Project Report

Project report

Green smart home solution using a sensors management system

Student: Nawras Mouala

The Faculty of Engineering
The Maersk Mc-Kinney Moller Institute
The University of Southern Denmark
Campusvej 55, 5230 Odense M

Contents

List of Figures

List of Tables

Summary

In Smart home management systems ,a set of IOT devices are combined to deliver a remotely controlled user-based behavior. The process of managing this behavior called 'smart home automation'. The primary purpose of smart home automation is to save both time and money. 'G.S.H' is a green smart home management system developed in collaboration with NTT Data to extend smart homes with integration with green energy sources. The Project is built on the top of two main concepts: Green energy and self-organized automation.

The development process followed the agile unified process methodology in order to provide iterative and disciplined development. Sprints were made to define tasks for each iteration using Teams "Road map" tool. Stakeholders were defined using stakeholders registry and analysed using a Power-Interest grid to understand their impacts on the project. Stakeholders' needs were defined followed by an analysis to define and limit the objectives of the project. A set of functional and non-functional Requirements were specified and prioritized in priority tables then captured in a usecase diagram in order to understand users interaction with the system. A deep analysing on the requirements and usecases is done in order to provide analysis elements. Analysis elements where modeled using the unified modeling language(UML). Design decisions were made based on the models provided for the analysis elements. Based on the modeled analysis elements a set of design decisions were gathered and modeled using the unified modeling language to understand the system structure and behavior in order to start the implementation. By using Docker, the software system is implemented as three functionally separate microservices. The services represented briefly as follows:

- "Automation service" has the responsibility for managing automation processes and communicating physical devices and implemented as a component-based server using Spring framework in java programming language. Automation service represents the main service in the system and exposes API endpoints enable other applications to interact with its resources.
- "Sensors service" has the responsibility for gathering data from sensors and implemented as a single java module. Sensors and Automation services communicate via distributed data partitions using Hazelcast.
- "Web service" has the responsibility for managing the interaction with the users and implemented in c# programming language as a MVC asp.NET web application. Web and Automation services communicate via Automation server API endpoints.

Through the implementation process, verification tools were applied to ensure that the design decisions made at the design level meet the specified requirements. Those tools include unit test, integration tests and Model checking simulation targets the main logical part of the system which responsible on managing the automation process.

1 Introduction

1.1 Background

This report is a part of the seventh term in software technology study program at the university of southern Denmark. The report documents the development process of a Minimum viable product of software system for a green smart home solution in collaboration with *NTT Data*, information technology (IT) service and consulting company.

NTT Data is part of the NTT group. The company's main focuses are consulting, industry solutions, business process services, IT modernization, and managed services. The company provides several services to their clients, including Cloud solutions, Cybersecurity, Data & Intelligence, Salesforce, SAP, ServiceNow, Application Development & Management, and delivers end-to-end solutions for various industries.

NTT Data, following a previous collaboration as an intern during the sixth term of software technology study program, has offered to work on creating and setting up a green smart home solution.

The idea is to create a sustainable solution, where green energy suppliers will be integrated with a smart home management software system.

The report will take in consideration the software part of the solution.

1.2 Problem definition & Motivation

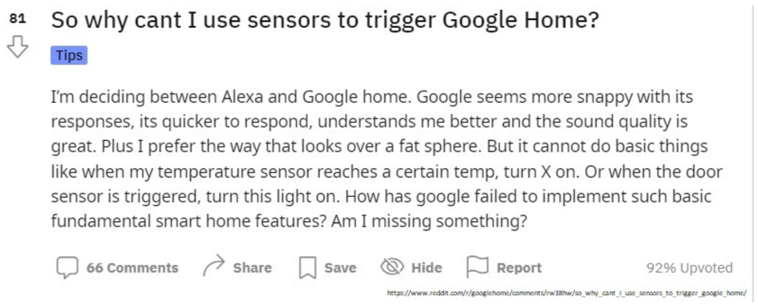
The world is moving into the new era of industrialization where users are seen as key figures in driving the further evolution of digitization. IOT becomes the technology that should be available to everyone[2].

As one of the world's leading companies in the field of digitization, *NTT Data* wants to build a an IOT solution that aligns its goal of making the software of the future sustainable, easy, intuitive, and accessible from anywhere in the world.

The proposed solution achieves a new relationship between end-users and IOT, making it possible for a non-IT person to create an IOT flow, by connecting smart devices of his choice and managing automation processes.

Smart home software developers predetermine the behavior of an smart device based on sensors of their choice. An example is the Google Home Assistance (Home & Away Routines)[3]. In this example, the application is provided with presets that make the lights or other smart devices automatically turn off when the user leaves home. In other words, the user is forced to adhere to sensors of developers choice to control smart devices.

Following is a real Life example where a user needs to turn on a device of his choice based on a sensor of his choice.


So why cant I use sensors to trigger Google Home?

I'm deciding between Alexa and Google home. Google seems more snappy with its responses, its quicker to respond, understands me better and the sound quality is great. Plus I prefer the way that looks over a fat sphere. But it cannot do basic things like when my temperature sensor reaches a certain temp, turn X on. Or when the door sensor is triggered, turn this light on. How has google failed to implement such basic fundamental smart home features? Am I missing something?

https://www.reddit.com/r/googlehome/comments/rw38hw/so_why_cant_i_use_sensors_to_trigger_google_home/

Figure 1: An example from reddit.com [1]

The self-organized automation concept is delivered by Giving the end-user the ability to determine the performance of a smart device of his choice based on a sensor's values for a sensor of his choice provides a better user experience. It also make dealing with smart devices and sensors easier and more flexible. An example is to stop a device that cannot work with a certain conditions when a sensor measures the conditions values. A concrete example is to make a device cannot work in some humidity conditions to stop when humidity sensor registers the condition's values. In this way, the user will have more options to deal with any smart device based on any sensor's values.

A second issue is that smart home applications do not provide a software environment that support integrating with green energy sources.This integration opens up many features that contribute to energy and money savings. For example, a software system can communicate with a weather API and calculate the average wind speed for the next seven days. It will then provide an estimate of the amount of green energy will be generated by the green energy source. It will also determine the consumption time for each device connected to the system.

By integrating the concept of green energy and self-organized automation, the proposed solution targets the second target in Global goal 12 [4] entitled "sustainable management and use of natural resources". It strive towards a more sustainable environment and contributes to reducing carbon emissions.

The proposed solution does not promise on a ready integration with the green energy sources from the first iteration. Alternatively, an iterative process will be applied step by step to achieve this goal.

1.3 Stakeholders

A stakeholder can be a person, group, or company that has an impact or is affected by the project.

There are three main stakeholders in this project: the project owner *NTT Data*, the non-It user and the university of southern Denmark represented by the project supervisor.

NTT Data and the project supervisor are in Key players role, as they have the highest impact on the project and can make decisions change its output. The non-it user is on Context-setter role. Project supervisor and *NTT Data*decisions aims to keep the non-it user satisfied.

1.4 Objectives

After a discussion with the stakeholders, they decided to divide the work on the project into two stages. The first stage will take in consideration the smart home management software system as a minimum viable product with minimal settings for the green energy sources. The second stage starts with building on the first stage's outputs to expand the green energy sources' software-components and setting up the physical parts. Based on that, a set of objectives have been defined as follows:

- Within the end of the first stage, a minimum viable product software system should be built to satisfy the following needs:
  - Gather data from sensors.
  - Do automation processes based on user-defined parameters.
  - Control physical IOT devices.
  - Provide informative, easy, clear, and secure interaction with the system's actors.
- Assess the related quality attributes and determine the software techniques will achieve its purpose on the project.

1.5 Limitations & use

- No application of the green energy source at the first stage of the project.
- No implementation of a separate users management system. Users will be managed by the system's admin using azure third party federated identity (Microsoft Identity Platform).
- As a minimum viable product, the system will only communicate with the IOT physical devices through MQTT, REST and Serial communication protocols.
- As a minimum viable product, the system will only gathers sensors' data from *LoRaWAN* sensors which integrated on *The Things Network* stack.

2 Methods

This section documents the used methods to manage the development process.

As a result of discussions with stakeholders at the beginning of the project, the developed product was viewed as extendable in each phase. In other words, a new needs/requirements could arise through the development process. Additionally, the development process is constrained by time since a minimum viable product should be ready within a short amount of time. In order to provide the stakeholders with concurrent results in each step of the development process and to ensure a disciplined, model-based and iterative development process, the agile unified process methodology is adopted.

The agile unified process is a combination between the unified process and the agile methodology where the unified process focuses on providing iterative phased software while agile methods focuses on delivering ready software peaces in defined time periods(Sprints) [5].

The development process will be divided in several phases as follows:

- Inception phase: This phase aims to provide the initial features of the project, defining the objectives to satisfy the stakeholders needs and create a first model for the requirements.
- Elaboration phase: This phase aims to continue updating and building on the inception phase while start in providing analysis elements and taking initial design decisions.
- Construction phase: This phase aims to build on the previous phases while providing a working pieces of software satisfies the prioritized requirements.
- Transition: This phase aims to deploy the developed software and apply verification tools in order to measure the software quality attributes.

3 Requirements

The section 3.1 and section 3.2 represent a set of functional/non-functional requirements to meet the needs defined in the 'Introduction' chapter 1. These requirements represent the built elements as they relate to finding analysis elements and taking design decisions.

3.1 Functional requirements

According to the analysis of stakeholders' needs and to provide a system that can fulfill the required tasks, a set of functional requirements are identified and prioritized as shown in figure 1

Id	**Functional requirement**	**Priority**
F01	The user must be able to mange *LoRaWAN* sensors	High
F02	The system must gather data from *LoRaWAN* sensors	High
F03	The user must be able to manage IOT devices supports MQTT, REST, SERIAL communication protocols	High

F04	The system must be able to connect to IOT devices supports MQTT, REST, SERIAL communication protocols	High
F04	The user must be able to manage automation parameters	High
F05	The system must do automation processes by validating sensors' data based on user-defined automation's parameters to control IOT physical devices	High
F06	The user could be able to manage green energy sources	Medium

F07	The user must be able to Control device manually	High
F08	The admin must be able to Manage users' access	High
F09	The user must be able continuously to see information about the status of system elements (sensors, physical devices, automation processes).	High
F10	The user could be able continuously to see statics about the system status based on the green energy sources.	Medium

Table 1: Functional requirements priority table.

3.2 Non-functional requirements

Table 2 shows the non-functional requirements of the system.

ID	Name	Description	Priority
NF01	Scalability	The home management software system must be scalable to keep in the track of possible additions or updates and to contain the pressure resulting from adding new possible software parts.	High
NF02	Usability	The proposed solution assumes that the end-user is a non-IT person. In order to use the system efficiently and learn how to navigate through its different functions effortlessly, the system must satisfy the KISS principle of usability[].	Medium
NF03	Reusability	Physical IOT devices with the same communication protocol may have other differences based on their communication provider and their supported IOT gateway. In order to handle each device's communication provider with the same communication protocol without needing to add new software components, the system must be reusable.	High
NF04	Testability	Data from sensors will be handled by this software system and logical operations will be performed on them. This means it will be necessary to provide a testing environment in order to validate the critical logical parts of the system, as well as to ensure optimal integration between the different components.	High
NF05	Security	The system will handle users' private data.The system must provide authentication and authorization layers. The layers should grant access to the system's parts based on the permissions given by the system administrator.	High

Table 2: System's quality attributes defined as Non-functional requirements.

3.3 Usecase diagram

The functional requirements are captured in detailed usecases in usecases table and a Usecase diagram is made as following.

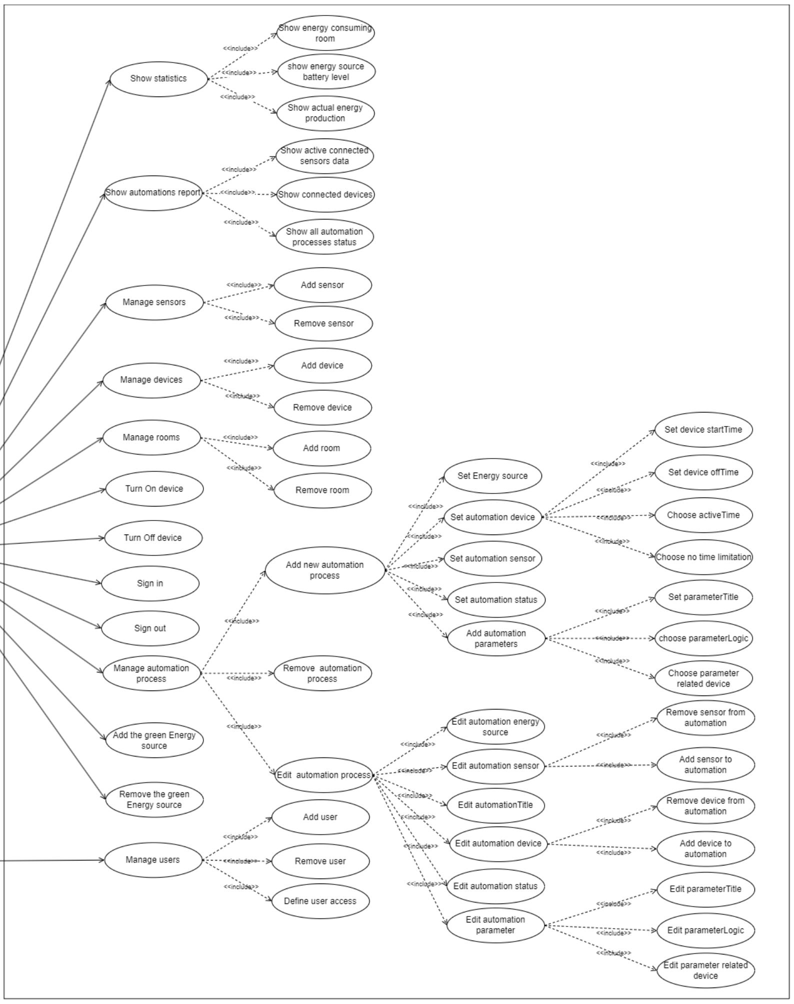


Figure 2: Usecase diagram shows the captured usecases

The usecase diagram explains the interaction between the system and the actors. The system interacts with tow actors, User and System admin. The admin has all grants a user has, while only admin can be responsible for managing users access.

On the diagram, the main usecases are shown to the left. In order to simplify the tasks that a user can perform on the system, a usecases dicomposition process is applied and usecase priority table is done as shown in figure 5 in the 'Appendix' chapter.

4 Analysis

In order to meet up stakeholders' needs, a set of functional and non-functional requirements are represented in the requirements chapter 3. This chapter represents analysis for both functional and non-functional requirements in order to provide analysis elements to help in taking design decisions.

4.1 functional Requirements analysis

This chapter goes through the stakeholders' needs in four subsections, analysing the related functional requirements for each need.

4.1.1 Gather data from sensors

This section focus on analysing the functional requirement F02(*The system must gather data from* LoRaWAN *sensor*) " A sensor is a device that detects and responds to some type of input from the physical environment. The input can be light, heat, motion, moisture, pressure or any number of other environmental phenomena" [6].

IOT sensor needs to connect to an IOT gateway that acts as a broker between a software system and the sensors' local network. Essentially, the IOT gateway manages traffic between physical sensors and software systems that use their data. To deliver sensor data, communication protocols must be used over a network.

4.1.2 Do automation processes based on user-defined parameters

This section focus on analysing the functional requirement F05 (*The system must do automation processes by validating sensors' data based on user-defined automation's parameters to control physical devices*)

'Automation is the technique of making an apparatus, a process, or a system operate automatically' [7].

As part of this project, an automation process aims to control a physical device for a specified period of time. As a result, the system needs to validate a sensor's live value based on values defined by a user.

To satisfy this requirement, a class-based analysis is performed, resulting in four elements: sensor, automation, automation's parameter, and physical device. Figure 3 shows those elements in context of analysis class diagram.

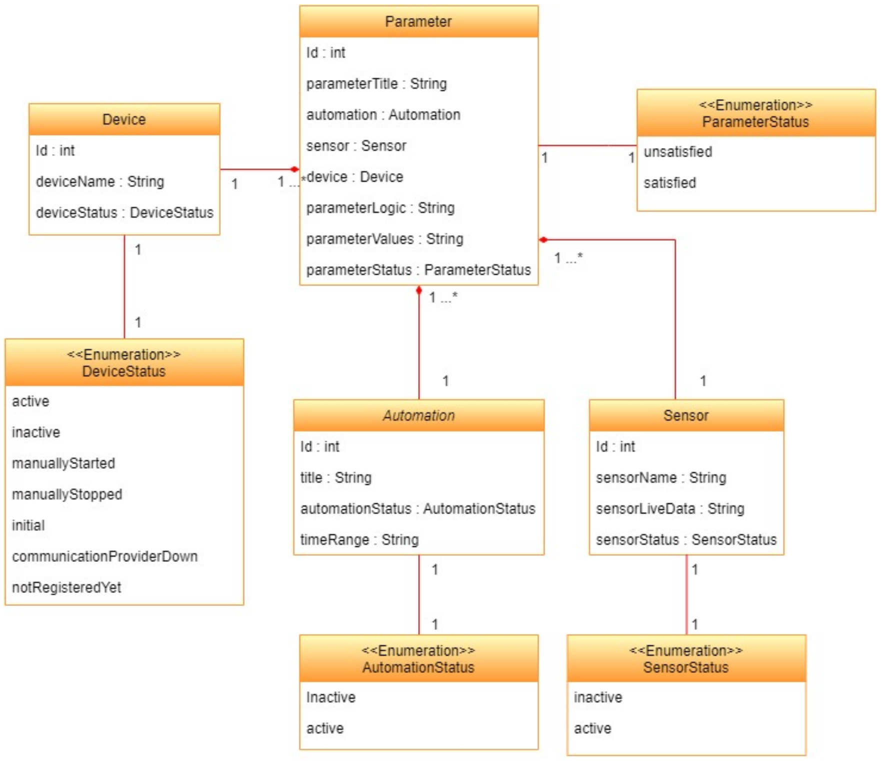


Figure 3: Analyzing the basis system's elements using an Analysis class diagram

The class diagram represents the basis elements needed to validate an automation process. Each element represents a class with attributes as following:

- Sensor class has Id, sensorName, sensorLiveData and sensorStatus attributes.
- Device class has Id, deviceName, deviceStatus properties.
- Automation class has Id, title, automationStatus and timeRange attributes.
- Parameter class has Id, title, parameterLogic, paremeterValue, automation, sensor, and device attributes.
- SensorStatus, DeviceStatus, AutomationStatus and ParameterStatus are Enumeration classes.

Composition relationship is shown by the filled diamond symbol. A composition relationship means that an object instance includes an instance of another class. In other words, each parameter object composed of one and only one Sensor, Automation and Device objects. In this way, handling an automation process only needs to call a parameter object instance and access related sensor, device and automation composed objects.

Automation processes either turn a device on or off. To accomplish this, the system needs a checklist of the elements that compose the automation process. Basically, the system must check multiple conditions before deciding whether to turn on or off a device.

To explain the validation process, an activity diagram is made as following.

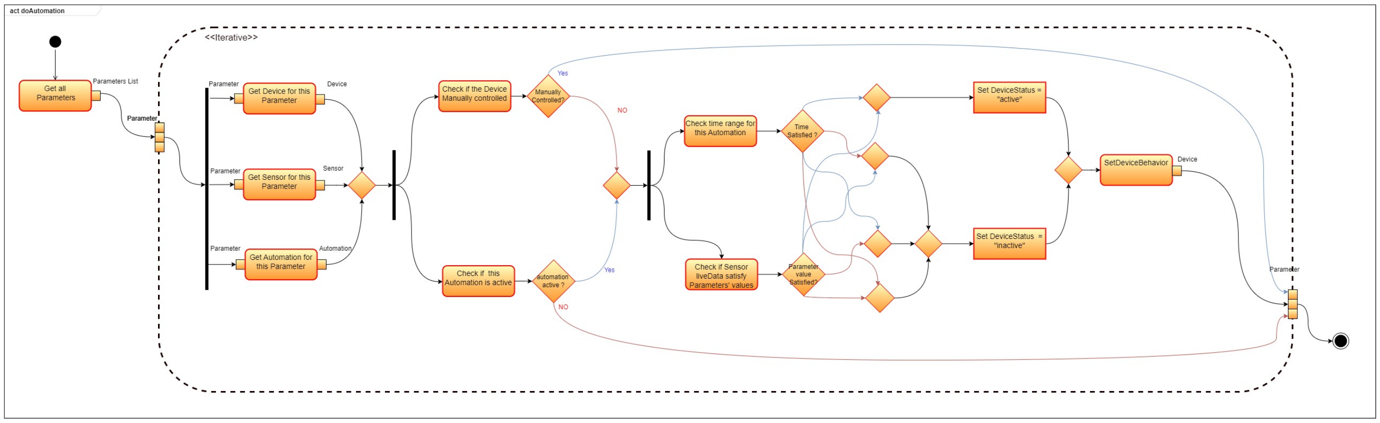

act doAutomation
<<Iterative>>
Get all Parameters
Parameters List
Parameter
Parameter
Get Device for this Parameter
Device
Parameter
Get Sensor for this Parameter
Sensor
Parameter
Get Automation for this Parameter
Automation
Check if the Device Manually controlled
Manually Controlled?
Yes
NO
Check if this Automation is active
automation active ?
Yes
NO
Check time range for this Automation
Time Satisfied ?
Check if Sensor liveData satisfy Parameters' values
Parameter value Satisfied?
Set DeviceStatus = "active"
Set DeviceStatus = "inactive"
SetDeviceBehavior
Device
Parameter

Figure 4: Activity diagram for doAutomation

Figure 4 shows the 'doAutomation' activity. It represents the flow to validate an automation process.
The little squares on sides of the blocks specify the input and output objects.
The hashed block with three vertical squares on each sides means an iterative flow (forEach parameter in parameters' list).
As described previously in the class diagram 3, validating an automation process goes through the automation parameter. That means to validate an automation process, the system needs to invoke a parameter object instance then access the Device, Sensor and Automation composed object instances.
The 'doAutomation' activity starts with getting all parameters then checking the following for each parameter:

- The automation process is active.
- The device is manually controlled.
- The sensor value satisfies the values defined by the user.
- The current time is within the automation's timeRange.

Listed above are the key elements that should be considered when validating an automation process. Based on the results of the check list, the system decides whether to turn a device on or off as follows:

- In cases where a device is being controlled manually or the automation process is not active, the system moves on to the next parameter without setting the related device status (it doesn't switch a device on/off).
- When an automated process is active and the device is not manually controlled, the system checks the following:
  - If sensor's live value satisfies the parameter's values and the automation time range is satisfied the system sets the device status to "active" (Turn on).
  - If sensor's live value satisfies the parameter's values and the automation time range is not satisfied the system sets the device status to "inactive" (Turn off).
  - If sensor's live value does not satisfy the parameter's values and the automation time range is satisfied the system sets the device status to "inactive" (Turn off).
  - If sensor's live value does not satisfy the parameter's values and the automation time range is not satisfied the system sets the device status to "inactive" (Turn off).

After finishing the validation process, setDeviceBehavior activity is invoked to handle the communication with the physical device server. More about setting the device behavior in section 4.1.4 and section 6.3

4.1.3 Interact with the end-user

This section focuses on analysing the following functional requirements:

F01 (*The user must be able to mange* LoRaWAN *sensors*) , F03 (*The user must be able to manage physical devices supports MQTT, REST, SERIAL communication protocols*) , F04 (*The user must be able to manage automation parameters*) , F05 (*The user must be able to manage automation parameters*) , F06 (*The user could be able to manage green energy sources*) , F07 (*The user must be able to Control device manually*) , F09 (*The user must be able continuously to see information about the status of system elements (sensors, physical devices, automation processes)*) , F10 (*The user could be able continuously to see statics about the system status based on the green energy sources*)

To explain the interaction with the end-user usecase description is done on two usecases: UC04 (*Add* LoRaWAN *sensors*) and UC16 (*Add automation parameter*).

Table 3 shows a usecase description for UC04 (*Add* LoRaWAN *sensor*).

Usecase: UC04 (Add *LoRaWAN* sensor)
Primary Actor: User.
Secondary Actor: Admin.
Short description: A user add a *LoRaWAN* sensor to the system.
Preconditions: - The system admin has added the *The Things Network* connection credentials to the system. - The *LoRaWAN* sensor is configured on *The Things Network* server. - The user added a room. - The sensor is connected to a *LoRaWAN* gateway. - The user on Add sensor page.
Main flow: - The user choose a room. - the user insert the sensor dev-Id. - The user insert the sensor name. - The user click on save.
Postcondition: The system starts automatically to gather data from the sensor when it is connected to *The Things Network*.

Table 3: Usecase discription for Usecase UC04 *Add* LoRaWAN *sensor*.

The usecase description represented in table 3 shows the flow to add a new sensor to the system. Each sensor has a unique dev-Id. The dev-Id attribute ensures recognizing a sensor by the system. When a user adds a new sensor with valid dev-Id, the system starts to gather the sensor's data from *The Things Network* web server. Figure 5 shows a sequence diagram explains the interaction to add a new sensor.

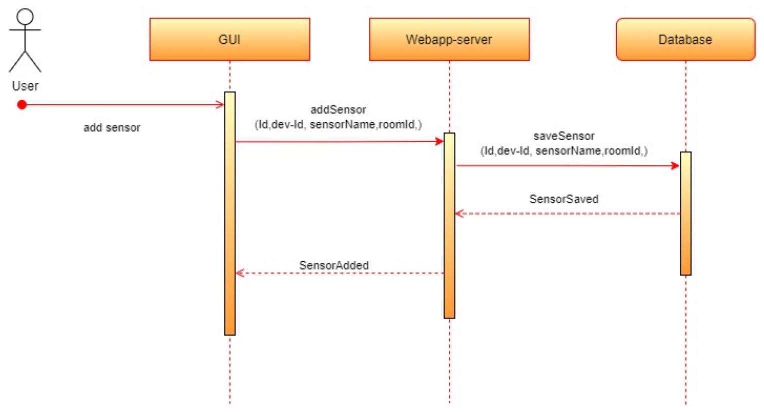


Figure 5: Sequence diagram for usecase UC04 (*Add* LoRaWAN *sensor*)

After a user adds a sensor to the system on the GUI, The web-app server stores the sensor information in the database. The same happens when a user adds a physical devices and a room. After a user added rooms, devices and sensors, he can manage an automation process. To manage an automation process, the user needs to define the automation's parameters. Automation parameter define which device will be controlled based on which sensor's value. for more explanation, a usecase description is done for the usecase UC16 *Add automation parameter* as shown in table 4.

UC16 *Add automation parameter.*
Primary Actor: User.
Secondary Actor: Admin.
Short description: A user add parameters for an automation process.
Preconditions: - The user added a room. - The user added a sensor. - The user added a physical device. - The user configured an automation process with title, status, time range, sensors and devices. - The user on add automation parameters page.
Main flow: - The system shows the user a page with sensors and devices chosen for this automation process. - The user selects which sensor to control a device based on its values. - The user fills the parameter title and click on add a new parameter. - The user choose the device to be controlled. - The user select the device intended behavior . - The user insert the parameter values. - The user click on save.
Postcondition: - The system start automatically to validate this parameter.

Table 4: Detailed Usecase diagram for Usecase UC16 *Add automation parameter.*

After a user added the automation parameters, the system starts automatically to validate this parameter and control the related device as shown in 4.1.4 section.

4.1.4 Control physical devices

This section fucos on analysing the functional requirements F04(*The system must be able to connect to physical devices supports MQTT, REST,* SERIAL *communication protocols*) and F04 (*The user must be able to Control device manually*).

A device can be controlled in two scenarios. The first scenario is after validating an automation process and setting device behavior 4.1.2. The second scenario is to turn on/off device manually by the user. Turning on and off a device occurs the same way in those scenarios.

To explain how the system control a device. a sequence diagram is made for the usecase UC50 (*Turn on device manually*) as following

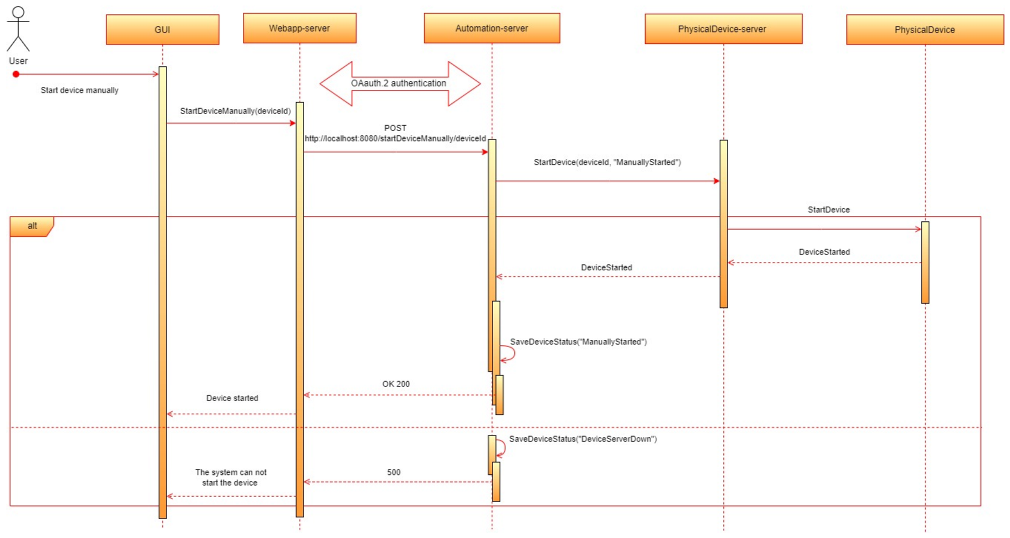


Figure 6: Sequence diagram for usecase UC50 (*Turn device on manually*)

The sequence diagram shows the flow to turn on a device manually by the user. The user click on startDevice. The front-end server for the web application sends a post request to the automation server with device Id as path parameter. The Automation server connect to the device server and ask to turn the device on. After the device-server responses with turning the device on, the Automation server sets the device status as "ManuallyStarted" and returns a 200 http response. If the severe does not receive a success response, then it sets the device status to "CommunicationProviderDown" and returns a 500 response to the web application.

4.2 Non-functional requirements analysis

4.2.1 Scalability

"Software scalability is the ability to grow or shrink a piece of software to meet changing demands on a business" [8]. A way to achieve a scalable software system is by adopting a microservices architecture. Microservices provide easier way to add or update features on the system, since the entire system will not have to be taken offline to do so. To satisfy the scalability requirement, the system will consist of functionally separated services: a service to handle gathering sensors' data, a service to handle automation validation and controlling the physical devices, a service to handle the interaction with the end-users. More information in design section5.

4.2.2 Testability

As mentioned in the previous section, the system will be consist of functionally separated services. This will make the process of testing each service happens in isolation from other services. In other words, testing automation's validation happens in isolation from testing gathering sensors data and interaction with end-users. To satisfy the system testability, the system must be able to conduct unit tests on the logical units. In addition it must be able to perform integration tests to validate the communication between the different services.

4.2.3 Security

Web-application security The stakeholders needs the graphical user interface to be protected with authenticated access using third party authentication service (MicrosoftIdentityPlatform). This means, the user must sign in before accessing the web application's views. Figure 7 shows the authentication process.

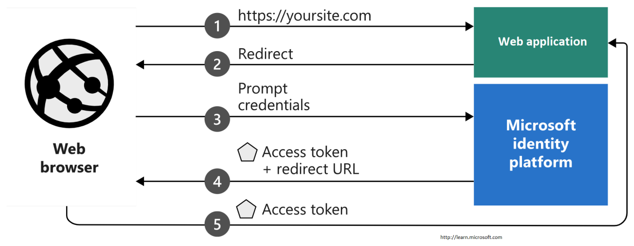


Figure 7: Web-application authentication

Whenever a user attempts to access the web application's GUI, the web application redirects them to Microsoft's login page. In the event that the user successfully signs in on the identity platform, the platform redirects the user to the URL of the web application with the access token. The web application validates the access token and gives the user access to the requested page.

System's information security Resources on the Automation service must be accessed securely. In other words, accessing automation, sensors and devices information must require authorization from other applications. To achieve this, OAuth2.0 protocol will be applied between the Automation server and the web application. Figure 8 represents a sequence diagram for usecase UC42 (*Show automations report*) explains the authorized interaction between the web application and the Automation service.

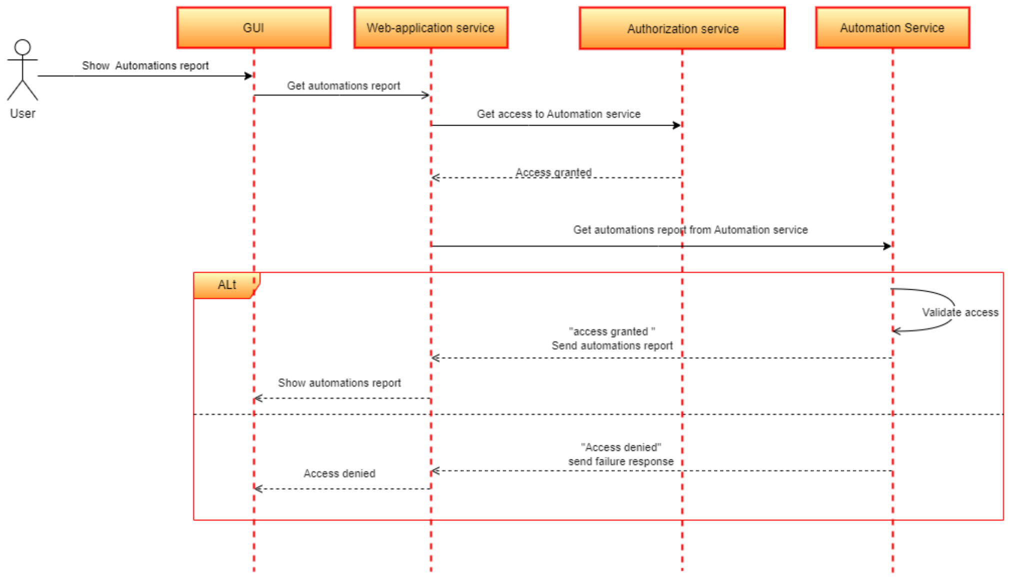


Figure 8: Sequence diagram for usecase UC (*Show automations report*)

As shown in figure 8, the web service needs to provide access from authorization service to interact with the Automation service's resources. When access is granted, then the web service can get the required information from the Automation service. If the access is denied, the Automation service return "Unauthorized" response.

Communication credentials security The Automation service is responsible for communicating with different physical device servers, so the communication credentials should not be available through the server infrastructure. For a secure storage of sensitive information, it will be appropriate to use keyVault secrets with authorized access.

4.2.4 Usability

Considering that the end user may not have any idea about the system, the system must provide a clear, easy, familiar, user-friendly interface. It must also provide a detailed explanation of the system's working mechanism. The user must be guided to manage sensors, devices and automation processes. More details in the design chapter 5.

4.2.5 Reusability

The communication with the physical devices must happens in a reusable context to handle each device's communication Protocol/provider separately. A component-based architecture is required to satisfy the reusability requirement. More information in the design section 5

5 Design

This section describes the structural and behavioral design of the system, including illustrations that demonstrate the design decision to be adopted.

5.1 Architectural design

To provide a scalable system, Microsrvices architecture is adopted. Essentially, loosely coupled services will be applied to ensure an effective management of the system. The services are represented as follows:

- Sensors service: this service is responsible for gathering data from the *LoRaWAN* sensors.
- Web service: this service is responsible for the interaction with end-users and displaying system information.
- Automation service: this service is responsible for handling automation's validation and controlling the the physical devices.

System's services are illustrated in Figure 9 as deployment diagram .

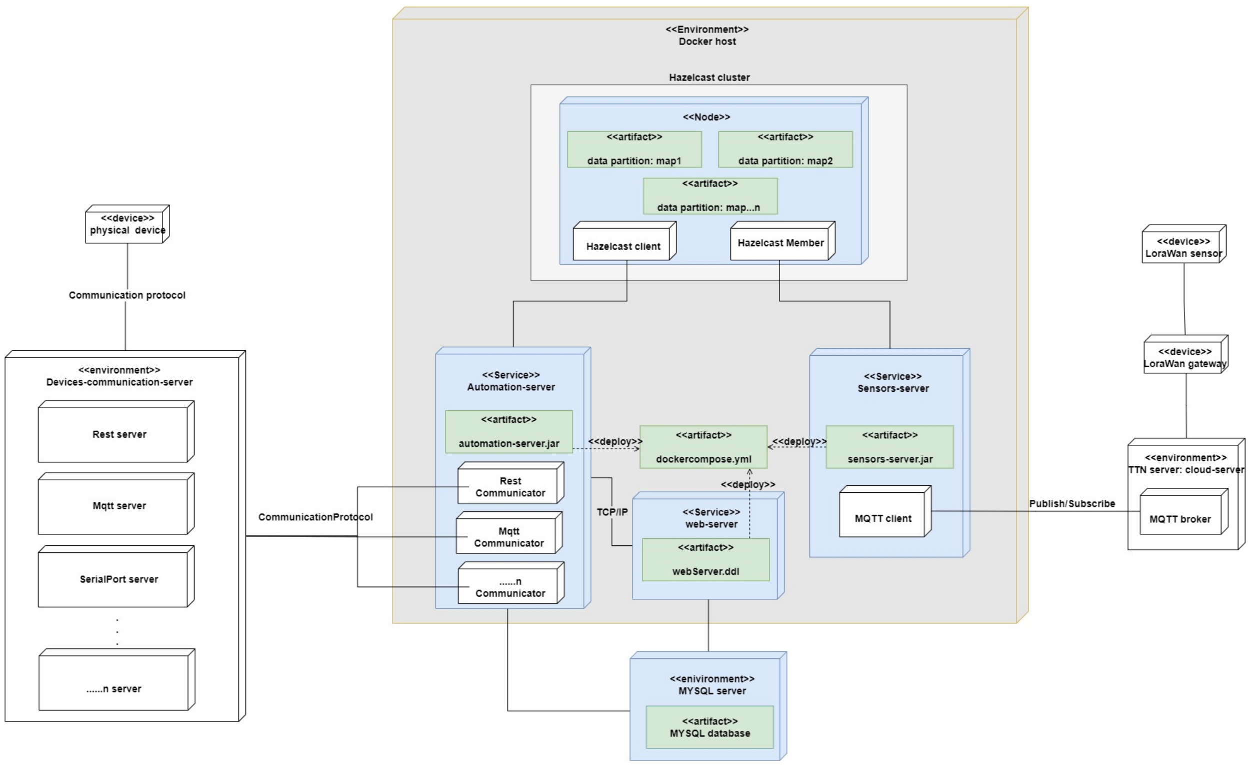


Figure 9: Deployment diagram for the system

Figure 9 shows the system's services represented as Docker containers. The block named Hazelcastcast cluster represents a distributed in-memory data grid. The cluster is responsible for managing distributed data partitions. Those partitions are saved in the cluster's Node and responsible for storing sensors' real-time data. In addition, the cluster provide Real-time event processing and distributed computing which will be discussed later in Behavioral design chapter 6.

Handling the data partitions happens using HazelCast instances' objects named (Hazelcast member and Hazelcast Client). The instances can join into events published by the cluster. More about the event-driven processes in Hazelcast integration chapter 6.4.

Hazelcast distributed architecture provides a scalable interaction with the Automation server. In other words, a single Hazelcast client in the server can receive sensors' real-time data from any member joining the cluster through other applications.

5.2 Web application server design

The web application is responsible for the interaction with end-users, managing system models and getting system information from automation server. In order to provide seperation of concerns in handling the mentioned tasks, MVC design pattern will be applied. The server will consist of three main functional parts: Controllers, Views and Models. Figure 10 illustrates the architectural design of the web server.

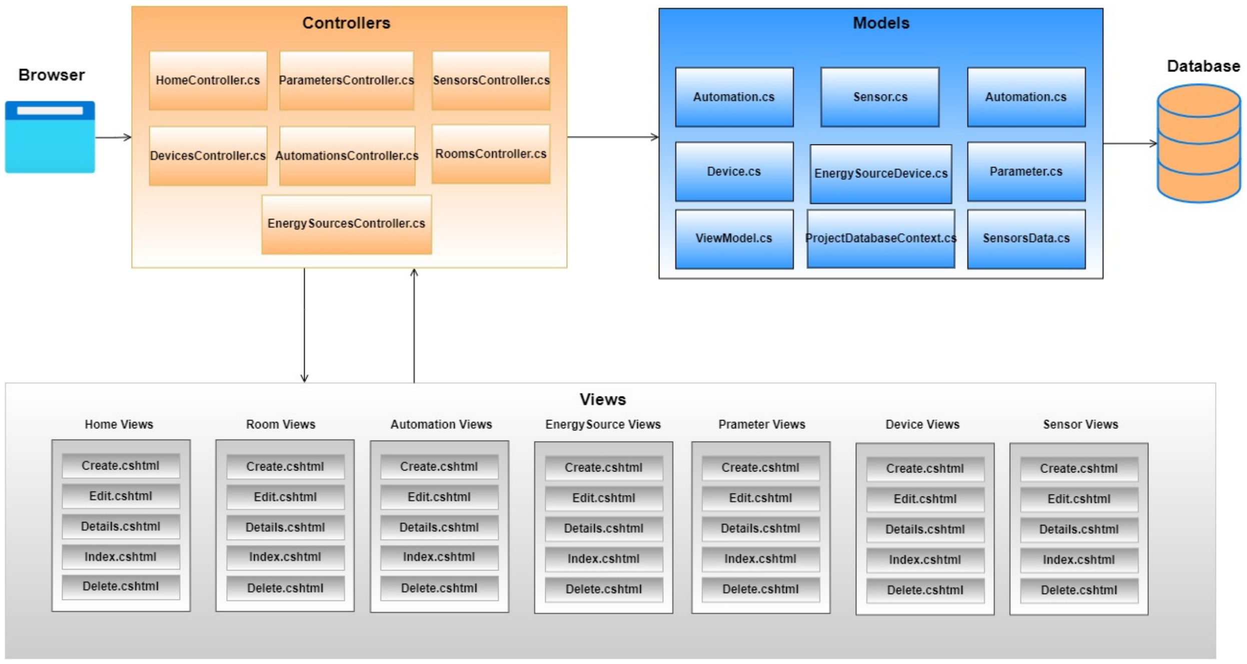


Figure 10: Web server architecture

As shown in figure 10, the Web server consists of three main functional blocks: Controllers block, Models block, and Views block.

The controllers block includes all controllers classes which have the responsibility to receive the user requests from the browser, map the requested information as models object and updating the views with the related models information.

Models block has the responsibility for interaction with database and map the business logic (database information) into models objects.

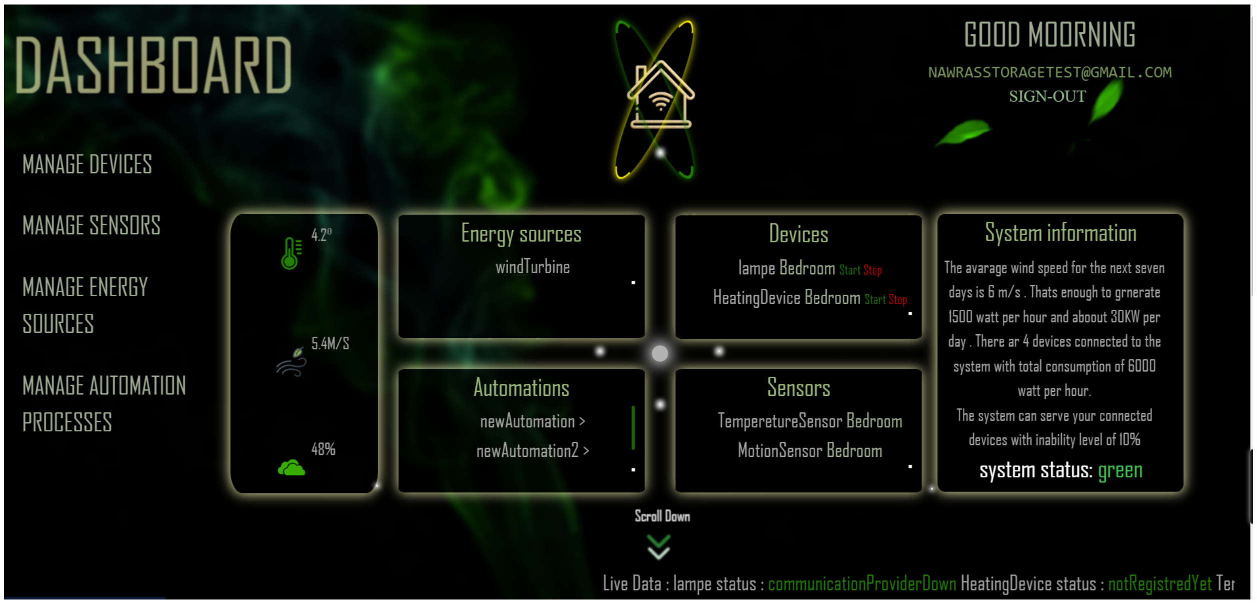
Views block has the responsibility to manage the visual interaction with the users and displaying the models' information sent by the controllers.

5.3 Graphical User Interface design

The Graphical User Interface has the responsibility for giving the users the ability to see system's models information and managing it. Buttons, links and informative elements which display the system information is placed directly at the home page while system models' lists and CRUD operations on them are represented in other pages.

Colors are chosen to reflect the idea of the product, attract the user and provide simple visual communication with the GUI.

In order to provide easy and guided interaction with the users, the home page is included animated elements. The animated elements help the user to understand the GUI's concepts and to guide him/her through its different parts. Moreover, animated elements improve the user experience when interacting with the system. Figure 11 shows the design of the home page.



The avarage wind speed for the next seven days is 6 m/s . Thats enough to grnerate 1500 watt per hour and aboout 30KW per day . There ar 4 devices connected to the system with total consumption of 6000 watt per hour.
The system can serve your connected devices with inability level of 10%
system status: green

Live Data : lampe status : communicationProviderDown HeatingDevice status : notRegistredYet Ter

Figure 11: Home page design

As shown in figure 11, the home page contains the elements responsible for displaying the system information and links to mange system's models. The list to the left represents links to manage the system's models(Sensors, Rooms, Devices and Automation processes). The four boxes in the middle displays the system models information.

The box named Devices lists all devices in the system and provides the ability to start/stop a device manually by the user. If a user start/stop a device manually, the device disconnected from all automation process.

Sensors, Energy sources and Automations boxes include all aensors, automations, energySources added to the system. The box to the left with temperature, wind and cloud symbols provides the real-time weather information. The box to the right shows the system status.

The Marquee down the page includes all System's models real-time values updated each 20 seconds. It shows the real-time values of all active sensors and the connection's real-time status for each device.

The animated elements are represented by the two green tree leaves on top-right, the green symbol named scroll down, the green& yellow blades covering the smart home symbol in the top of the page and the shining white circles in the middle of the page.

When a user click on Manage sensors, Manage rooms, Manage Energy sources and Manage automation processes, the user directed to related model list where he/she can add, delete, edit and show details on each system model.

5.4 Automation server design

In order to satisfy the reusability requirement, The automation server will be component-based. In other words, the server will be composed of several components as follows:

- A core component has the responsibility for receiving sensors' data from the Hazelcast cluster, validating automation and managing access to the server information.
- A common component defines the services required by the core component to communicate with the physical devices.

- Service providers components (Connectors) provide the logic required by the core component to control the physical devices.
- Persistence component manages accessing the data sources.

Figure 12 shows a component diagram for the automation server.

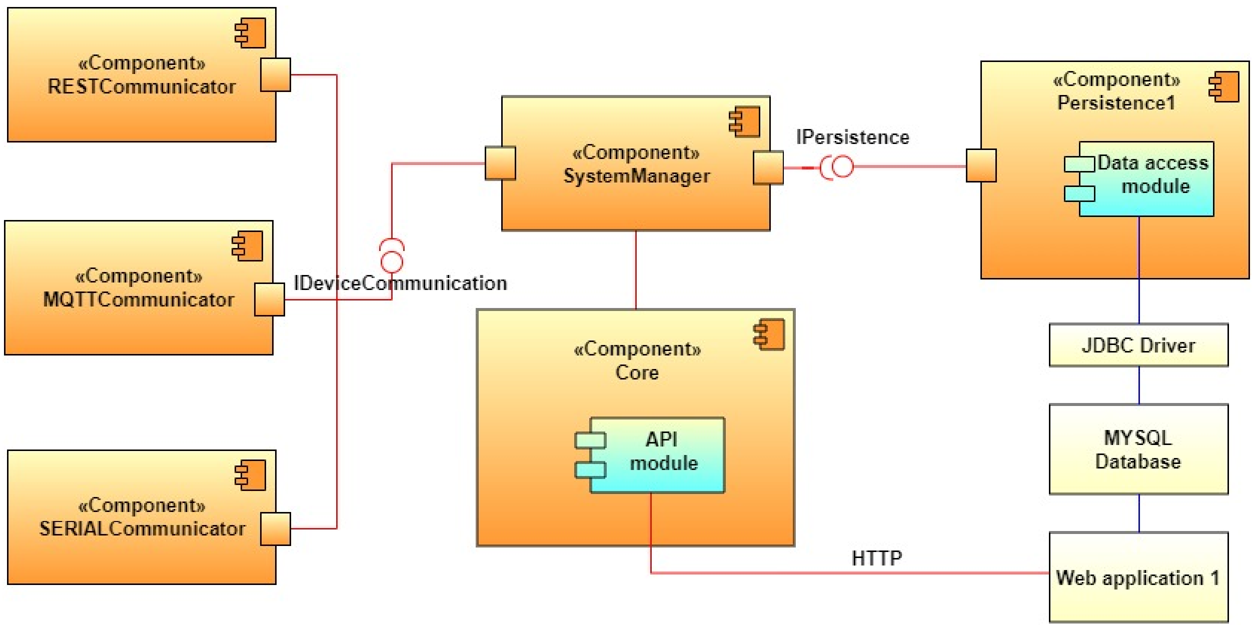


Figure 12: Components diagram for Automation server

The component diagram in figure 12 illustrates the structure of the automation server. Blocks with orange color represents the internal components as follows:

- SystemManager component represent the common module which provide the services needed by the core component(IDeviceCommunication, Ipersistence) in order to connect to the physical devices and get system information from data sources.
- The three vertical blocks to the left are the communicators components which represent service providers. Each communicator component has its own logic to connect to the physical devices. More about the communication in the behavioral design chapter, 'interaction with physical device' section 6.3.
- Core component represents the domain . It includes modules for gathering data from the Hazelcast cluster, validating automation processes and communicating with the physical devices.
- The orange block to the right represents the Persistence service provider. The Persistence service provider provides Automation server's information(Sensors, Devices, Automation and Parameters) from a MYSQL database at the start of the system. Managing database entities happens through a web application(the yellow block entitled Web application1). The web application can communicate with the automation server through its API endpoints. More about the communication in the behavioral design chapter 6.

5.5 Database design

After defining and analysing the system entities, an ER diagram is made to explain the structural design of the database as shown in figure13.

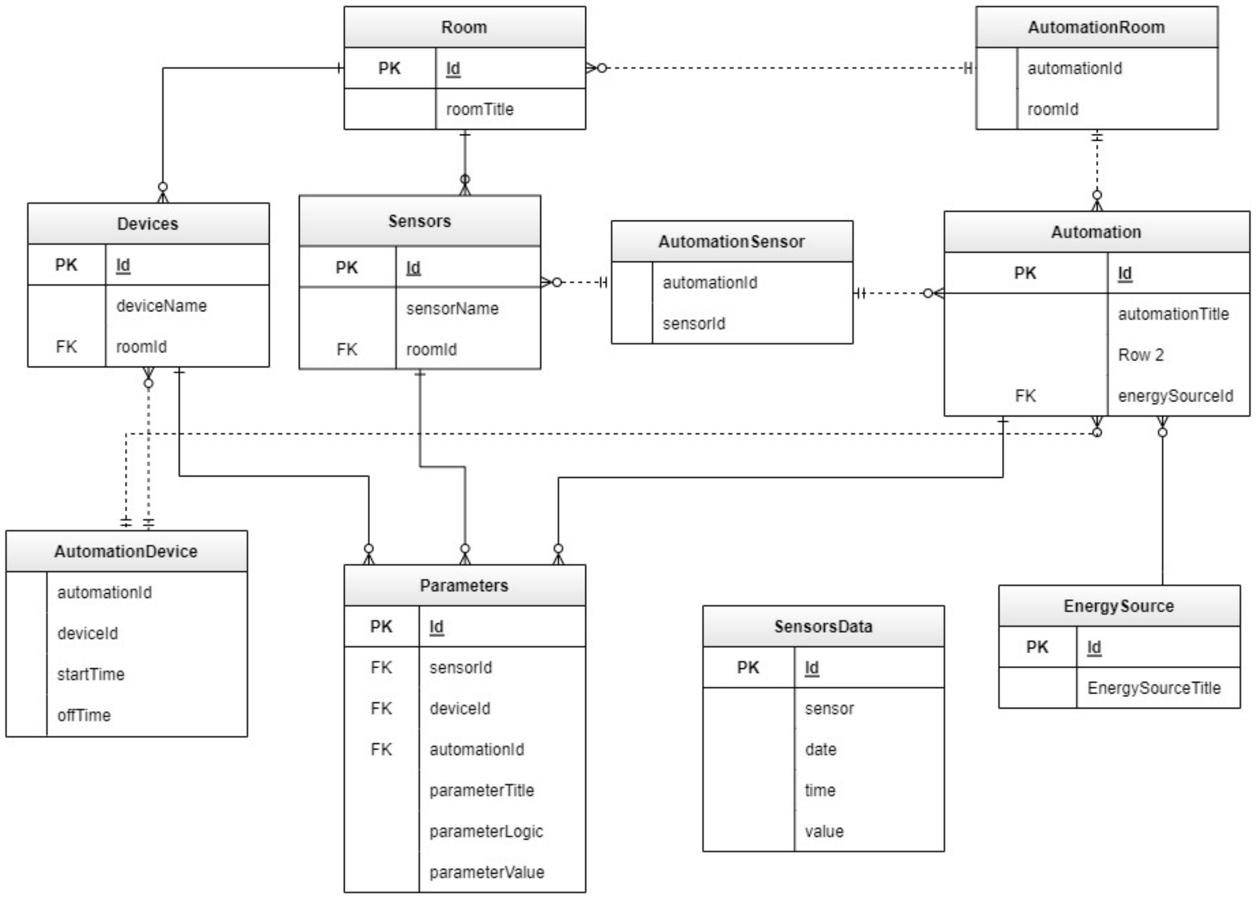


Figure 13: Caption

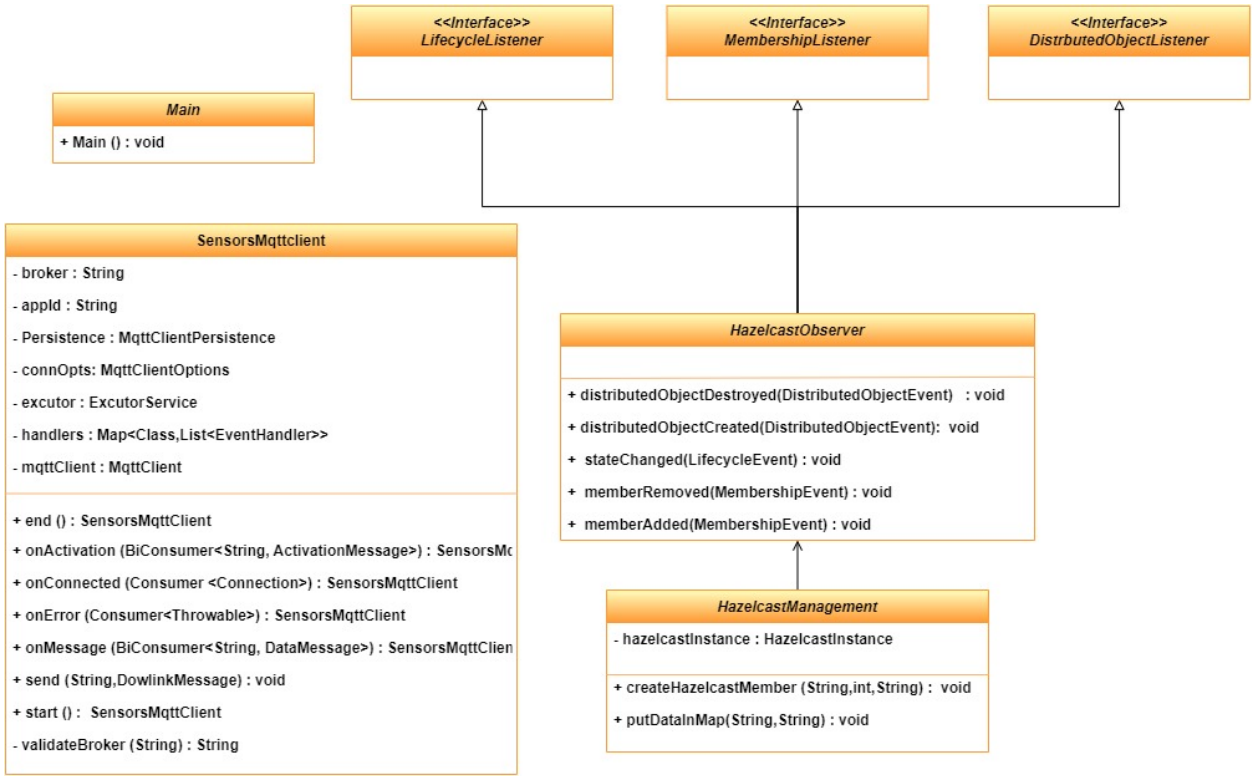
Figure 13 shows the database's tables and relationships between them in form of an ER diagram. The database consists of seven entity tables and 4 junction tables. Each junction table represents many to many relationship between two entities. The diagram explains the relationships between the system's entities as follows:

- A Room entity has many sensors and many devices(one to many relationship (Room - Sensor and Room - Device).
- A Room entity can join many automation processes and an automation entity can include multiple rooms(many to many relationship (Room - Automation).
- A Sensor entity can join many automation processes and an Automation entity can include several sensors(many to many relationship (Sensor - Automation).
- A Device entity can join many automation processes and an Automation entity can include several devices(many to many relationship(Device - Automation).
- An EnergySource entity can join many automation processes but an automation entity can only have one energySource(one to many relationship(EnergySource - Automation).
- A Parameter entity can iclude only one sensor, device and automation process while each sensor, device and automation entities can join multiple parameters(one to many relationship (Parameter - Sensor , Parameter - Device and Parameter - Automation).

5.6 Sensors server design

Sensors server has the responsibility for receiving sensors' real-time data and save them on the Hazelcast cluster.

As discussed in 'limitations' section 1.5, the stakeholders agreed to use one type of sensors(*LoRaWAN* sensors). the main responsibility the Sensors server must provide is connecting to the *The Things Network* server and listen to the provided sensors' data. Figure 14 represents the structural design of Sensors server using a design class diagram.



<<Interface>> LifecycleListener
<<Interface>> MembershipListener
<<Interface>> DistrbutedObjectListener
Main
+ Main () : void
SensorsMqttclient
- broker : String
- appId : String
- Persistence : MqttClientPersistence
- connOpts: MqttClientOptions
- excutor : ExcutorService
- handlers : Map<Class,List<EventHandler>>
- mqttClient : MqttClient
+ end () : SensorsMqttClient
+ onActivation (BiConsumer<String, ActivationMessage>) : SensorsMc
+ onConnected (Consumer <Connection>) : SensorsMqttClient
+ onError (Consumer<Throwable>) : SensorsMqttClient
+ onMessage (BiConsumer<String, DataMessage>) : SensorsMqttClien
+ send (String,DowlinkMessage) : void
+ start () : SensorsMqttClient
- validateBroker (String) : String
HazelcastObserver
+ distributedObjectDestroyed(DistributedObjectEvent) : void
+ distributedObjectCreated(DistributedObjectEvent): void
+ stateChanged(LifecycleEvent) : void
+ memberRemoved(MembershipEvent) : void
+ memberAdded(MembershipEvent) : void
HazelcastManagement
- hazelcastInstance : HazelcastInstance
+ createHazelcastMember (String,int,String) : void
+ putDataInMap(String,String) : void

Figure 14: Design class diagram for Sensors server

As shown in figure 14, Sensors server represents a single module consists of four classes: SensorsMqttClient, HazelcastManagement, HazelcastObserver and A Main class. SensorsMqttClient class represents a mqtt client to connect to *The Things Network* mqtt broker and provides callback methods to handle the incomming data messages. HazelcastManagement class has the responsibility to create hazelCast members and put the data into Hazelcast cluster's data partition. HazelcastObserver class provides methods to listen on the changes happens through the cluster.

6 Behavioral design

6.1 Data sources integration

As discussed in figure 12, Automation server will be able to handle different Persistence components. The persistence component has the responsibility to deliver the data from data sources at the start of the system. The interaction aims to provide the system with the information required to validate automation processes and control the physical devices. Data access module in the Persistence component must extends the system models defined in the common component and implement the related service contract (interface) in order to provide Automation server with the information. More about system models and interaction with data sources in 'Implementation' chapter 7. To explain data sources integration with automation server, an interaction diagram is done as shown in figure 15.

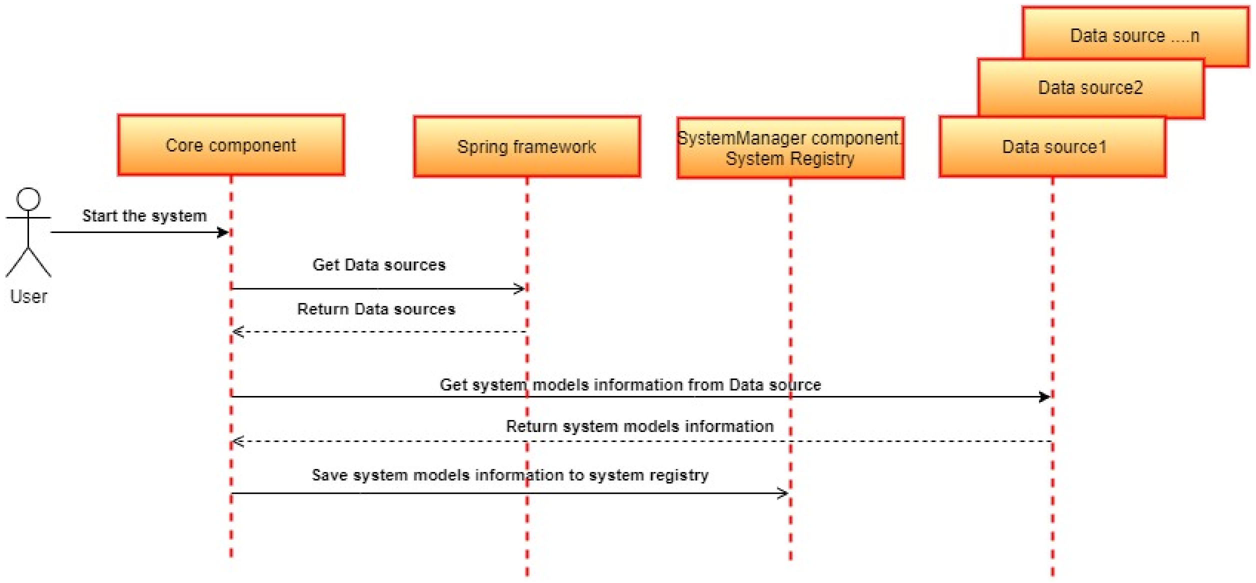

Data source ....n
Data source2
Core component
Spring framework
SystemManager component System Registry
Data source1
User
Start the system
Get Data sources
Return Data sources
Get system models information from Data source
Return system models information
Save system models information to system registry

Figure 15: Interaction diagram for data sources integration.

Figure 15 shows the interaction between Automation server and Data sources described as follows:

- When the system starts, the core component communicates with the active data sources and fetch all information for devices, automations, sensors and parameters from the data source.
- Data source's information saved as system models objects in the system registry mapped by their data source name.
- The server has no communication with the data sources after system registries are filled.
- After system registries are filled, all real-time operations on the system models happens through the API endpoints. More about the interaction with those endpoints in 'Automation server's API interaction' section 8.3.1.

.

6.2 Automation server's API interaction

As described in 'Data sources integration' section 6.1, after system registries are filled, all real-time operations on the system's models happens through the API endpoints. In other words, if a user adds a new sensor on the web application's GUI, the web application needs to update the system models by sending a HTTP request to the server API' endpoint.More information in section 7.1.4

6.3 Interaction with the physical devices

The Automation server is composed of several components. Communication with the physical devices happens through connectors components. Each connector component represents a service provider which provides a communication client to connect to the physical device. Managing gluing each service provider component on the service component happens through Orchestrator component.

Figure 16 represent an activity diagram for "setDeviceBehavior" function to explain the communication with the physical devices.

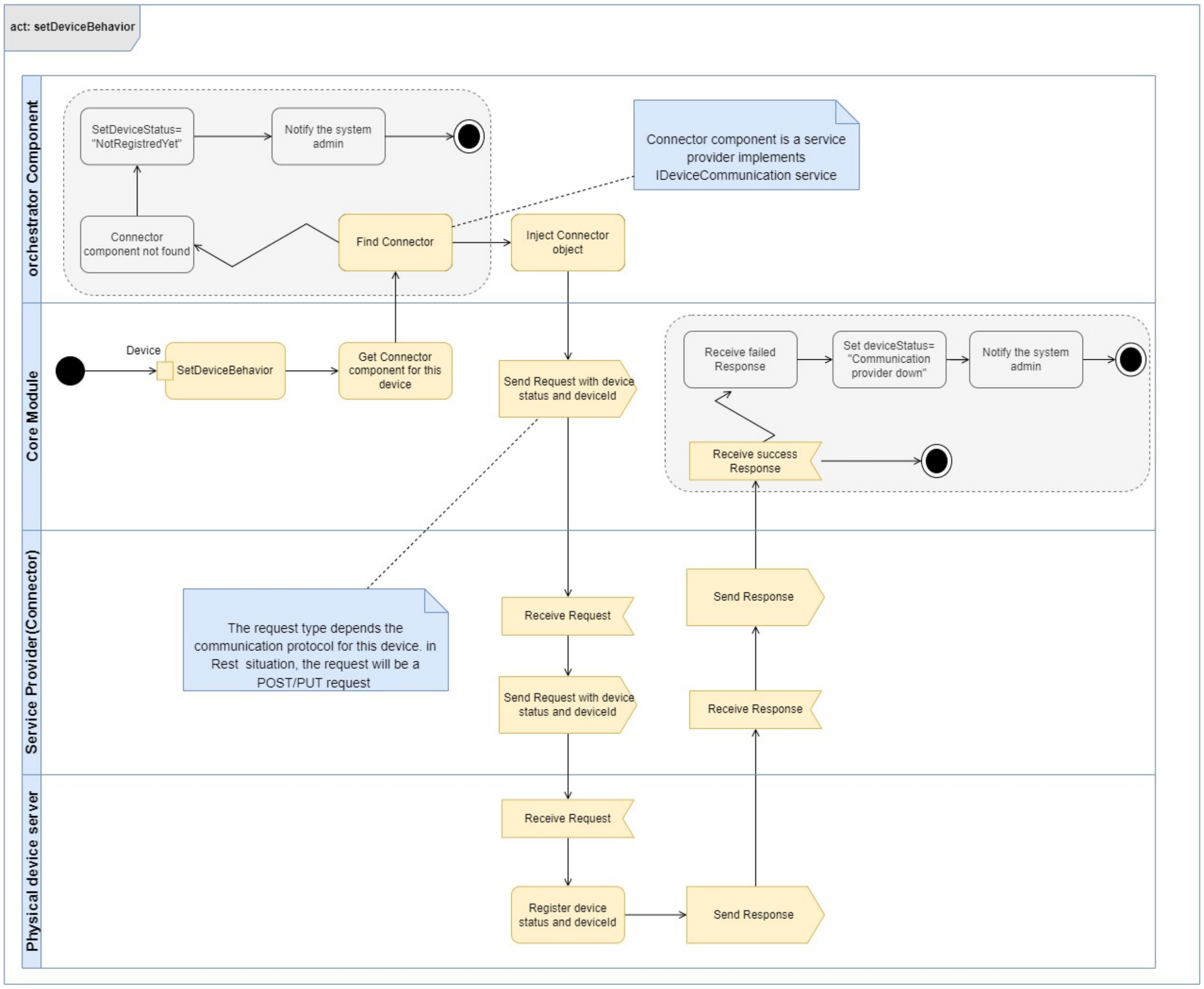

act: setDeviceBehavior
orchestrator Component
Core Module
Service Provider(Connector)
Physical device server
SetDeviceStatus= "NotRegistredYet"
Notify the system admin
Connector component not found
Find Connector
Inject Connector object
Connector component is a service provider implements IDeviceCommunication service
Device
SetDeviceBehavior
Get Connector component for this device
Send Request with device status and deviceId
Receive failed Response
Set deviceStatus= "Communication provider down"
Notify the system admin
Receive success Response
The request type depends the communication protocol for this device. in Rest situation, the request will be a POST/PUT request
Receive Request
Send Request with device status and deviceId
Send Response
Receive Response
Receive Request
Register device status and deviceId
Send Response

Figure 16: Activity diagram for setDeviceBehavior

Figure 16 describes the system behavior to connect to a physical device. Each device object has two specific properties to determinate its behavior (deviceId and status). The activity starts after finishing an automation's validation or after a user turn on/off a device manually.

This diagram has four lanes. Each lane represents a component that takes on a part of the responsibility in order to connect to the physical device. The activity starts with getting the connector object from the Orchestrator component. If the Orchestrator could not find the connector, that means the communication provider credentials are not included in the server and the device status will be set to "Device is not registered yet".

If the Orchestrator component finds the connector, it glues the connector instance into the core component. Then the core component sends a request through the connector instance to the physical device server.

As soon as the physical device server returns a success message, the system updates the device status. The system notifies the system administrator if the physical device server responds with a failure message and set the device behavior to "Communication provider down".

6.4 Hazelcast integration

Sensors service must deliver real-time sensors' data to the Automation service in order to validate automation processes. Delivering live data happens by adopting a distributed data structure between Sensors server and Automation server using Hazelcast. Hazelcast provides in-memory data storage accessible by several parties. In other words, a storage where data structures can be stored and accessed by multiple applications. Data transfer between Sensors server and Automation server using Hazelcast happens as follows:

1. Sensors server create a data partition in the Hazelcast cluster.
2. Automation server add observer object on the data partition.
3. Sensors server collects real-time sensor's data and saves it in the data partition in the cluster.
4. Automation server get updated from the observer object and obtain the data added by the Sensors server.

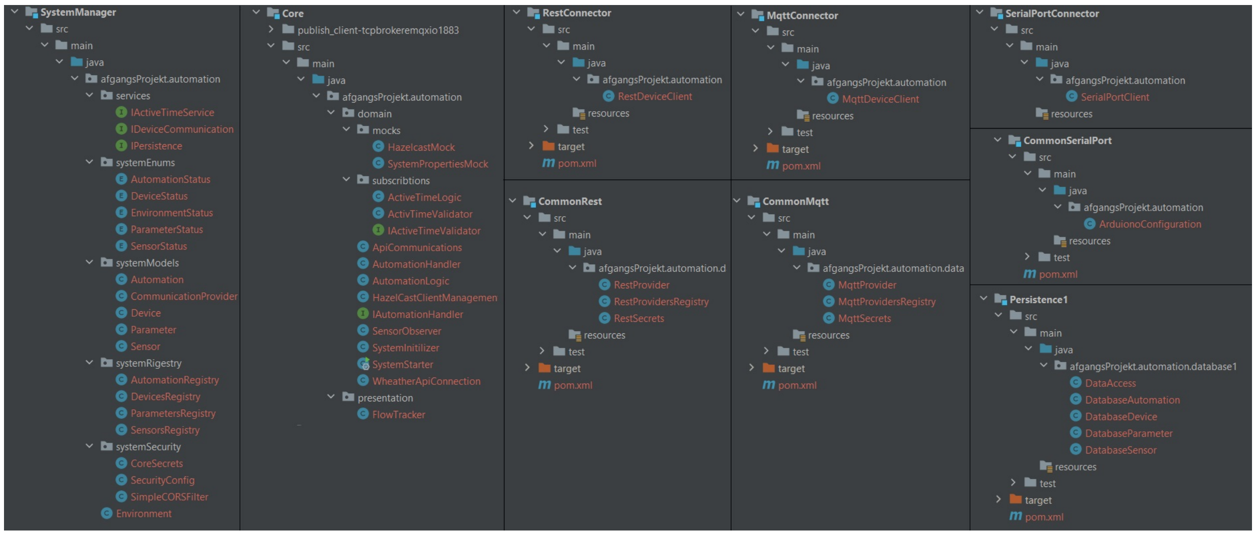
7 Implementation

The source code of the implemented system is available on this link [9]

This section goes through the implementation process focusing on the used technologies in order to met the design decisions made at the design level. Figures and code snippets are included to provide more clarification of the implementation techniques.

7.1 Automation server implementation

The server implemented in java language using spring framework. The structural design defined in 'design' chapter 5 is reflected by implementing separate java modules. Implementing several java modules with specified functions could satisfies the reusability requirement. This is by giving each module a specific reusable function. Figure 17 shows the java modules form the Automation sever.



```
SystemManager
  src
    main
      java
        afgangsProjekt.automation
          services
            IActiveTimeService
            IDeviceCommunication
            IPersistence
          systemEnums
            AutomationStatus
            DeviceStatus
            EnvironmentStatus
            ParameterStatus
            SensorStatus
          systemModels
            Automation
            CommunicationProvider
            Device
            Parameter
            Sensor
          systemRigestry
            AutomationRegistry
            DevicesRegistry
            ParametersRegistry
            SensorsRegistry
          systemSecurity
            CoreSecrets
            SecurityConfig
            SimpleCORSFilter
          Environment
```

```
Core
  publish_client-tcpbrokeremqxio1883
  src
    main
      java
        afgangsProjekt.automation
          domain
            mocks
              HazelcastMock
              SystemPropertiesMock
            subscribtions
              ActiveTimeLogic
              ActivTimeValidator
              IActiveTimeValidator
            ApiCommunications
            AutomationHandler
            AutomationLogic
            HazelCastClientManagemen
            IAutomationHandler
            SensorObserver
            SystemInitilizer
            SystemStarter
            WheatherApiConnection
          presentation
            FlowTracker
```

```
RestConnector
  src
    main
      java
        afgangsProjekt.automation
          RestDeviceClient
      resources
    test
  target
  pom.xml
```

```
CommonRest
  src
    main
      java
        afgangsProjekt.automation.d
          RestProvider
          RestProvidersRegistry
          RestSecrets
      resources
    test
  target
  pom.xml
```

```
MqttConnector
  src
    main
      java
        afgangsProjekt.automation
          MqttDeviceClient
      resources
    test
  target
  pom.xml
```

```
CommonMqtt
  src
    main
      java
        afgangsProjekt.automation.data
          MqttProvider
          MqttProvidersRegistry
          MqttSecrets
      resources
    test
  target
  pom.xml
```

```
SerialPortConnector
  src
    main
      java
        afgangsProjekt.automation
          SerialPortClient
      resources
```

```
CommonSerialPort
  src
    main
      java
        afgangsProjekt.automation
          ArduionoConfiguration
      resources
    test
  pom.xml
```

```
Persistence1
  src
    main
      java
        afgangsProjekt.automation.database1
          DataAccess
          DatabaseAutomation
          DatabaseDevice
          DatabaseParameter
          DatabaseSensor
      resources
    test
  target
  pom.xml
```

Figure 17: Automation server modules

Figure 17 is a screenshot of Intellij IDE shows the components of Automation server represented as java modules. MqttConnector, RestConnector, SerialPortConnector and Persistence1 modules represent the service providers which provide the services to be used by the core module. The core module represents the consumer component and provides the methods uses the services. CommonMqtt, CommonRest and CommonSerialPort modules provide the information needed by the service providers to perform the services. SystemManager module represents the common component which provides the following : services' contracts in form of Interfaces, system models needed to validate automation processes, SystemRegistries to map system models into system registry objects. More information in 'data sources integration chapter' 7.1.2.

7.1.1 System starter

Spring framework facilitates the mission of satisfying the reusable structure shown in figure 17. This is because Spring framework can works as Orchestrator component to mount each service provider on the related service using dependency injection. Dependency injection, in simple definition means mounting java objects without direct dependency between them. In other words, loose coupling between objects. Those objects called beans. Managing beans' lifecycle and mounting them happens through the ApplicationContext interface which could be declared as ConfigurableContext at a spring boot starter application class to be accessed through the code.

A code snippet is taken from the systemStarter class in order to explain the declaration of the context object as shown in listing 1

Listing 1: SystemStarter.java

```
1
2
3
4  @EnableWebSecurity
5  @EnableGlobalMethodSecurity(prePostEnabled = true)
6  @SpringBootApplication
7  @ComponentScan("afgangsProjekt.automation")
8  public class SystemStarter  {
9     public static ApplicationContext applicationContext;
10    public SystemStarter(ApplicationContext _applicationContext) {
11       applicationContext = _applicationContext;
12    }
13
14    public static void main(String[] args) throws InterruptedException,
         SQLException {
15       applicationContext = SpringApplication.run(SystemStarter.class, args);
16
17    }
18    public static ApplicationContext getApplicationContext() {
19
20       return applicationContext;
21    }
22
23 }
```

Listing 1 shows the systemStarter.java class. This class has the responsibility to start the application. The static public applicationContext attribute declared at line 9 represents a configureable Applicationcontext object has the responsibility to manage the application's beans life cycle and grant accessing them in different parts in the code through getApplicationContext() method. When the application starts, the spring container search the directory defined using @ComponentScan in line 7 to find declared beans. Among the ways uses to declare a bean, the server used the annotation method. The annotation method declares a bean by adding the @Component annotation on the top of the class.

7.1.2 Data sources integration

As described in behavioral design section 6 figure 4, the server must be able to handle multiple data sources at the start of the system. In order to satisfy this design-decision, a System initializer class is implemented as shown in listing 2

Listing 2: SystemInitializer.java

```
@DependsOn(value = {"Environment"})
@Component("SystemInitilizer")
public class SystemInitilizer {
    DevicesRegistry devicesRegistry;AutomationRegistry automationRegistry;
       ParametersRegistry parametersRegistry;SensorsRegistry sensorsRegistry;
       ApplicationContext applicationContext;Environment environment;

    @Autowired
    public SystemInitilizer(DevicesRegistry _devicesRegistry, AutomationRegistry
       _automationRegistry,          ParametersRegistry _parametersRegistry,
       SensorsRegistry sensorsRegistry,ApplicationContext _applicationContext,
       Environment _environment
    ) {

        devicesRegistry = _devicesRegistry; automationRegistry =
            _automationRegistry; parametersRegistry = _parametersRegistry;
            sensorsRegistry = _sensorsRegistry; applicationContext =
            _applicationContext; environment = _environment;
11          initialize();
12      }
13
14      private void initialize() {
15          if (environment.getEnvironmentStatus().equals(EnvironmentStatus.production
            ) || environment.getEnvironmentStatus().equals(EnvironmentStatus.
            development)) {
16              applicationContext.getBeansOfType(IPersistence.class).forEach((k, v)
                    -> {
17                  if (v.isActiveDatabase()) {
18                      String databaseName = v.getDatabaseName();
19                      FlowTracker.trackFlow("Initializing system entities from
                            database :" + databaseName);
20                      try {
21                          devicesRegistry.setAllDevices(databaseName,v.getAllDevices
                                ());
22                          sensorsRegistry.setAllSensors(databaseName, v.
                                getAllSensors());
23                          automationRegistry.setAllAutomations(databaseName, v.
                                getAllAutomations());
24                          parametersRegistry.setAllParameters(databaseName, v.
                                getAllParameters());
25
26                      } catch (SQLException e) {
27                          e.printStackTrace();
28                      }
29                  }
30              });
31          }
32      }
33  }
```

Listing 2 shows the implementation of the SystemInitializer.java class. In this class, System registry beans are injected in the class's constructor. Each system registry instance represents a Map to store system models' information to be used later in automation validation.

To ensure filling the system registries at the start of the system, intialize() method is included in the constructor as shown in line 11 . By this way, when the systemInitializer bean is created, initialize() method is activated. The intialize method get the applicationcontext object and use it to find all beans implement the IPersistence interface(line 16) then it iterates all found beans and check if the database is active. The isActiveDatabase() method ensures that the dataAccess module at the persistence component is ready to provide system models' information. If the database is active then a setter methods set the system information obtained using the service providers (Lines 21 to 25) to the system registries.

The DataAcess modules in the service providers needs to extend the system models classes in order to provide a valid instances can be used through automation validation.

7.1.3 Automation Validation

Automation validation process has been analysed and described in 'Analysis' chapter, section 4.1.2 . That in order to satisfy the functional requirement Nf05 *The system must do automation processes by validating sensors' data based on user-defined automation's parameters to control physical devices.* In this section, the implementation of the validation process will be documented using a snippet code represented in listing 3.

Listing 3: doAutomation() method.

```java
@Override
    public void doAutomation() throws ParseException, InterruptedException {
        flowTracker.trackFlow("AutomationValidation started");
        // get All Parameters
        ArrayList<Parameter> parameters = ParametersRegistry.getAllParameters();
        flowTracker.trackFlow("Iterating parameters ");
        for (Parameter p: parameters) {
            // get parameter device
            Device device = p.getDevice();
            // get parameter sensor
            Sensor sensor = p.getSensor();
            // get parameter automation
            Automation automation = p.getAutomation();
            // is device manually controlled?
            boolean deviceManuallyControlled= AutomationLogic.
                isDeviceManuallyControlled(device);
            // is automation process active?
            boolean automationActive= AutomationLogic.isAutomationActive(
                automation);
            // is sensor active?
            boolean senesorActive = AutomationLogic.isSensorActive(sensor);
            // is automation time satisfied?
            boolean automationTimeSatisfied= AutomationLogic.
                checkTimeConditionIsSatisfied(automation);
            // is parameter values satisfied
            boolean parameterValuesSatisfied=AutomationLogic.
                validateParameterValues(p);
            flowTracker.trackFlow(
                    "Automation validation result: /n" +
                    "Parameter :"+p.getId()+" values("+p.getParameterValue()+")
                        satisfy sensor(" +p.getSensor().getSensorName()+")live
                        value("+p.getSensor().getLiveValue()+") --> Parameter
                        status :"+p.getParameterStatus().toString()+"/n"+
                    "The automation "+ automation.getAutomationTitle() + "is" +
                        automation.getAutomationStatus()+"/n"+
                    "Time conditions are" + automationTimeSatisfied + "/n"+
                    "The device " + device.getDeviceName() + "is" + device.
                        getStatus());

            if (automationActive&&!deviceManuallyControlled&&senesorActive){

                if(parameterValuesSatisfied&&automationTimeSatisfied){
                    device.setStatus(DeviceStatus.activ);
                    flowTracker.trackFlow("Setting device "+ device.getDeviceName
                        () + "status to active");

                }else {
                    device.setStatus(DeviceStatus.inactive);
                    flowTracker.trackFlow("Setting device "+ device.getDeviceName
                        () + "status to inactive");
40
41                  }
42              }
43
44                 AutomationLogic.setDeviceBehavior(device);
45
46          }
47          flowTracker.trackFlow("
                ..........................................................");
48      flowTracker.printFlow();
49      }
```

Listing 3 shows the implentation applied in order to validate automation processes.

In the 'analysis' chapter figure 3, the system models are defined in four entities (Sensor, Device, Parameter and Automation). Furthermore, a composition relationship is defined between those elements. In other words, each parameter instance should be composed of Sensor, Device and Automation instances. In the same chapter, an activity diagram 4 is made to explain the activities/ actions needed to validate an automation process.

The code snippet shown in Listing 3 represents the implementation of the doAutomation() method. The method has the responsibility for validating an automation process.

The method starts with getting all parameters' instances initialized and stored in the system parameters' registry as shown previously in the 'Data sources integration' section 6.1. Then the method iterates the Parameters list and gets the composed Sensor, Device and Automation instances for each parameter (lines 8-13).

'automationActive', 'sensorActive' and 'deviceManuallyController' boolean variables (lines 15-19) represent the validators. Those validators are responsible for deciding if the method will continue the validation process or not. If the value of one of those variables equals false, then the method ignores the current parameter and moves on to the next one without setting the composed device status (lines 31-49). Otherwise, if the values of all the validator variables equal true, the method continues to set the device instance's status.

Setting the device status depends on the value of the two boolean variables 'automationTimeSatisfied' and 'parameterValuesSatisfied'(lines 20-23) . If the value of both of those two variables equal true the device status sets to 'active' (line 33-36), while if one or both of those variables equal false the device status sets to 'inactive' (line 37-42).

After setting the device status, the server is ready to connect to the physical device server by invoking the setDeviceBehavior() method (line 44).

The flowTracker object shown within the code represents the presentation layer of the server. The FlowTracker class provide printFlow() as a method that prints documentation for the operations acquired by the server from the first time the server starts to the final method it performs. That done only if the working envirnoment is development

7.1.4 API interaction

The Automation server represents a REST API and exposes endpoints to enable the other applications interacting with system's models. The API endPoints are handled using a REST controller class implemented in the core module. Listing 4 represents a snippet code to show the interaction whith an API endpoint in order to add a new sensor instance to the system models.

Listing 4: addsensor()

```
     @PostMapping("/addSensor")
    public void addSensor(@RequestBody String body){
    JSONObject requestBody = new JSONObject(body);
    Sensor sensor = new Sensor(requestBody.getInt("sensorId"),requestBody.getString
        ("name"),"0", SensorStatus.inactive,requestBody.getString("devId") );
    sensorsRegistry.addSensor(requestBody.getString("databaseName"),sensor);

    }
```

Listing 4 shows the interaction with the Automation server to add a new sensor. @PostMapping annotation shows the HTTP request type needed to call the API endpoint(POST request) and defines the request path as string value. The @RequestBody annotation define a String variable named 'body' in order to provide the needed information to add a sensor to the server.

7.1.5 Automation server security

The Automation sever represents a REST API that can be accessed through its exposed endpoints. To access those end points, the client application (The application makes the requests) needs to provide valid access token and to pass the cors policy definded by the server. Listing 5 shows the implementation of SecurityConfig.java class to provide authorized access to the server.

Listing 5: SecurityConfig.java

```
1  @Configuration("conf")
2  @EnableGlobalMethodSecurity(securedEnabled = true, prePostEnabled = true)
3  public class SecurityConfig extends WebSecurityConfigurerAdapter {
4          @Override
5          protected void configure(HttpSecurity http) throws Exception {
6              http.cors().and() // (1)
7                      .authorizeRequests().anyRequest().authenticated() // (2)
8                      .and()
9                      .oauth2ResourceServer().jwt(); // (3)
10         }
```

In Listing 5, a snippet of code from SecurityConfig.java class is shown to illustrate how the Automation server validates access to its resources. The class is annotated with '@Config' and '@EnableGlobalMethodSecurity' annotations. The '@Config' annotation informs the Spring framework that the class is a configuration class, while the '@EnableGlobalMethodSecurity' annotation instructs the framework to use the class methods to manage server security.

The class implements the 'WebSecurityConfigurerAdapter' class, which uses the configure() method to validate each HTTP request reaches the server's endpoints. This is done by checking if the request header contains a valid access token. Request header token must be obtained from the same Authorization service defined in 'app.properties' file.

At line 6, .cors() method is used to check if the client application is identified by the cors policy defined by the server. 'authorizeRequests (), anyRequest (), authenticated (), oauth2ResourceServer (), jwt () ' methods (line 7-9) ensures that the client applications makes the requests are authenticated by the Authority service and have valid access tokens from it.

7.2 Web server implementation

MVC design pattern is adopted by the web server, as noted in the 'Design' chapter 5. The server is implemented using C programming language using ASP.Net Core framework.

The purpose of this section is to demonstrate the implementation of the Create() method in the SensorsController.cs class. The create() method represents the implimantaion of the UC04 (*(Add LoRaWAN senso*) that analysed in the 'analysis' chapter 4 and described in details in 'Automation server API interaction' section 7.1.4.

Create() method is responsible for receiving adding sensor request from the browser, validating and mapping the information included in the request as Model object, save the model object in the database, create a HTTP request, include the query information required by Automation server in the request body, send the http request to Automation server. To provide more clarification a snippet code for the Create method is represented in listing 6

Listing 6: Create() language

```
1  [HttpPost]
2  public Task < IActionResult > Create([Bind("Id,sensorName,devId,roomId")] Sensor
       sensor) {
3    if (ModelState.IsValid) {
4    // adding the sensor to the database
5      _context.Add(sensor);
6      _context.SaveChanges();
7    // obtaining access token to call Automation server
8      string accessToken = await _tokenAcquisition.GetAccessTokenForUserAsync(
           scopes);
9    // create httpClient to call Automation server
10     HttpClient client = new HttpClient();
11     client.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("
           Bearer", accessToken);
12   // add request body parameters
13     string requestBody = "{" + $ "\"sensorId\":{sensor.Id}" + "," +
14       "\"databaseName\":\"database1\"" + "," +
15       $ "\"devId\":\"{sensor.devId}" + "\"" + "," +
16       $ "\"name\":\"{sensor.sensorName}" + "\"}";
17     HttpContent content = new StringContent(requestBody, Encoding.UTF8, "
           application/json");
18   // sending the request
19     HttpResponseMessage response = client.PostAsync("http://localhost:8080/
           addSensor", content);
20     ViewData["roomId"] = new SelectList(_context.Room, "Id", "Id", sensor.roomId)
           ;
21     ViewBag.context = _context;
22     return RedirectToAction(nameof(Index));   }
23   return View(sensor);
24 }
```

Lsting 6 shows the implementation of Create() method. '[HttpPost]' tag (line 1) define the request type the browser must provide to interact with the method. The method parameter ' Bind (" Id , sensorName , devId , roomId ")] Sensor sensor '(line 3) represents the information included in the request body mapped as Sensor model instance.

First, the method check if the information provided in the request are mapped correctly to a system model (line 3). if the model instance is valid(Mapped correctly), it uses the _context object to add a row in the database table(sensor)(lines 5-6). After a sensor is added succesfully to the database, the sensor's instance information sends to the Automation server API to update the server models.

Calling the Automation server API endpoint happens as follows:

- A string variable with name 'accessToken' is made to store the access token required to call Automation server.
- After the access token is obtained, a http client is created (line 10).
- The accesstoken is added to the client requests' headers (line 11).
- A String variable with name 'requestBody' is made to store the sensor instance information as request body json parameters (lines 12-16).
- The request body added to the client requests' conentent parameter (line 17).
- The request is sent (line 19)

After Sensor's instance information saved on the database and sent to the Automation server, the method return a redirect action which redirects the user to the index view page(line 23).

7.3 Sensors server implementation

The Sensors server has the responsibility for receiving sensors data and send them to the Automation server. The stakeholders were agreed to use a specific type of sensors. These sensors are based on *LoRaWAN* technology and are connected to *The Things Network* IOT gateway using a *LoRaWAN* gateway. *The Things Network* IOT gateway provides a MQTT broker to fetch sensors' data.

The system needs to provide a MQTT client that has access to *The Things Network* web-server.

Figure 18 shows a sequence diagram explaining the connection with sensors.

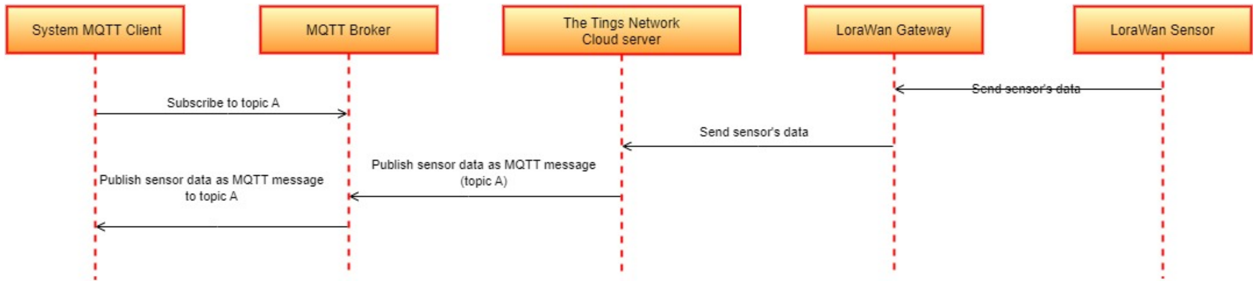

System MQTT Client
MQTT Broker
The Tings Network Cloud server
LoraWan Gateway
LoraWan Sensor
Send sensor's data
Subscribe to topic A
Send sensor's data
Publish sensor data as MQTT message (topic A)
Publish sensor data as MQTT message to topic A

Figure 18: Interaction diagram shows gathering data from *LoRaWAN* sensors

Figure 18 Shows sensors use the *LoRaWAN* technology requires a *LoRaWAN* gateway. This gateway send sensor's data to *The Things Network* server which publish data to a MQTT broker with a specific topic. A topic is an UTF-8 string that the broker uses to filter messages for each connected client [10]. The system's MQTT client needs to subscribe to this topic to get sensor's data. Topics management happens on *The Things Network* web portal[11]. Listing 7 shows the implementation of the Main() method in order to explain how the server gathers sensors' data and send them to the Automation server.

Listing 7: Main.java

```
1      public static void main(String[] args) throws Exception {
2        //Create HazelCast member
3        String clusterName = "GreenSmartHome";
4        int port = 5701;
5        String publicAdress = System.getenv("publicAdress");
6        HazelcastManagement.createHazelcastMember(clusterName, port, publicAdress);
7        String brokerTCPIP = System.getenv("broker");
8        String applicationId = System.getenv("appId");
9        String accessKey = System.getenv("accessKey");
10       //connect to TTN
11       SensorsMqttClient sensorsMqttClient = new SensorsMqttClient(brokerTCPIP,
             applicationId, accessKey);
12       sensorsMqttClient.onMessage((String devId, DataMessage data) -> {
13         System.out.println(devId);
14         HazelcastManagement.putDataInMap(devId, data.toString())});
15       sensorsMqttClient.onError((Throwable _error) -> System.err.println("error: "
               + _error.getMessage()));
16       // start the client
17       sensorsMqttClient.start();
18   }
```

Listing 7 explains how the Sensors server gather and store sensors' data. In the Main method, a Hazelcast cluster is created and a member instance is added to the cluster(line 6) . A Mqtt client is created and callback methods are sat to receive messages from the broker, then the start method is called to start the connection(lines 11-17). When the Mqtt client receive a callback message from the broker, the related data partition in the cluster is updated (line 14).

8 Verification

In this section, verification tools are used to ensure that the development process follows the defined requirements. [12]

8.1 Units testing

Unit tests are conducted during the implementation phase to ensure that the logical units behave as intended. A unit test determines whether the software unit was implemented correctly or if it needs to be modified. In this section, unit tests are shown for the following doAutomation() and validateParameterValues() units. Other unit tests are attached in the 'Appendix' chapter **??**.

To provide the required inputs to these methods, a separate class 'SystemModelsMock.java' was created which contains Mock objects. The mock class is used to initialize the system model instances (Sensor, Automation, Device, Parameter) needed to validate an automation process. The full implementation of the 'SystemModelsMock.java' class represented in the Appendix chapter 11.

In order to isolate the units testing from the integration with data sources, the 'Environment.java' class is created. Through the system, the class is injected as a bean on the constructors of the system's classes. A system environment can be in one of four states: development, production, unitTesting, or testing. 'Appendix' chapter 10.4 contains the full implementation of 'Envirnoment.java' class.

8.1.1 validateParameterValues() unit test

validateParameterValues() method tests whether the system compares sensor values with those defined by the user correctly as shown in listing 8.

Listing 8: sensorDataSatisfyParameterValues() unit test

```
1          @Test
2      void sensorDataSatisfyParameterValues() throws ParseException,
          InterruptedException {
3          Sensor sensor = SystemModelsMock.getSensor();
4          sensor.setLiveValue("10");
5          Parameter parameter = SystemModelsMock.getParameter();
6          parameter.setParameterLogic("onBetween");
7          parameter.setParameterValue("{\"value1\":\"5\",\"value2\":\"15\"}");
8          sensorsRegistry.addSensor("test", sensor);
9          parametersRegistry.addParameter("test", parameter);
10         AutomationLogic.validateParameterValues(parameter);
11         //Check that the sensor value is between the two values defined by a user
12         Assertions.assertEquals(ParameterStatus.satisfied,parameter.
               getParameterStatus());
13         // Change the sensor's live data value to not satsfying the values defined
               by a user
14         sensor.setLiveValue("20");
15         AutomationLogic.validateParameterValues(parameter);
16         Assertions.assertEquals(ParameterStatus.unsatisfied,parameter.
               getParameterStatus());
17     }
```

Testing validation of sensor live value can be seen in listing 8. In this test, two Sensor and Parameter mock objects are created, with a live value of 10 for the sensor and 5 and 15 for its parameter. Also, this parameter logic is set to "onBetween", which means that the sensor value must be between two defined values. As soon as the validateParameterValues() method is invoked, the test asserts that the value is satisfied(line 12). Line 14 shows that the sensor's value has been changed to 20, which falls outside of the parameter's defined values. The test checks if the parameter status equals to 'satisfied' when the sensor's value falls between the parameter values and 'unsatisfied' when it falls outside of the parameter values(lines 12-16).

8.1.2 doAutomation() unit test

Listing ?? shows a snippet code that tests the doAutomation() unit in order to ensure that the validation of an automation process works correctly.

Listing 9: doAutomation() test

```java
 @Test
void doAutomation() throws ParseException, InterruptedException {

  // creating the mock object instances
  Device device = SystemModelsMock.getDevice();
  Sensor sensor = SystemModelsMock.getSensor();
  Parameter parameter = SystemModelsMock.getParameter();
  Automation automation = SystemModelsMock.getAutomation();
  // fill the system registry with the mock instances
  devicesRegistry.addDevice("test", device);
  sensorsRegistry.addSensor("test", sensor);
  parametersRegistry.addParameter("test", parameter);
  automationRegistry.addAutomation("test", automation);
  // set the device status to startedManually
  device.setStatus(DeviceStatus.manuallyStartted);
  // ensure the system does not validate the automation process when the device
      is manually controlled
  Assertions.assertEquals(DeviceStatus.manuallyStartted, device.getStatus());
  // set the automation to inactive
  automation.setAutomationStatus(AutomationStatus.inactive);
  // check the system does not validate the automation process when it is not
      active
  Assertions.assertEquals(DeviceStatus.manuallyStartted, device.getStatus());
  // change the device status to initial
  device.setStatus(DeviceStatus.initial);
  // set the automation to active
  automation.setAutomationStatus(AutomationStatus.activ);
  // set the sensor's live value
  sensor.setLiveValue("10");
  // set the parameter's values and parameter logic
  parameter.setParameterLogic("onBetween");
  parameter.setParameterValue("{\"value1\":\"5\",\"value2\":\"15\"}");
  // set a time range for this automation satisfy the current time between
      22/11/2022 to 22/11/2023
  automation.setAutomationTimeRange("{\"start\":\"22-11-2022-10-00\",\"off
      \":\"22-11-2023-10-30\"}");
  //ensure the system sets the device status to "active" when values and time
      conditions are satisfied
  automationHandler.doAutomation();
  Assertions.assertEquals(DeviceStatus.activ, SystemModelsMock.getDevice().
      getStatus());
  //Change the time condition to be invalid values
  automation.setAutomationTimeRange("{\"start\":\"22-11-2022-10-00\",\"off
      \":\"23-11-2022-10-30\"}");
  // ensure the system sets the device status to inactive when the time
      condition is not satisfied
  automationHandler.doAutomation();
  Assertions.assertEquals(DeviceStatus.inactive, device.getStatus());
  // set the time conditions back to valid values
  automation.setAutomationTimeRange("{\"start\":\"22-11-2022-10-00\",\"off
      \":\"22-11-2023-10-30\"}");
  // change the sensor value to a value does not satisfy the parameter values
  sensor.setLiveValue("30");
```

```
    automationHandler.doAutomation();
    // ensure the system sets the device status to inactive when the parameter
        values are not satisfied
    Assertions.assertEquals(DeviceStatus.inactive, device.getStatus());
  }
```

Listing 9 represents the doAutomation() unit test.The test that verifies the validity of an automation process by checking all logical situations that can arise. Each step is explained in green color in the listing.

8.2 doAutomation model checking

This section aims to verify the flow of validating an automation process represented by the 'doAutomation' activity using Uppaal. The 'doAutomation' activity is described in details in the 'Analysis' section using an activity diagram4

'Uppaal is an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata, extended with data types (bounded integers, arrays, etc.)' [12].

The activity diagram is modeled using uppaal in order to check if the validation of an automation process happens without deadlocks. In other words, model checking ensures that the flow contains no logical conflicts between its states. A conflict could arise between two or more logical situations' results. If a conflict is present, the system will not be able to arrive at a decision. The decision could be either to set the device status to "active" or to "inactive".

Figure 19 shows the doAutomation activity modeled using Uppaal. This model is built as 4 separate automata to simulate the behavior of the system models explained through the doAutomation activity. Each automaton represents a flow between different states in order to simulate a specific part of the entire behavior of the activity.

The process of validating an automation process happens through automation's parameter. Each parameter element represents an automation process. In more details, each parameter instance composed of an automation, a Device and a Sensor instances. In order to validate an automation process, the system needs to check the following:

- The automation process is defined as active by the user.
- The sensor related to this automation process satisfies the values defined by the user.
- The device related to this automation process is not manually controlled by the user.
- The time conditions defined by the user to this automation process is satisfied by the current time.

The automaton named 'doAutomation' in figure 19 represents the main flow to validate an automation process using a list of 3 automation parameters.

The automaton named 'Iterator' has the responsibility to iterate the parameters list.

The automaton named 'validator' has the responsibility to validate each bullet point represented in the above check list.

The automaton named "Device" has the responsibility for setting the device status based on the check list validation results.

All those automata are connected using synchronizing channels (represented in turquoise color in the figure). A synchronizing channel activates a partial flow on an automaton based on a signal receives from another automaton.

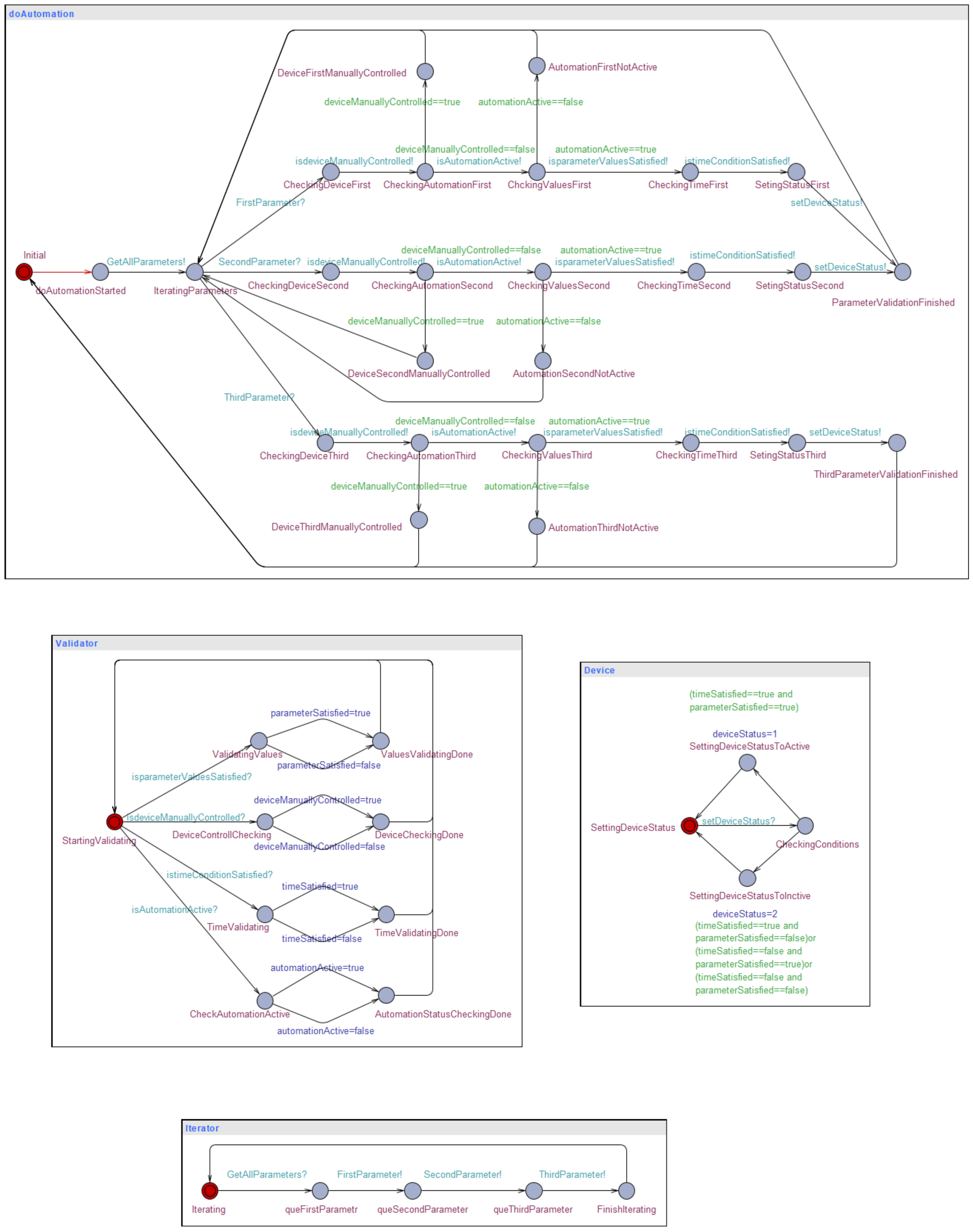
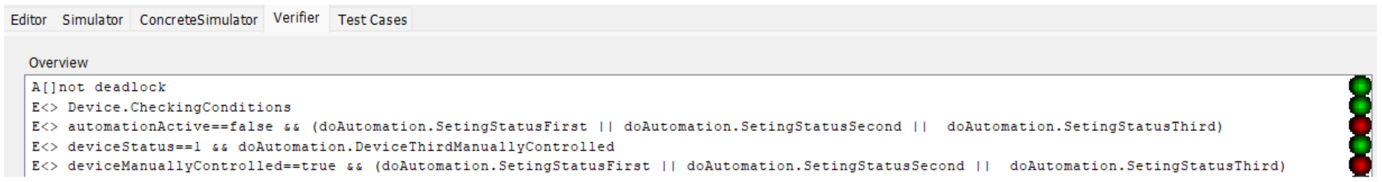


Figure 19: doAutomation activivty(Uppaal model)

As shown in figure 19, the model assumes that a list of three automations' Parameters are initialized at as input for the activity and needed to be validated. The initial states (filled red cycles) represent the start states of the automata. The automata move between several states until it finishing validating all parameters and setting a device status. Flow between all states of the model is described as follows:

- The model starts from the initial state in 'doAutomation' automaton. When the automaton reaches the 'doAutomationStarted' state a synchronizing channel named 'GetAllParameters' sends a signal to the 'Iterator' automaton in order to start iterating the parameters list.
- The 'Iterator' automaton starts to iterate parameters and send a signal using sycronizing channel named 'FirstParameter' to the 'doAutomation' automaton.
- When the 'doAutomation' automaton receives the signal, it moves to 'CheckingDeviceFirst' state where it sends a signal to the 'Validator' automaton using 'isDeviceManuallyControlled' synchronizing channel in order to check if a device is manually controlled.
- The 'Validator' automaton validates the device status and update a variable named 'deviceManually-Controlled' value.
- When validating the device status is finished the 'doAutomation' automaton move to 'CheckingAutomationActive' state.
- If the value of the variable 'deviceManuallyControlled' equals to 'true', the automaton move to 'DeviceFirstManuallyControlled' state. That means the device is manually controlled and the automaton will not continue the validation process and it moves to the next parameter.
- If the value of the variable 'deviceManuallyControlled' equals to 'false', then the automaton continues the validation process and sends a signal 'isAutomationActive' to the 'Validator' automaton.
- The 'Validator' automaton validates the automation status and update the value of a variable named 'isAutomationActive'
- If the value of the 'isAutomationActive' variable equals to 'false', the 'doAutomation' automaton does not continue the validation process and continues to the next parameter.Otherwise, it continues the validation process and ask the 'validator' automaton to check time conditions and the values defined by a user are satisfied.
- When parameter validation is finished, the 'doAutomation' automaton sends a singal to the 'Device' automaton using a synchronizing channel named 'setDeviceStatus'.
- The 'Device' automaton, check the updated variables through the validation process and updates an int variable named 'deviceStatus' to 1 "active" if the time conditions and sensor value satisfies the values defined by the user. Otherwise, it set the value of the variable to 2 "inactive".

In order to check the reachability of each state in the model and to ensure the absence of deadlocks, a model checking is done using the Uppal verifier 20. The verifier can do a wide variety of things, such as checking if all the model's states are not deadlocked or a specific state is reachable based on defined constrains. Figure 20 shows the verifying process.



```
A[]not deadlock                                                                                                          🟢
E<> Device.CheckingConditions                                                                                            🟢
E<> automationActive==false && (doAutomation.SetingStatusFirst || doAutomation.SetingStatusSecond ||  doAutomation.SetingStatusThird)   🔴
E<> deviceStatus==1 && doAutomation.DeviceThirdManuallyControlled                                                        🟢
E<> deviceManuallyControlled==true && (doAutomation.SetingStatusFirst || doAutomation.SetingStatusSecond ||  doAutomation.SetingStatusThird)   🔴
```

Figure 20: Uppaal verifier

In figure 20 5 quieries are made to check the model states as follows.

- The first query A [ ] not deadlock' is true if (and only if) every reachable state satisfy that there are no deadlocks. In other words, check if all states in the model has no logical conflicts.
- The query E <> Device.CheckingConditions represents a reachability query. The query is true if the system able to provide a path where CheckingConditions state is reachable.
- E <> deviceStatus==1 && doAutomation.DeviceThirdManuallyControlled query verify that there is a path where the the deviceStatus variable is equal to 'true' and the 'DeviceThirdManuallyControlled' state is reachable. That means there is no conflicts in verfying each parameter separately.
- E <> automationActive==false && doAutomation.SetingStatusFirst is true if the system provides a path where the 'automationActive' variable is equals to 'false' and the 'settingDeviceStatus' state is reachable in one of the three parameters. As we can see in the figure, the result of the query is false. This means if the the automation is not active, the system will not set the device status.
- E <> deviceManuallyControlled==true && doAutomation.SetingStatusFirst is true if the system can provides a path where the deviceManuallyControlled variable equals to 'true' and the 'settingDeviceStatus' state is reachable in one of the three parameters. the result of this query is false. This means if the the device manually controlled, the system will not set the device status.

As a result of the model verification, the doAutomation activity states are reachable and the flow is none deadlocked. More about the uppaal model in the 'uppaal.xml' included with the report's files.

8.3 Automation server API endpoints Integration tests

Integration testing ensures that system components work together correctly as a group. After system units have been tested, integration tests are conducted. That in purpose of verifying that Automation Server API endpoints and Hazelcast clusters work correctly, see section 8.3.1

8.3.1 interaction

This section illustrates how automation server API endpoints are tested. By interacting with Automation server API endpoints, applications and users can perform operations on the system's models or obtain information from it. Listing 10 shows testing the interaction with a server's endpoint for adding a new sensor instance to Automation Server models.

Listing 10: integration test for adding a new sensor to Automation server

```
        @Test
    void addSensor() throws Exception {

```

```java
      String requestBody = "{\"databaseName\":\"test\",\"sensorId\":" + 10 + ",\"devId\":\"testDevId\"" + ",\"name\":\"testSensor\"" + " }";

      MvcResult mvcResult = mockMvc
        .perform(
          MockMvcRequestBuilders.post("http://localhost:8080/addSensor").header("authorization", "Bearer" + " " + token).content(requestBody)).andReturn();

      // assert that the request reaches the server
      Assert.assertEquals(mvcResult.getResponse().getStatus(), 200);
      // assert the sensor is existing in the registry
      Assert.assertNotEquals(sensorsRegistry.getSensor("testDevId"), null);
    }
```

As shown in listing 10, the test uses mockMvc object to perform a request to the server endpoint "http://localhost:8080/addSen
In order to access the Automation server, the request headers must contain a valid access token. The valid token is obtained manually using POSTMan and sat manually as String value in the code. The test check if the response is 200 http response and a sensor object instance is created and added to the system registry.

Refer to section 10.3.1 in the 'Appendix' to see integration tests for the other endpoints.

9 Conclusion

The developed software system, by Adopting the self-organized automation concept adds a value point to delivering a better user experience in managing smart homes automation. The possibility to manage an automation process based on sensors/devices of the user choice addresses the main issue was represented at the the start of the project.

The technologies used in the implementation of the system achieved its goal in reflecting the models created in analysis and design levels. Furthermore, the tests conducted through the implementation process was effective in showing how the software components collaborating in order to achieve a singular, organised and disciplined behavior.

Time constraints limited the ability to measure all quality attributes captured as specified requirements. Anyway addressing all related quality attributes was not one of the main objective of the project at its first stage. Otherwise, the behavioral, structural design and tests conducted during the implementation process suggest that the most of the quality attributes are satisfied. In other words, the scalable structure modeled through the design level was manually tested many times before continuing to the next stage of the project.

Analysing design and model checking on the doAutomation activity showed how the development process in its all phases was directed in the iterative rhythm. In other words, the many changes happens through the project phases on the logical behavior of the doAutomation activity, it was easy as a developer to adopt the new changes on all related parts of the system.

The implementation process was challenging in order to find a balance with other parts of the development process, as implementing three separate services using different programming languages and different implementation platforms requires good experience. In addition, the implementation process consumed the most of time comparing with other parts of the development process.

References

[1] CyberlinkASP. What is software scalability and why is it important? - cyberlinkasp. https://www.reddit.com, Sep 2021.

[2] TechTarget . What is the internet of things (IoT)? . https://www.techtarget.com/iotagenda/definition/Internet-of-Things-IoT, 2023.

[3] Google Homeassistant . Automating Home Assistant. . https://www.home-assistant.io/getting-started/automation/, 2023.

[4] United Nations .12-responsible-consumption-and-production. . https://www.globalgoals.org/goals/12-responsible-consumption-and-production/, 2023.

[5] Tim Barenscheer. What is the difference between agile and unified process methodology? =https://www.teamly.com/blog/difference-between-agile-and-unified-process-methodology/, Feb 2022.

[6] TechTarget . What is a sensor? . . https://www.techtarget.com/whatis/definition/sensor, 2023.

[7] TechTarget . what is automation? . . https://www.techtarget.com/iotagenda/feature/Using-an-IoT-gateway-to-connect-the-Things-to-the-cloud, 2023.

[8] CyberlinkASP. What is software scalability and why is it important? - cyberlinkasp. https://www.cyberlinkasp.com/insights/what-is-software-scalability-and-why-is-it-important, Sep 2021.

[9] Azure Devops. Git rpositry for the developed system. https://dev.azure.com/namou19/namou19/_git/AfgangsProjekt.

[10] HiveMqtt - Official Website. MQTT Topics, Wildcards, Best Practices - MQTT Essentials: Part 5. www.hivemq.com/blog/mqtt-essentials-part-5-mqtt-topics-best-practices/.

[11] The things stack - Official Website. . https://console.cloud.thethings.network/, 2023.

[12] UPPaal - Official Website. . https://console.cloud.thethings.network/.

10 Appendix

10.1 Usecases table

ID	**Usecase Name**	**Actor**	**Parent Usecase**	**Priority**
UC01	Manage sensors	User,Admin		High
UC02	Manage devices	User,Admin		High
UC03	Manage automations	User,Admin		High
UC04	Add *LoRaWAN* sensor	User,Admin	UC1	High
UC05	Remove sensor	User,Admin	UC1	High
UC06	Add device	User,Admin	UC2	High
UC07	Remove device	User,Admin	UC2	High
UC08	Add automation process	User,Admin	UC3	High
UC09	Edit automation process	User,Admin	UC3	High
UC10	Remove automation process	User,Admin	UC3	High
UC11	Set energy source	User,Admin	UC10	Medium
UC13	Set automation sensor	User,Admin	UC10	High
UC14	Set automation device	User,Admin	UC10	High
UC15	Set automation status	User,Admin	UC10	High
UC16	Add automation parameters	User,Admin	UC10	High
UC17	Set device startTime	User,Admin	UC14	High
UC18	Set device offTime	User,Admin	UC14	High
UC20	Choose activeTime	User,Admin	UC14	High
UC21	Choose no time limitation	User,Admin	UC14	Medium
UC22	Set parameterTitle	User,Admin	UC16	High
UC23	Choose parameterLogic	User,Admin	UC16	High
UC24	Choose parameter related device	User,Admin	UC16	High
UC25	Edit automation energy source	User,Admin	UC09	Medium
UC26	Edit automation sensor	User,Admin	UC09	High
UC27	Edit automation Title	User,Admin	UC09	High
UC28	Edit automation device	User,Admin	UC09	High
UC29	Edit automation state	User,Admin	UC09	High
UC30	Edit automation parameter	User,Admin	UC09	High
UC31	Remove sensor from automation	User,Admin	UC26	High
UC32	Add sensor to automation	User,Admin	UC26	High
UC33	Remove device from automation	User,Admin	UC28	High
UC34	Add device to automation	User,Admin	UC28	High
UC35	Edit parameterTitle	User,Admin	UC30	High
UC36	Edit parameterLogic	User,Admin	UC30	High
UC37	Edit parameter related device	User,Admin	UC30	High
UC38	Sign in	User,Admin		High
UC39	Sign out	User,Admin		High
UC40	Manage rooms	User,Admin		High
UC41	Add rome	User,Admin	UC40	High
UC42	Show automations report	User,Admin		High
UC43	Show active connected sensor data	User,Admin	UC42	High
UC44	Show all automation processes status	User,Admin	UC42	High
UC45	Show connected devices	User,Admin	UC42	High
UC46	Show statistics	User,Admin		Medium
UC47	Show energy consuming by room	User,Admin	UC46	Medium

UC48	Show actual energy production	User,Admin	UC46	Medium
UC49	Show energy source battery level	User,Admin	UC46	Medium
UC50	Turn on device	User,Admin		Medium
UC51	Turn off device	User,Admin		Medium
UC52	Manage users	Admin		High
UC53	Add user	Admin	UC52	High
UC54	Remove user	Admin	UC52	High
UC55	Define user access	Admin	UC52	High
UC56	Edit sensor	Admin	UC52	High
UC57	Edit device	Admin	UC52	High
UC58	Show connected devices status	User,Admin	UC42	High

Table 5: Use cases priority list.

10.2 units tests

Listing 11: UnitTests.java class

```
    package afgangsProjekt.automation.domain.integrationTest;

import afgangsProjekt.automation.domain.AutomationHandler;
import afgangsProjekt.automation.domain.AutomationLogic;
import afgangsProjekt.automation.domain.mocks.SystemModelsMock;
import afgangsProjekt.automation.systemEnums.AutomationStatus;
import afgangsProjekt.automation.systemEnums.DeviceStatus;
import afgangsProjekt.automation.systemEnums.ParameterStatus;
import afgangsProjekt.automation.systemModels.Automation;
import afgangsProjekt.automation.systemModels.Parameter;
import afgangsProjekt.automation.systemModels.Sensor;
import afgangsProjekt.automation.systemRigestry.AutomationRegistry;
import afgangsProjekt.automation.systemRigestry.DevicesRegistry;
import afgangsProjekt.automation.systemRigestry.ParametersRegistry;
import afgangsProjekt.automation.systemRigestry.SensorsRegistry;
import org.json.JSONException;
import org.json.JSONObject;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit.jupiter.SpringExtension;

import java.text.ParseException;

import static java.lang.Thread.sleep;

@ExtendWith(SpringExtension.class)
@SpringBootTest
class UnitTests {

    @Autowired
    DevicesRegistry devicesRegistry;
    @Autowired
    SensorsRegistry sensorsRegistry;
    @Autowired
```

```java
    AutomationRegistry automationRegistry;
    @Autowired
    ParametersRegistry parametersRegistry;
    @Autowired
    AutomationHandler automationHandler;



//The following test ensure that the system components interact successfully to
    perform an automation process
// The test assumes that a sensor object with valid data, a device not connected
    to its communication provider's server
// , an automation process with valid timeRange, and a parameter  has initialized
    by the system.

    @BeforeEach
    void initializeSystemProperties() {
        SystemModelsMock.initialize();

    }


    @Test
    void doAutomation() throws ParseException, InterruptedException {

        devicesRegistry.addDevice("test", SystemModelsMock.getDevice());
        sensorsRegistry.addSensor("test", SystemModelsMock.getSensor());
        parametersRegistry.addParameter("test", SystemModelsMock.getParameter());
        automationRegistry.addAutomation("test", SystemModelsMock.getAutomation())
            ;
        automationHandler.doAutomation();
        Assertions.assertEquals(DeviceStatus.communicationProviderDown,
            SystemModelsMock.getDevice().getStatus());
    }

    @Test
     void DataSatisfyTimeConditions() throws ParseException,  JSONException {
        // first situation the device has startTime on 22/11/2022 10:00 o'clock
            and offTime on 29/11/2023 10:30
        Automation automation1 = new Automation(1,"test", AutomationStatus.activ,"
            {\"start\":\"22-11-2022-10-00\",\"off\":\"29-11-2023-10-30\"}");
        // the second situation the device has startTime on 22/11 10:00 o'clock
            and offTime on 23/11 10:30
        Automation automation2 = new Automation(1,"test", AutomationStatus.activ,"
            {\"start\":\"22-11-2022-10-00\",\"off\":\"23-11-2022-10-30\"}");
        // expected result first situation time is satisfied
        Assertions.assertEquals(true, AutomationLogic.
            checkTimeConditionIsSatisfied(automation1));
        JSONObject jsonObject2 = new JSONObject();
        // expected result second situation time is not satisfied
        Assertions.assertEquals(false, AutomationLogic.
            checkTimeConditionIsSatisfied(automation2));

    }


    @Test
    void sensorDataSatisfyParameterValues() throws ParseException,
        InterruptedException {
```

```
        Sensor sensor = SystemModelsMock.getSensor();
        sensor.setLiveValue("10");
        Parameter parameter = SystemModelsMock.getParameter();
        parameter.setParameterLogic("onBetween");
        parameter.setParameterValue("{\"value1\":\"5\",\"value2\":\"15\"}");
        sensorsRegistry.addSensor("test", sensor);
        parametersRegistry.addParameter("test", parameter);
        doAutomation();
        //Check that the sensor value is between the two values defined by a user
        Assertions.assertEquals(ParameterStatus.satisfied,parameter.
            getParameterStatus());
        // Change the sensor's live data value to not satsfying the values defined
             by a user
        sensor.setLiveValue("20");
        doAutomation();
        Assertions.assertEquals(ParameterStatus.unsatisfied,parameter.
            getParameterStatus());
    }

    @Test
     void testGetActiveHours() throws SQLException {

        String[] dataRow = new String[6];
        ArrayList<String[]> dayData= new ArrayList<>();
        HashMap<String,ArrayList<Integer>> expectedActiveHoursByDay= new HashMap<
            String,ArrayList<Integer>>(){{
            put("10-10-2022",new ArrayList<>(Arrays.asList(10,13)));
            put("11-10-2022",new ArrayList<>(Arrays.asList(10,11,12)));
            put("12-10-2022",new ArrayList<>(Arrays.asList(10,14)));
        }} ;
        // System.out.println(expectedActiveHoursByDay);
        HashMap<String, ArrayList<String[]>> daysMap = new HashMap<>();
        int idCounter=0;
        int hoursTestRange=5;
        int daysTestRange=3;
        int minutesTestRange=1;
        int propelatyRange=(int) ( Math.random())  ;
        for (int day = 0; day < daysTestRange; day++) {
            for (int hour = 0; hour < hoursTestRange; hour++) {
                for (int minute = 10; minute < 60; minute += minutesTestRange) {
                    idCounter++;
                    String dayDate="1" + String.valueOf(day) + "-10-2022";
                    String hourTime="1"+String.valueOf(hour) +"-"+ String.valueOf(
                        minute)+"-00";
                    AtomicReference<String> sensorValue = new AtomicReference<>("0
                        ");
                    dataRow[0] = String.valueOf(idCounter);
                    dataRow[1] = "kitchen";
                    dataRow[2] = "occupancySensor";
                    dataRow[3] = dayDate;
                    dataRow[4] = hourTime;
                    String[] finalDataRow = dataRow;
                    String compareHour = "1"+String.valueOf( hour);
                    expectedActiveHoursByDay.forEach((d, activeHoursList)->{
                        // System.out.println(finalHour);
                        //  System.out.println(dayDate + " " +d);
                        // System.out.println(compareHour + " "+activeHoursList);
                        // System.out.println(d.equals(dayDate)&activeHoursList.
                            contains(compareHour));
                        if(d.equals(dayDate)&activeHoursList.contains(Integer.
                            valueOf(compareHour))){

                            sensorValue.set(String.valueOf((((int) (10 * Math.
                                random()) & 1)|((int) (10 * Math.random()) & 1))));
                        }
                    });
                    dataRow[5]=sensorValue.get();
                    // System.out.println(finalDataRow);

                    // System.out.println(idCounter);
                    dayData.add(dataRow);
                    dataRow=new String[6];
                }



            }


        }


      //  Assertions.assertEquals(expectedActiveHoursByDay, ActiveTimeLogic.
            getActiveHours( dayData));

        Assertions.assertEquals(expectedActiveHoursByDay.values(), ActiveTimeLogic
            .getActiveHours(dayData).values());


    }

    @Test
     void testSensorValueChanged() throws InterruptedException {
        Sensor sensor = SystemPropertiesMock.getSensor();
        System.out.println(sensor.getLiveValue());
        sensor.setLiveValue("20");
        SensorValueListener.updateSensorValue(sensor,"19");

        Assertions.assertEquals(true,SensorValueListener.isValueChanged(sensor));
        sensor.setLiveValue("20");
        SensorValueListener.updateSensorValue(sensor,"20");

        Assertions.assertEquals(false,SensorValueListener.isValueChanged(sensor));
        System.out.println("sasa");
    }


}
```

10.3 IntegrationTest

10.3.1 API interactions

Listing 12: Integrationtest.java

```

```

```java
package afgangsProjekt.automation.domain.integrationTest;

import afgangsProjekt.automation.domain.WheatherApiConnection;
import afgangsProjekt.automation.domain.mocks.SystemModelsMock;
import afgangsProjekt.automation.systemEnums.DeviceStatus;
import afgangsProjekt.automation.systemModels.Device;
import afgangsProjekt.automation.systemModels.Sensor;
import afgangsProjekt.automation.systemRigestry.AutomationRegistry;
import afgangsProjekt.automation.systemRigestry.DevicesRegistry;
import afgangsProjekt.automation.systemRigestry.ParametersRegistry;
import afgangsProjekt.automation.systemRigestry.SensorsRegistry;
import com.google.gson.Gson;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.
    AutoConfigureMockMvc;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit.jupiter.SpringExtension;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.MvcResult;
import org.springframework.test.web.servlet.request.MockMvcRequestBuilders;
import org.springframework.web.context.WebApplicationContext;
import org.testng.Assert;


@ExtendWith(SpringExtension.class)
@SpringBootTest
@AutoConfigureMockMvc
public class RestControllerIntegration {
    String token="
        eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Ii1LSTNROW5OUjdiUm9meG1lWm9YcWJIWkdldy
        ";
    @Autowired
    DevicesRegistry devicesRegistry;
    @Autowired
    SensorsRegistry sensorsRegistry;
    @Autowired
    AutomationRegistry automationRegistry;
    @Autowired
    ParametersRegistry parametersRegistry;
    @Autowired
    MockMvc mockMvc;
    @Autowired
    WebApplicationContext webApplicationContext;




    @BeforeEach
    void initializeSystem(){
        SystemModelsMock.initialize();
        System.out.println("Mocks initialized");
        devicesRegistry.addDevice("test", SystemModelsMock.getDevice());
        sensorsRegistry.addSensor("test", SystemModelsMock.getSensor());
        parametersRegistry.addParameter("test", SystemModelsMock.getParameter());
        automationRegistry.addAutomation("test", SystemModelsMock.getAutomation())
            ;
    }


    @Test
    void addSensor() throws Exception {

        String requestBody = "{\"databaseName\":\"test\",\"sensorId\":"+10+",\"
            devId\":\"testDevId\""+",\"name\":\"testSensor\""+ " }";

        MvcResult mvcResult = mockMvc
                .perform(
                        MockMvcRequestBuilders.post("http://localhost:8080/
                            addSensor").header("authorization","Bearer"+" "+token).
                            content(requestBody)).andReturn();

        // assert that the request reaches the server
        Assert.assertEquals( mvcResult.getResponse().getStatus(),200);
        // assert the sensor is existing in the registry
        Assert.assertNotEquals(sensorsRegistry.getSensor("testDevId"),null);
    }

@Test
void removeSensor() throws Exception {

//get the Mocked sensor
    Sensor sensor=sensorsRegistry.getAllSensors("test").get("sensors").get(0);

    String devId=sensor.getdev_Id();
    String requestBody = "{\"databaseName\":\"test\",\"devId\":\""+devId+"\"}";

    MvcResult mvcResult = mockMvc
            .perform(
                    MockMvcRequestBuilders.post("http://localhost:8080/
                        deleteSensor").header("authorization","Bearer"+" "+token).
                        content(requestBody)).andReturn();

    // assert that the request reaches the server
    Assert.assertEquals( mvcResult.getResponse().getStatus(),200);
    // assert the sensor is removed from the registry
Assert.assertEquals(sensorsRegistry.getAllSensors("test").get("sensors").contains(
    sensor),false);
}
//this test ensures changing device status manually

    @Test
    void stopDeviceManually() throws Exception {
        String requestBody = "{\"databaseName\":\"test\"}";

        Device device = devicesRegistry.getallDevices("test").get("devices").get
            (0);
        //set the device status to initial
        device.setStatus(DeviceStatus.initial);
        MvcResult mvcResult = mockMvc
                .perform(
                        MockMvcRequestBuilders.post("http://localhost:8080/
                            stopDevice/"+device.getDeviceId()).header("
                            authorization","Bearer"+" "+token).content(requestBody)
                            ).andReturn();
        // assert that the request reaches the server
        Assert.assertEquals( mvcResult.getResponse().getStatus(),200);
        // assert that deviceStatus for device with id 1 is changed
        Assert.assertNotEquals(device.getStatus(),DeviceStatus.initial);
    }

    @Test
    void startDeviceManually() throws Exception {
        String requestBody = "{\"databaseName\":\"test\"}";
        Device device = devicesRegistry.getallDevices("test").get("devices").get
            (0);
        device.setStatus(DeviceStatus.initial);
        MvcResult mvcResult = mockMvc
                .perform(
                        MockMvcRequestBuilders.post("http://localhost:8080/
                            startDevice/1").header("authorization","Bearer"+" "+
                            token).content(requestBody)).andReturn();
        // assert that the request reaches the server
        Assert.assertEquals( mvcResult.getResponse().getStatus(),200);
        // assert that deviceStatus for device with id 1 is changed
        Assert.assertNotEquals(device.getStatus(),DeviceStatus.initial);
    }

    @Test
    void stopManualControl() throws Exception {
       //Getting the mock device created within beforeEach method
        Device device = devicesRegistry.getallDevices("test").get("devices").get
            (1);
       //Setting device status to manuallyStarted
        device.setStatus(DeviceStatus.manuallyStartted);
        MvcResult mvcResult = mockMvc
                .perform(
                        MockMvcRequestBuilders.post("http://localhost:8080/
                            stopManualControl/1").header("authorization","Bearer"+"
                             "+token)).andReturn();
        // assert that the request reaches the server
        Assert.assertEquals( mvcResult.getResponse().getStatus(),200);
        // assert that deviceStatus for device is not the same as before the
            request
        Assert.assertNotEquals(device.getStatus(),DeviceStatus.manuallyStartted);


    }
    @Test
    void getSystemInfo() throws Exception {
        Gson gson = new Gson();

        String sensors = gson.toJson(sensorsRegistry.getAllSensors("test"));
        String devices = gson.toJson(devicesRegistry.getallDevices("test"));
        String wheather = gson.toJson(WheatherApiConnection.getWeathrtData());
        MvcResult mvcResult = mockMvc
                .perform(
                        MockMvcRequestBuilders.get("http://localhost:8080/
                            getAllInfo/test").header("authorization","Bearer"+" "+
                            token)).andReturn();

        System.out.println("fromTest"+ wheather);
        Assert.assertEquals( mvcResult.getResponse().getStatus(),200);

        Assert.assertEquals( mvcResult.getResponse().getContentAsString(),"["+
            devices+","+ sensors +","+wheather+"]");

    }


}
```

10.4 Environment.java class

Listing 13: Environment.java class

```
@Component("invirnoment")
    public class Environment {
  EnvironmentStatus environmentStatus;

  public Environment() {
    try {

      if (System.getenv("env").equals("development")) {
        environmentStatus = EnvironmentStatus.development;
      } else if (System.getenv("env").equals("production")) {
        environmentStatus = EnvironmentStatus.production;
      } else if (System.getenv("env").equals("testing")) {
        environmentStatus = EnvironmentStatus.testing;
      }
    } catch (Exception e) {
      System.out.println(e.getMessage());
      environmentStatus = EnvironmentStatus.development;

    }
  }
  public EnvironmentStatus getEnvironmentStatus() {
    return environmentStatus;
  }

}
```

11 SystemModelsMock.java class

```
     static Sensor sensor;
   static Device device;
   static Automation automation;
   static Parameter parameter;

    public static void setSensor(Sensor sensor) {
        SystemModelsMock.sensor = sensor;
    }

    public static void setDevice(Device device) {
        SystemModelsMock.device = device;
    }

    public static void setAutomation(Automation automation) {
```

```java
            SystemModelsMock.automation = automation;
    }

    public static void setParameter(Parameter parameter) {
        SystemModelsMock.parameter = parameter;
    }

    public static Sensor getSensor() {
        return sensor;
    }

    public static Device getDevice() {
        return device;
    }

    public static Automation getAutomation() {
        return automation;
    }

    public static Parameter getParameter() {
        return parameter;
    }



    public static void initialize(){

         sensor = new Sensor(1,"testSensor","19", SensorStatus.activ,"DEVIDSSSSSD"
             );
         device = new Device(1,"rest2", DeviceStatus.initial,"testDevice");
         automation = new Automation(
                1,
                "testAutomation",
                AutomationStatus.activ,
                "{\"start\":"+"\"10-11-2022-14-49\","+"\"off\":"+"
                   \"04-12-2022-14-53\"}"  );
        String parameter1Logic = "onBetween";
        String parameter2Logic = "onOnly";
        String parameter1Value = "{\"value1\":\"18\",\"value2\":\"21\"}";
        String parameter2Value = "20";
        String sensorValue = "19";
         parameter = ( new Parameter(
                1,
                sensor,
                device,
                automation,
                parameter1Logic,
                parameter1Value,
                ParameterStatus.satisfied));

    }

}
```