Register(string username, string email, string password, string confirmPassword)
        {
            try
            {
                if (password == confirmPassword)
                {
                    UserValueObject registeredUser = new UserValueObject(username, email, password);
                    //Talk with AR for database
                }
            }
            catch(Exception e)
            {
                // Error Log;
            }
        }

        private Privilege GrantPrivilege()
        {
            //Database stuff
            return Privilege.User;
        }
    }

    public class ConnectedUser
    {
        private UserValueObject Info { get; }
        public Privilege Role { get; set; }

        public ConnectedUser(UserValueObject user, Privilege priv)
        {
            Info = user;
            Role = priv;
        }
    }

    public class SqlConnection
    {
        private const string connectionString = "";
        public string AuthQuery { get; }

        public SqlConnection(ConnStringValueObject connectionString)
        {


        }
    }

    public class Database
    {
        private ConnStringValueObject connString;
        public SqlConnection sqlConnection;

        public Database()
        {
            connString = new ConnStringValueObject("server", "database", "id", "serverPassword");

            sqlConnection = new SqlConnection(connString);
        }
        public void OpenConnection()
        {

        }
        public void CloseConnection()
        {

        }
        public void Update(string query)
        {

        }
        public void Insert(string query)
        {

        }
        public void Remove(string query)
        {

        }
    }
}
```

Figure 20: Code template example of our logical structure

The mockup was made to test the authentication system without any tokens, just a simple username/email password authentication. All team members worked on it, taking into consideration how this system would work as a microservice, the API Endpoints, the database implementation and how the code should have been organized.

The expected output of this mockup was the authentication of a user when the right username and password were given and error checking when the initials where incorrect.