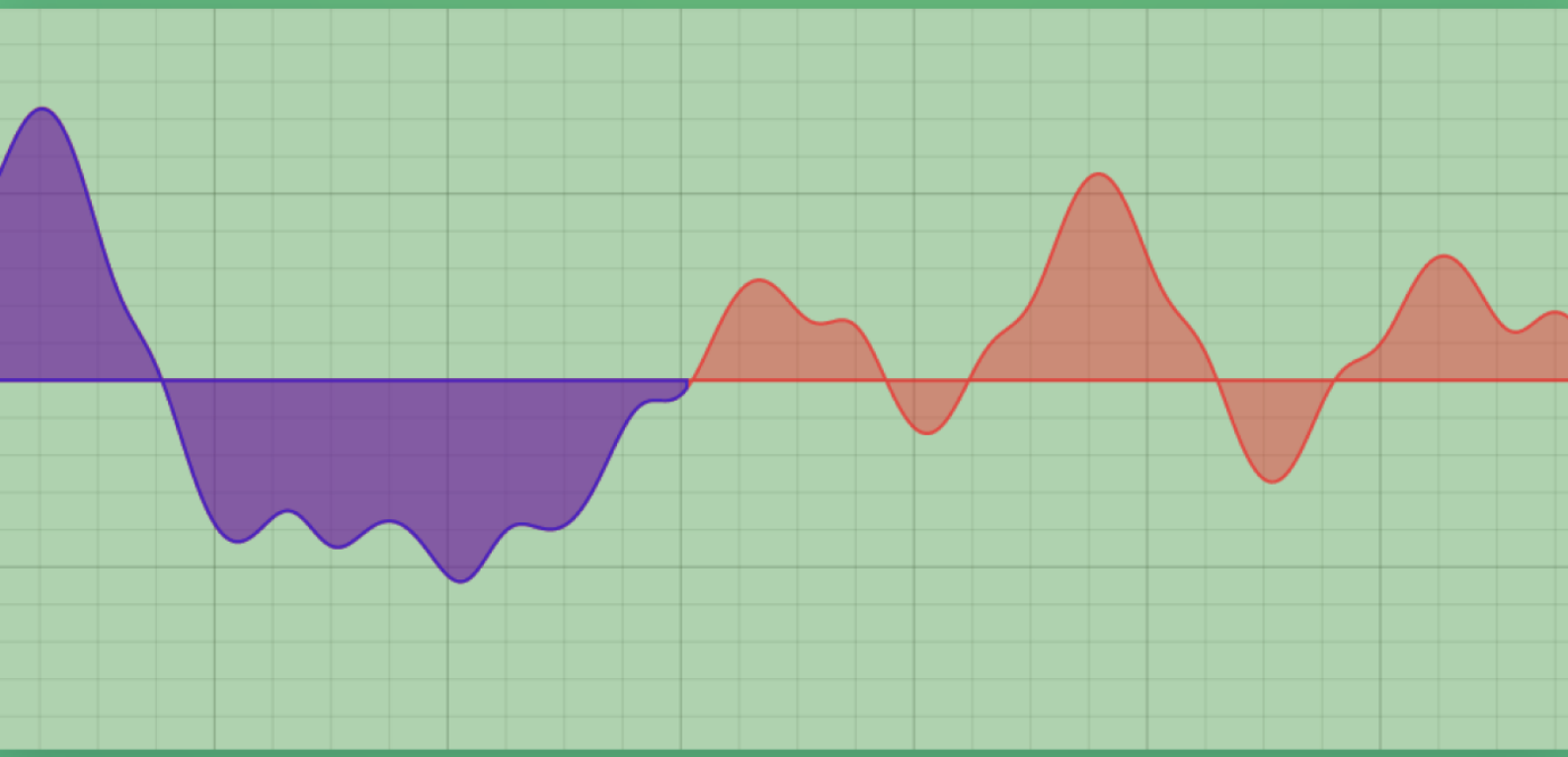
IoT Based
Energy Utilization Bot

Spring Semester 2020 – P2 Project by DAT2 C1-13 – Computer Science – Aalborg University



AALBORG UNIVERSITY
DENMARK

A REPORT BY DAT2 C1-13
AALBORG

AALBORG UNIVERSITET
STUDENTERRAPPORT

2nd Semester
Computer Science
Strandvejen 12-14
9000 Aalborg
http://www.cs.aau.dk

Title:

IoT Based Energy Utilization Bot

Project:

P2 project

Project Period:

Spring Semester 2020

Project Group:

DAT2 C1-13

Participants:

Andreas Sebastian Sørensen
Emil Gybel Henriksen
Niels Ulrik Gajhede
Sarah Mølleskov Frandsen
Theodor Risager

Supervisor:

Ramoni Ojekunle Adeogun

Page Numbers: 89

Date of Completion: 27-05-2020

Abstract:

This report investigates the development of a web application for managing power through an IoT network. Motivated by the benefits of moving towards renewable energy and improving the efficiency of the energy grid, the group decided on developing an application that can schedule IoT appliances for improved renewable energy utilization. Through thorough analysis of different approaches, it was found that a centralized system would be ideal for implementation. Before developing the implementation, a model of the system was designed, which includes device simulation, server architecture and information flow.
The developed software uses simulated data to schedule water heaters to run when there is a surplus of renewable energy, and turns them off when there is a deficit.
Based on the tendencies seen in the results of the final product, it can be concluded that if a system like this was implemented, it would be able to help stabilize the energy grid.

Preface

This is the report for the P2 Project from Aalborg University, made by group DAT2 C1-13; a group of Computer Science students on their second semester. The overall theme for the project is "A larger program developed by a group". The goal of the project is to make a "large" program of high quality that solves a problem deduced from the problem analysis.

This report uses the “Vancouver method” for references, meaning for every reference, a number is placed in the text. This number refers to the section called “References”. The list of references is sorted by the occurrence of the source in the report.

At the end of this report, a glossary can be found.

A REPORT BY DAT2 C1-13
AALBORG

Contents

Chapter 1: Introduction

Due to the increasing temperature in the earth's atmosphere [1], there has been an increased focus on global warming and its influence on our environment, as can be seen by the global demonstrations that took place in September 2019, where an estimated 185 countries took part.[2] Therefore, the electricity business needs to adjust its production and technologies. A popular solution for producing electricity without releasing carbon dioxide is through Renewable Energy (RE) sources.[3]

However, many RE-producing technologies rely on phenomenons that are not directly controllable by humans, such as weather conditions. This means that when there is an energy demand, it is not necessarily possible to produce enough RE to cover the demand.

The main objective for this report is to see if a centralized Internet of Things (IoT) network can help move energy consumption to when RE is produced and thereby help stabilize the energy grid. IoT is a network of devices all interconnected through the Internet, and the devices are therefore able to communicate without human interference.[4] An application that utilizes this quality will be made in order to help stabilize the energy grid by heating up water heaters at times where there is an RE surplus. The remaining part of the report is organized as follows:

Chapter 2 contains the problem analysis, in which it is discussed which approach to take to help stabilize the energy grid. In this chapter, it is defined what an appliance is, and the program requirements for the application have also been formulated.

Chapter 3 shows the modeling and design of the application. This chapter presents the application structure and showcases how the different parts of the application should communicate with each other.

Chapter 4 describes the implementation of the designed models. This chapter includes code snippets from the implementation to show interesting and advanced sections of the code.

Chapter 5 is the discussion. In this chapter, the project as a whole will be discussed; this means there will be a discussion of the results of the project as well as some of the sources of error.

Chapter 6 contains the conclusion where the problem statement is concluded on.

Chapter 2: Problem Analysis

2.1 Problem Domain

In 2050, Denmark aims to have all its energy supply come from Renewable Energy (RE) sources, and already by 2030, it is expected that 100% of the electricity supply comes from RE alone. To do this, the electricity business needs to adjust. However, a solution to achieve this goal has not yet been found and will require new technologies.[5, p2]

Many RE-producing technologies rely on phenomenons that are not directly controllable by humans, such as weather conditions. Popular examples of these technologies are wind turbines and solar cells, which rely on the wind and the sun, respectively.[6, p958] RE sources lead to a fluctuating electricity production, where the amount of produced electricity is difficult to balance with the current energy consumption.

Energy production is not the only area that will challenge the electricity networks in the future: Because of the recently increased focus on climate changes, the car industry is changing the way cars are fueled, replacing gas with batteries driven by electricity.[5, p3] Cars are used as an easily accessible transportation tool at any given time of day, for an unknown amount of time. Because of this usage, cars must be able to charge quickly and contain enough energy to transport the vehicle for long distances at all times.[7, p9]

During the weekdays, most cars will be used to drive to and from work. A large part of the population goes to work at roughly the same time, meaning that a lot of cars will start charging simultaneously. This will require a lot of energy production.[5, p6] The demand for high energy production at specific times conflicts with RE production, because it does not provide the needed flexibility.

A way to better balance energy production and consumption is an Internet of Things (IoT) network. IoT networks connect devices to the Internet or another network so that they can communicate with each other. This technology makes it possible to adjust devices from a centralized system, e.g turning on your coffee machine from your computer.[4]

A way to solve Denmark's energy problem could be to create an IoT network that adjusts energy consumption by turning devices on and off in regards to the current RE production. This means that when there is a surplus of RE, the devices will be turned on to utilize the surplus in energy.

2.2 Problem Delimitation

The problem of making an energy utilization platform discussed in the previous section, Section 2.1, is rather broad considering the differences in energy demands at various levels. Addressing the entire domain may require complex solutions, which are beyond the scope of this project. The time limited nature of the semester project requires a careful delimitation of the problem via the problem analysis. In the delimitation process, the scope is narrowed down; this process is illustrated in Figure 2.1. The goal of this section is to showcase the process leading to the specific problem choice made before ultimately formulating a problem statement.

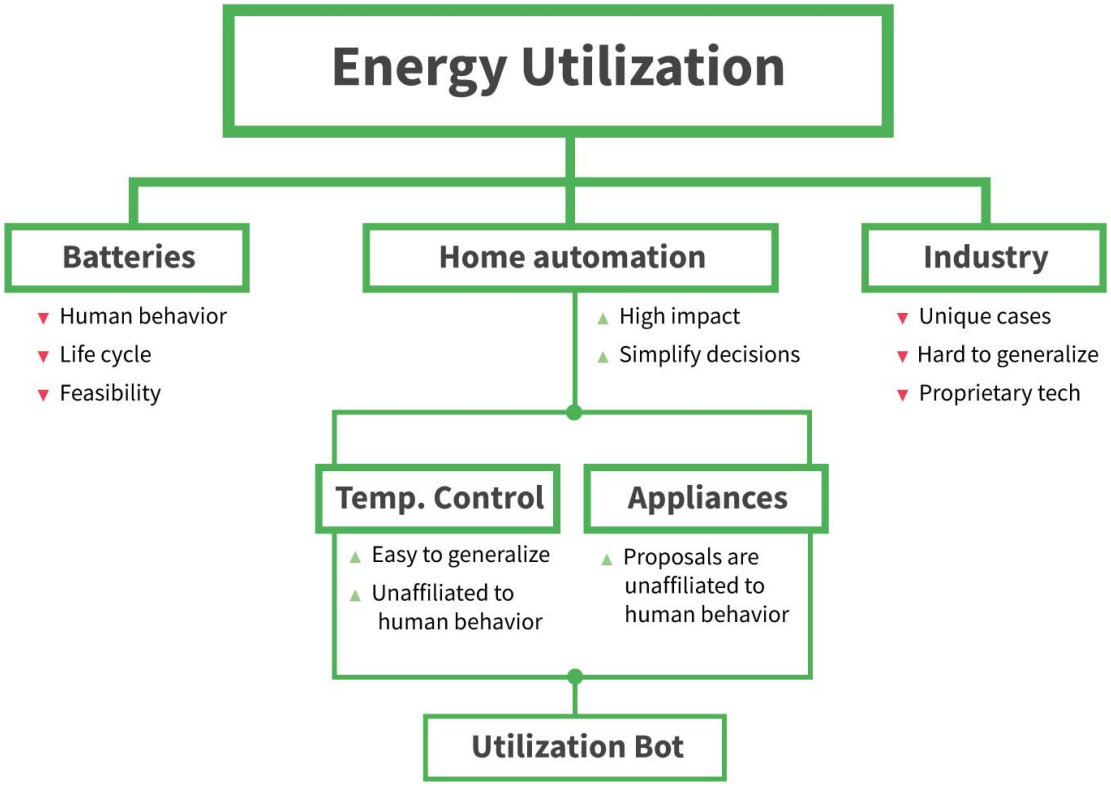


Figure 2.1: This figure shows the delimitation progress of the project. It clarifies which problems have been picked out for further investigation and immersion, and which problems have been considered out of scope. Two problems are combined into one, creating a clear pathway for the rest of the report.

2.2.1 Batteries

This section will cover the different options related to power storage through the use of standalone batteries and the Vehicle to Grid method.

Standalone batteries

Batteries have potential to stabilize the power grid through storing electric energy with a small power loss during the transfer. However, mass manufacture of modern batteries produces several toxic chemical compounds, which can harm the environment.[8, s1] This environmental impact can be lessened from new non-toxic chemical compounds in the production, or from the production of batteries with a larger capacity.[8, s1]

Vehicle to Grid

Another way of producing batteries specific for the power grid, is to use the large amount of batteries that are already in use from other products, namely those from Electric Vehicles (EVs). This approach is called "Vehicle to Grid", as the grid will be able to use the large amount of electric storage capacity found in the EVs. The Vehicle to Grid method works by recharging the batteries of inactive vehicles when the demand for electricity is lower than the production, and discharging the batteries of inactive vehicles once the demand for electricity is higher than the production. However, this can prematurely wear down the batteries and reduce the capacity quicker than a normal use of the battery would, leading to faster replacement and higher upkeep costs for the vehicle owner.[9, s5.2.2] Another factor to take into consideration when wanting to implement this method is the EV owners' willingness to go below a certain power threshold, as this could limit the EV owners' mobility on short notice. According to a survey, this is the largest concern for the EV owners, but other concerns may also be relevant to look into when working with this method.[7, p9]

In conclusion, the topics of "Batteries" and "Vehicle to Grid" are full of research and peer-reviewed articles that are easily accessible, since it has been a popular topic in recent years. However, it has been decided to focus on another solution to improve grid stability, due to issues with battery life cycle and the feasibility of producing a proper solution. Therefore, these topics will be considered outside of the project scope.

2.2.2 Industry

When looking at the problem scope, the industrial sector seems like an obvious place to start when wanting to balance out energy production and consumption, as the industry sector's power consumption levels are quite high [10]. However, it is a very broad sector to look at, which means that it will be difficult to find generalized data on this topic, such as how much energy different companies use, how much money they spend on energy, etc. Even if such data exists and is available to the public, this would only cover one instance, meaning it cannot be used to generalize data for the entire industrial sector. In short, every company's case is unique, which is why the problem of stabilizing the energy grid in terms of balancing out companies' energy consumption is not solvable by gathering data from one case.

It is concluded that the industrial sector is very broad and consists of many unique cases, and because of the high level of uniqueness, it will be difficult to analyze this sector and thereby optimize the energy grid with a software-based solution.

2.2.3 Home Automation

According to safewise.com, home automation is *"a step toward what is referred to as the "Internet of Things," in which everything has an assigned IP address, and can be monitored and accessed remotely."*[11] This means that all kinds of objects, from mirrors to entertainment and blinds, are connected on a central network where everything can be accessed from the home owner's mobile device. The home owner is then able to control the connected objects and devices with a few taps on their mobile device, meaning they can control whether the blinds should be open or shut, whether the microwave should turn on, etc.

When researching the topic of home automation and the most popular IoT devices, it is seen that smart thermostats were one of the most frequently bought IoT devices on Amazon.com [12] in 2019. Smart thermostats, like the Nest Learning Thermostat, use the home owner's energy habits to better schedule when to heat up the house, and thereby decreasing the owner's energy bills.[13] These thermostats would be able to adjust the consumption to fit the RE surplus.

In Denmark, the energy production can be found investigated at energinet.dk[14], however, there is no easy recognizable way to figure out when it is a good choice to increase or lower a person's electricity consumption.

Due to the amount of decisions an average person has to make during a day, many people become overwhelmed[15]. This means that choosing when to turn on your devices can become difficult, and you will most likely end up turning your devices on during less optimal hours, since you have too little energy to research when the most optimal hours to consume energy are.

Therefore, after researching the home automation topic, this project found that a software system that would have large potential, would be a system that can guide and help people in a simple and time efficient way to adjust their electricity consumption. Such a system could help stabilize and lower the energy consumption without demanding too much of people's time and focus.

2.2.4 Temperature Control

The topic of temperature control covers all kinds of appliances that work with temperature in different ways. These kinds of appliances include freezers, water heaters and indoor climate control systems. At the time of writing this report, these appliances all try to keep the temperature at a constant level[16]. For example, an indoor climate control system would keep the temperature at a consistent 21 °C.

Instead of keeping a constant temperature, the temperature-regulating appliances could use power to adjust the temperature when there is an RE surplus and turn off completely when there is an RE deficit. However, if the temperature surpassed a certain threshold, such as the minimum temperature of a heating device, the appliances would turn on again. In other words, if you would like your indoor climate to be 21 °C, you might want to increase the temperature to 22 °C when there is a surplus of RE, and then let the temperature drop to a lower threshold at 20 °C when there is an RE deficit. However, you would start heating up your room again regardless of whether or not there is an RE surplus when it falls below

the minimum temperature of 20 °C.

The Danish building regulation states that all houses must have some type of installation that can heat up water[17]. Therefore, it would help stabilize the energy grid if it was possible to utilize this capacity.

The temperature-regulating appliances have been chosen to be a focus point in this project because of the large potential impact these may have on the energy grid. In Denmark, there are more than 1 000 000 single family homes[18], and this project assumes that each of them is going to have a 50 L water heater or one of similar size, because of the aforementioned Danish regulations[17]. After running some calculations, it is noted that if it is possible to utilize all of the 50 L water heaters in the 1 000 000 single family homes, it would give the system a potential of 581 MWh per 10 °C of variation.

This number is derived from the equation used to calculate the energy needed to heat water. Equation 2.1 calculates the needed energy for heating 50 kg of water 10 °C. Equation 2.2 converts joules into kWh. The last Equation 2.3 scales it up to the 1 000 000 households.

$$50\,\text{kg} \times \nabla 10\,\text{K} \times 4186\,\frac{\text{J}}{\text{kgK}} = 2\,093\,000\,\text{J} \tag{2.1}$$

$$2\,093\,000\,\text{J} \times \frac{1}{3600}\,\frac{\text{Wh}}{\text{J}} \times \frac{1}{1000}\,\frac{\text{kWh}}{\text{Wh}} \approx 0.581\,\text{kWh} \tag{2.2}$$

$$581\,\text{kWh} \times \frac{1}{1000}\,\frac{\text{MWh}}{\text{kWh}} \times 1\,000\,000\,\text{houses} = 581\,\text{MWh} \tag{2.3}$$

Equation 2.1 uses the equation $c = \frac{Q}{m \times \nabla T}$[19] to calculate the amount energy 50 L of water can store. Equation 2.2 converts Joule into kWh, and this number is then scaled up to accommodate the 1 000 000 households in Equation 2.3.

2.2.5 Appliances

Appliances are an important topic to research when trying to stabilize the energy grid, since they make up for a great part of the energy consumption in the average Danish household. Cooking and Washing devices alone make up for around 31 % of total residential energy consumption. [20] Therefore, it is key to look at how the appliances' energy usage can be optimized and help stabilize the grid.

Another interesting factor to look at when researching this topic is that while some appliances are manual and need human planning to run, others run automatically. This will be further discussed in section 2.3.2. An example of such automated appliances are robotic vacuum cleaners. Although not all home appliances are automated, it is possible that some might go from being manual to becoming automated in the near future, meaning appliances such as a washing machine could do its job without human intervention. If more home appliances become automated in the future, it would become easier to stabilize the energy grid, since there would be more appliances that would be able to move their "on-time" to times with RE surpluses.

It is seen that not all appliances use the same amount of power when running; some appliances use a lot of energy, some use less. The appliances that use the most power while running are fridges and other cooling appliances, washing machines and tumble dryers. [21] Additionally, many appliances also use a lot of energy when they are not turned on, or

when they are inactive; not performing their primary function. Therefore, it makes sense to look at high energy-consuming appliances when looking to stabilize the energy grid, since the more energy an appliance uses, the larger an impact it will have on the energy grid.

After reviewing the Appliances topic, it becomes clear that this topic combines well with the topic of Temperature Control, since they are very similar. Therefore, these two topics will be combined into one topic, containing the main focus points from both topics. The combined topic will be further analyzed in Section 2.3.

In conclusion, appliances are key to look at when balancing out the energy grid due to their large percentage wise stake in the average household's energy consumption. If more home appliances were automated, it would be easier to stabilize the energy grid via an IoT solution because of less human intervention. Furthermore, Danish households use on average 4450 kWh a year[22], which totals to 4450 GWh a year for all 1 000 000 single family homes[18]. 20 % of this energy is used on washing[20], which is 890 GWh, and if 1 % of this would be utilized, it would yield a potential of 8.9 GWh a year.

2.3 Smart Temperature Control and Appliances

This section will further analyze the two components Smart Temperature Control and Appliances. This section aims to outline the requirements for these components. Furthermore, the feasibility and the current state of the technology will be analyzed to better understand how they work and how they could be implemented to better utilize RE.

2.3.1 Smart Temperature Control

Smart temperature control means that instead of heating up water or a building and sustaining a specific temperature at all times, a smart system would intelligently increase or decrease the temperature depending on current and future power supply.

An example of a smart device could be a water heater. Normally, the water heater should be set to 55 °C[23], and it keeps this temperature at all times and therefore uses a consistent amount of energy to account for heat lost to the surroundings. A smart version of this water heater would be one that could account for the production of RE and heat up the water above the 55 °C when there is RE surplus. This increased temperature would act as a battery for when the surplus of RE declines. As the surplus in RE becomes a deficit, the smart water heater would turn off and wait until the temperature has dipped below the minimum of 55 °C or until the production of RE has increased, before turning on.

Water has a very high heat capacity[24], and because of this, a water heater would be able to store lots of energy by heating up the water. A 50 L water heater would be able to store 0.58 kWh by increasing the temperature of the water by 10 °C. Additionally, a water heater can heat up water to at least 75 °C[16] if not more in some cases, which would give it the potential to store up to 1.74 kWh.

This type of technology could be used to store excess RE production for times when RE is not being produced or for when the demand exceeds the production. This would help

stabilize the energy grid by moving some demand away from peak demand periods.

In a study made by researchers at the Technical University of Denmark in 2012, it was found that a building can be used to store energy in the form of thermal potential[25, p5-9]. This means that instead of passively heating up a building to a constant 21 °C, a smart building could use more energy to heat up the building in preparation of a shortage in energy production. When the shortage arrives, the building would then lower the energy consumption towards heating and wait for the temperature to become too low, or for the energy production to rise again, before consuming energy for heating.

This study shows that a smart heating system can be made and will be able to lower the strain on the energy grid at peak hours and be used as a battery for excess energy production.

The smart temperature technology described above could become a big part of Danish energy grid in the future. One instance where this technology could be implemented is in the capital region of Denmark, where by 2025 all central heating plants will be electrified.[26, p52] This means that a system that can take advantage of the flexibility and storage of fluid heating will become an essential part of the future energy grid.

Going forward in this project, the term "smart temperature control" will be considered to be a part of appliances. This is due to the similarities with autonomous appliances, which will be described in section 2.3.2.

2.3.2 Appliances

In this report, appliances will be defined as being on a scale from needing human planning to being autonomous. Figure 2.2 illustrates this scale. Appliances that fall within the human planning extreme are appliances that need some type of human planning to complete their primary tasks. These could be appliances that require humans to plan when and for how long the appliance should run. On the other end of the scale is the autonomous extreme, which includes appliances that do not need human intervention to complete their primary task.

Figure 2.2: The appliance autonomy scale

The reason behind the inclusion of the appliance scale is that some appliances may need human intervention today, however, a future version of said appliance might slowly move closer to the other extreme or even become completely autonomous. When an appliance becomes autonomous, it means that it becomes more flexible in relation to when it is able to run. If managed correctly, this means that the autonomous appliances can be turned on when there is a surplus of RE, and they can be switched off during peak hours; when there is an RE deficit. This could help balance out the electricity grid.

The report will provide the reader with three examples of different types of appliances, these being appliances that fall into different spectres of the appliance autonomy scale. This should help the reader get a better understanding of how the scale works and what appliances fall into the different categories.

Human planning appliances

This type of appliance needs human help both when starting up the appliance and powering down the appliance, hence the name. A good example of such an appliance is a washing machine, since it needs help being loaded before starting and should be unloaded promptly after finishing[27].

Therefore, a washing machine will be considered inflexible, because it is not able to run at all times. However, if the clothes do not need to be removed from the machine immediately, it would make the washing machine pseudo autonomous.

Pseudo autonomous appliances

This type of appliance might only need a human to set it up before it runs its primary task. Although it does require human intervention to turn on or set up the appliance before it starts, exactly *when* it runs and *when* it finishes is of little importance, as long as it finishes within a given amount of time. This given amount of time could be any time limit; hours or maybe even days. An example of this could be a dishwasher, which is an appliance that only needs loading before it is able to run purposely, and thereafter the dishwasher can be set to finish within the next few hours. An appliance like this could be loaded in the morning, just before the owner leaves for work, and set to be finished before getting home from work 8 hours later. This means that the dishwasher is required to finish at a given time, however exactly when it starts is of less importance.

Autonomous appliances

Autonomous appliances are appliances that run completely without - or with minimal - human intervention. This means that they can be scheduled to run when there is a surplus in RE, which makes these appliances very flexible. An example of an autonomous appliance is a robotic vacuum cleaner. A robotic vacuum cleaner is able to clean the house at all times of day. However, it could be scheduled to only run within certain time periods, depending on what is more convenient for the owner. This means that the robotic vacuum cleaner only needs to be charged up for specific times of the day. Therefore, the robotic vacuum cleaner could utilize spikes in RE production for charging, meaning it would charge while there is a surplus of RE.

In conclusion, it is noted that fully autonomous appliances, like the robotic vacuum cleaner, are highly flexible appliances. This means that flexible appliances are the most optimal appliances to connect to an IoT network to stabilize the energy grid and thereby also to save money for the user of the IoT platform.

2.3.3 Potential for Integration of Different Appliances into an IoT Network

This section will discuss what different types of appliances need in order to be implemented into an IoT network. This network should be able to control all the connected appliances, which ultimately would help stabilize the energy gird. The specific use cases looked at in this section will be the following appliances: Oven, washing machine, robotic lawnmower and water heater.

Oven

The oven is a prime example of a inflexible appliance. An oven should only run when a dish needs to go in the oven and while it is baking inside. After the dish has finished baking in the oven, it should be taken out promptly before it becomes cold. This means that an oven should only run for the amount of time that it takes to bake/heat up the dish, and not a minute longer. Furthermore, the oven cannot simply be set to finish within X amount of time before dinner, because the dish would become cold over time, which would be inconvenient for the user of the IoT platform and thereby cancel out the purpose of automating the oven.

However, if the oven as an appliance was to be changed in the future and thereby become more autonomous, then there are some things that should be taken into consideration. If the oven became more autonomous, the system that schedules if and when it should run, would need to know some information about the status of the appliance, as well as its schedule.

Additionally, the oven itself would also need to know some information about its status. The oven would need information on the amount of time a task would take in order for the system scheduling it to determine when it should start. Furthermore, the system that controls the oven should have information on whether the oven is already running, and if it is not running, then the system should know if it is able start.

Washing Machine

The washing machine is an example of an appliance that needs human planning, meaning it is on the manual side of the appliance autonomy scale illustrated in Figure 2.2. This is due to required human interaction before the machine is turned on, where dirty clothes should be put into the machine. Human interaction is also required when the washing machine finishes running, since the clothes need to be taken out of the machine fairly quickly. This is due to the fact that some clothes, like dress shirts, crumple up when inside the washing machine and upon drying, the wrinkles will dry into the shirt if not removed from the washing machine promptly[27]. Due to this, it is important for the IoT network to work around a somewhat tight schedule, which results in the estimated time to complete a task being important information. The network will also need to have the ability to verify whether the machine is turned on or switched off, and whether it has the ability to start running right now.

Robotic Lawnmower

The robotic lawnmower is a fully autonomous appliance. To benefit from an IoT network, it would need to have an estimated time to complete a task, i.e., mowing a section of the lawn, to determine the minimum charge of the battery needed. On top of this, the condition of the battery could be useful information to determine the maximum charge, as well as whether or not the robotic lawnmower is nearing end of life status.

The IoT network will also need to be aware of whether or not the device is turned on, and if it in a state in which it can run. This can be determined through the previously stated information, and the time frames it is allowed to work on a task.

Water Heater

A water heater is also a fully autonomous appliance. When a water heater is set up, no human interaction, other than maintenance, is required to operate it. Scheduling when a water heater should be on or off would require the water heater to send data about the temperature of its water to a server. Additionally, the server should be able to turn the heater on or off and check whether it should change state or not.

Common Characteristics of the Use Cases

Now that the needs of the above appliances have been analyzed, it has become clear that some requirements are the same among multiple appliances. These requirements are the following:

- The time it takes to complete a task
- Is it on or off?
- Can it run?
- Schedule

Since these four requirements are common among almost all types of appliances, it means that it is vital that these are met by the system in order to make it as simple as possible. However, the system should also be able to enable the connection of as many different types of appliances as possible. To enable the connection of other appliances that have more specific needs, such as the robotic lawnmower, an extension of the original system should be implemented. For example, the robotic lawnmower would, on top of the aforementioned requirements, also need to know something about its battery condition.

2.3.4 The Consumers and Their Needs

This section looks to give insight on what today's energy consumers would need from an energy software application. When the report uses the word "need", it should be understood as what a consumer would want the application to include for them to actually

use the application. This means that, in this section, energy consumers' needs will be described and analyzed.

In Section 2.1 on page 2, the report mentions Denmark's energy- and electricity-related goals of 2030 and 2050. Since this project looks to help Denmark reach these goals, the target audience for the application will be Danish electricity consumers. Therefore, it is important to look at what the target audience needs the application to contain for them to use it.

In a report made by Accenture, it is found that having easy access to a stable energy grid and being self-sufficient when it comes to energy are becoming some of the top priorities of energy consumers [28]. Additionally, energy consumers also find it important to be able to access their energy bills through their energy app [28]. The latter indicates that it is important for the users to be able to see their bills and see whether or not they are saving money on their bills by using said energy app.

For the application to be useful, it must contain the features that the users prioritize the highest. How to achieve these benefits will be discussed in this section.

The information that the consumers of the energy app need can be split into two categories: General information and Appliance information.

General information is information about general trends or information that can be summarized across different appliances:

- Current energy price
- Current amount of RE production
- Optimal hours to use energy
- Total energy used
- Total energy saved

The first three points in the list aim to inform the consumer of the current energy situation. These points would enable the consumer to change their habits, to further lower their CO2 emissions or electricity expenses by using their energy intensive appliances when there is a surplus of green energy.

The last two points in the list give the consumer insight into how much energy they are using and when they are using it. Furthermore, combining all the points would enable the consumer to see if it is possible to move some of their energy consumption to another time of the day in order to save money and the thereby also saving the environment.

The total energy used and total energy saved would be shown in kWh, and this information shows the consumers how much energy their appliances have used and how much money they have saved using the platform. In addition to how much energy the consumer has saved, this point could be further extended to show how much they have reduced their CO2 emissions.

Appliance information is information about the current state of the appliance:

- Current energy usage

- Time to complete
- Is it running?
- Can it run?
- Schedule

These points are very general and would be useful for most, if not all appliances. However, some appliances like the robotic lawnmower would require showing more specific information like the estimated time to end of life as presented in section 2.3.3. But all appliances would need to show when they are scheduled to run and whether or not they actually can run. Furthermore, they should be able to show the consumer how much energy they are currently using. This information would also be used by the server to count up the amount of energy used by the appliance.

2.4 Different Approaches

This section describes different possible approaches, in relation to which type of IoT network to proceed with. The following types of systems will be described: a decentralized system, a locally hosted device manager, and a regional device manager. Additionally, this section will look into pros and cons related to the different types of systems. The discussion is followed by a conclusion on which approach will be the best choice for this project.

2.4.1 Decentralized System

One type of approach to the problem is a decentralized system. The main idea of this approach is that there is a central server that forecasts the RE surplus for the day. The connected IoT devices then make their own decisions on when to run on the basis of the forecasted optimal times to run. After a device has figured out when it would like to run, it notifies the server of this and sends equivalent information back to the server. After receiving this new information, the server updates the forecast to be more precise. This should happen every time a new device sends information about running/startup times to the server. Note that in this type of system, the IoT devices will have to be smart, meaning they will be able to calculate the most optimal times to run.

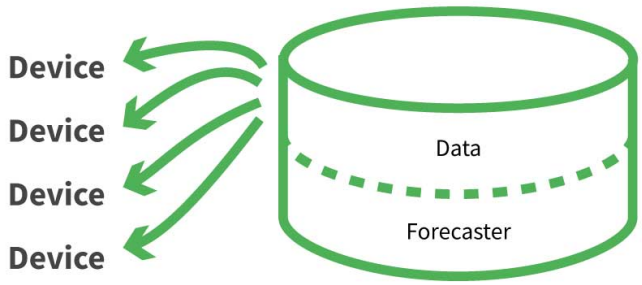


Figure 2.3: Illustration of the relationship between all appliances in a decentralized system.

Now, it will be discussed why the decentralized system is suboptimal when it comes to the problem that this report looks to solve: A decentralized system is fairly complicated and has some problem areas. One of the main issues are the race conditions that emerge when too many devices want to run at the same time. Say, if the most optimal time to run according to the server forecast would be at 15:00-16:30, then all of the IoT devices would like to start at this time, since they are all individually programmed to start at the most optimal hours. This would cause the server to experience an overload of run requests at the optimal hours, which could end up meaning that the forecasted optimal hours would no longer be the most optimal, because of the large amount of devices starting at the exact same time. If the optimal hours were only forecasted to be optimal with up to 20 devices running at the same time, and 1000 devices would like to run at the given time, it would no longer be optimal in terms of saving money for the user. Therefore, it is possible that it could be more optimal to spread out the running times of these devices, so that the peak in energy consumption could be reduced.

Another problem with creating a decentralized system is that it would require a lot of bandwidth, because of all the information that the devices would send to the server and vice versa. The problem in it itself is not that the devices and the server have to communicate regularly, but rather the amount of data being sent each time. If the server has to send a graph illustrating the RE surplus of the day, that is a huge amount of data being sent over the Internet. This explains why the decentralized system would require more bandwidth.

The last problem about the decentralized system that will be mentioned is the rising price of the IoT devices that comes naturally with this approach. If a decentralized system was to be implemented, the IoT devices would have to be smart to be able to send information to the server, as mentioned previously. If the devices were able to calculate the most optimal time to run, the price of each individual device would increase, since they would need more computational power than devices that cannot. Therefore, the smart devices would have higher production costs, resulting in more expensive devices, which would conflict with the idea of saving money for the user of the system.

2.4.2 Local Device Manager

An alternative to a completely decentralized system is a system based on each house being a central point. This type of system will be called a Local device manager. This type of system still needs to have a connection to the central forecasting- and data server, but the only devices that would communicate with the forecasting- and data servers would be the local request- and device managers. The request- and device managers are still decentralized, as each individual house needs a local request- and device managing server. When compared to the decentralized system, a per house device manager reduces the production cost for appliances, but it also requires a small server in every house. The local device manager could potentially also suffer from the same issue as the decentralized system; the issue regarding race conditions, as there is no centralized control of how many appliances run in a specific time span.

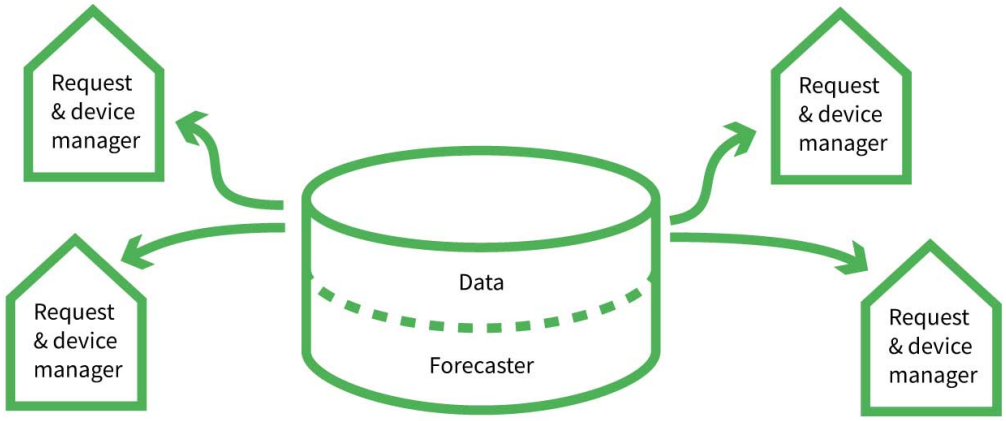


Figure 2.4: An illustration of the relations between the forecasting- and data server, and the locally hosted request- and device managers

The benefits of a implementing the system on a local device manager is in giving the consumer a large amount of freedom and control of their personal IoT devices. However, this will also result in an installation that requires some technical knowledge, as managing devices and local servers is unlikely to be within the average consumer's technical knowledge. It could, however, be be a largely beneficial system when the consumer is a person/company with enough technical skills to set up and manage the server, especially if they want a custom made device manager.

2.4.3 Regional Device Manager

Instead of having all the appliances be controlled by themselves or by a central hub in the house, they can be controlled by a regional server that controls thousands of appliances. Such a system is composed of four components as seen in figure 2.5 below:

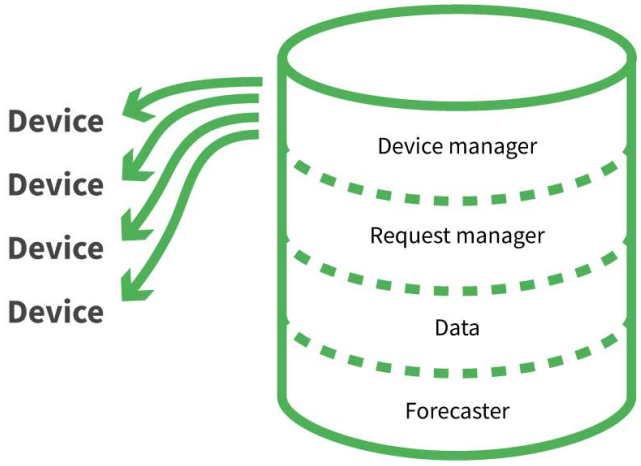


Figure 2.5: Illustration of the relationship between all appliances and a regional server.

The device manager handles all the appliances like the one in section 2.4.2. Additionally, it uses a request system to find out when an appliance should run. The request system uses the data provided by the forecasting system to make an informed decision on when the device should run. When the device manager has a schedule for an appliance, it will send a command to the appliance telling it when to run.

By handling all the scheduling and management on one device manager, the amount of internet traffic will be lowered. This is due to the fact that in a decentralized network, all appliances communicate back and forth with the forecasting server, whereas in a centralized system this communication is lowered to only include simple commands and requests for information.

Instead of communicating back and forth over the Internet when finding a fitting time to schedule an appliance, this happens inside the server itself. When a schedule is found, a command is sent over the Internet to the appliance, commanding it to start at the given time.

When the scheduling happens on the server instead of a decentralized network, race conditions become less of an issue, because all the requests are handled sequentially. This can be enabled because the server has full control over when each device manager would try to schedule an appliance.

Offloading all the computational power to a server makes the computer on the appliance much simpler, because the appliance does not have to make complicated calculations. It only has to receive commands and send small amounts of data. This makes the appliance cheaper to develop and produce.

For the consumer, such a system means that they have less control over their appliances, as scheduling of the appliance is not done by the appliance itself, but offloaded to a regional server.

Centralizing the control over so many appliances is, however, a high impact target in the case of a security breach. Having a single server controlling many appliances means that if a breach is made on the central server, all of the appliances connected could potentially be compromised.[29] This does not only apply in the case of a cyber attack, it can also be caused by a power outage, which can shut down the server and therefore shut down the connection to the service.

2.4.4 Conclusion on Choice of Approach

After reviewing the different types of approaches, one approach has been chosen to be the best one for this project, namely the regional device manager. This type of system has been chosen because of various reasons: The regional device manager has less Internet traffic than the other approaches. This is due to the fact that a regional device manager only needs to send calculated data from the server to the IoT devices and not the other way around. The IoT devices only need to send simple responses to the server, meaning the server handles the complex part of the communication. Another reason for choosing this type of system is that the individual devices will be simpler to maintain and develop for the manufactures. This is because the main calculations are handled by the centralized server.

2.5 Problem Statement

The problem analysis of this report was based on the following intializing problem: *How can IoT help maintain a stable energy grid with an increase in fluctuating energy sources and increased energy consumption?*

After analyzing the problem field in great depth and thereafter researching different possible approaches, the following problem statement has been formulated:

How can a centralized IoT network use flexible appliances to move energy consumption to when RE is produced and thereby help stabilize the energy grid?

With the adoption of RE sources, the energy grid will become less flexible in when it can produce energy because of the increased dependence on environmental phenomena. To keep the energy grid stable with this loss in flexibility, appliances could part of a possible solution. The reason for this is that these appliances have been researched and found flexible as to when they run and use energy. Therefore if these appliances could be scheduled efficiently, they could be used to stabilize the energy grid.

2.5.1 Requirements

For an application to be successful, it would need to fulfil the following requirements. The requirements have been split into three sections: Central server, devices on the network and a user interface.

The centralized server should be able to:

- Gather data from devices on the network
  - Current state
- Turn on/off devices on the network

A device on the network should be able to:

- Receive commands from the server
- Send information to the server
- Communicate to the server what type of device it is when initializing

The user interface should be able to:

- Visualize data in a simple way
  - Forecasted renewable energy surplus
  - Each device and its parameters

2.6 Chapter Summary

This section looks to give a summary on the the Problem Analysis Chapter.

The goal of this project is to develop an application that helps stabilize the energy grid in order to reach Denmark's energy goals, as mentioned on page 2.1. However, the problem of making an energy utilization platform is rather broad, and therefore a problem delimitation is needed.

In the Problem Delimitation, different ways of creating an energy utilization platform are analyzed: Using batteries to stabilize the grid, planning home automation, which includes temperature control and controlling appliances, and stabilizing the grid by adjusting the industry sector's energy usage. Here, home automation was chosen as the focus point of the project.

After the delimitation, different approaches on how to implement the IoT platform have been investigated. These include a decentralized system, a local device manager and a regional device manager. The regional device manager is chosen as the best choice for this project's IoT network, because it has less Internet traffic and the devices will be simpler to maintain than on another type of system.

After choosing which type of IoT system will be developed, the following problem statement is formulated:

How can a centralized IoT network use flexible appliances to move energy consumption to when RE is produced and thereby help stabilize the energy grid?

Chapter 3: Modelling and Design

When developing software, it is key to create models that show the structure and flow of the application before developing the implementation. This section will explain the models made for the developed software. The models described have been chosen because of their relevancy to the application.

Designing the system was done following the top down approach. This means that the core of the program based on the requirements was defined first, which was then divided into smaller, more detailed parts. This choice of approach made it possible to design the program in a sophisticated way, as well as in a detailed way.

The goal of this chapter is to show the main idea behind the structure of the program. Furthermore, it is important to understand how the different modules communicate with each other.

3.1 Overall Application Structure

Figure 3.1 shows the model of the overall structure of the server-side application. The server-side has been divided into modules. This approach has been chosen to make the server simple to build and maintain. When working in a module based environment, the developer can isolate errors to a given module, which allows for multiple developers to work on different parts of the system at the same time, without worrying about version control or overwrites. Another great advantage of module based environments is the improved overview of the application. This overview makes it easier to navigate the system and expand it. The developer can simply build a new module and deploy it to the system when finished.

Note that the model of the overall structure shows the server modules. Therefore, it is abstracted from detail. Each module will be described in further depth in their subsequent sections.

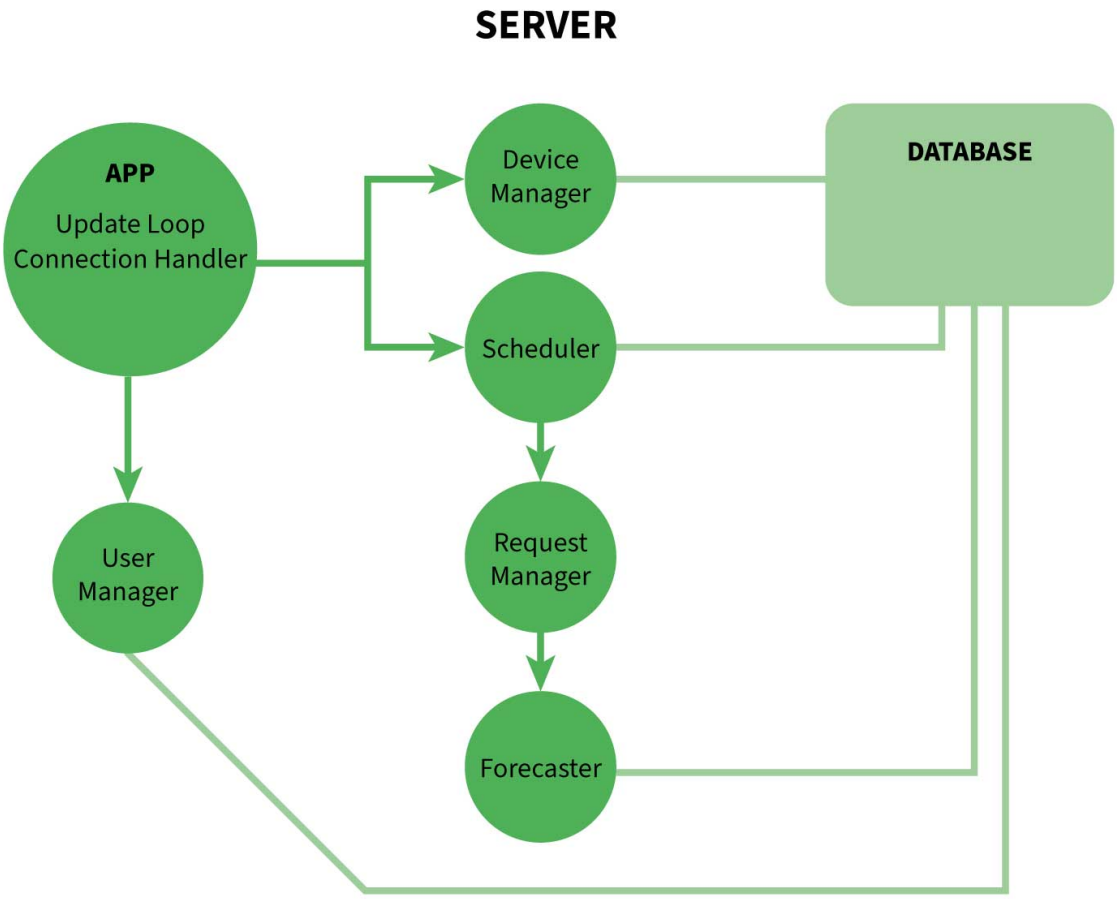


Figure 3.1: Model of overall structure of the server-side application.

The core of the server is the Application File (APP), which is in charge of updating the system and calling the server-side modules, when their functionality is needed.

3.1.1 Update Loop

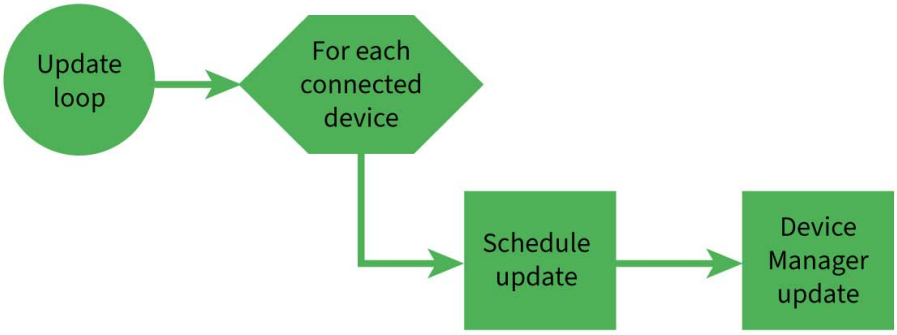


Figure 3.2: Model of the update loop in APP.

In each update iteration, APP goes through each connected device and sends it first to the Scheduler and the then to the Device Manager, as shown in Figure 3.2.

The Scheduler will schedule the device if it is not already. When it schedules a device it calls the Request Manager.

When the Request Manager is called, it requests a graph from the Forecaster. It uses this graph to schedule the device at the optimal time. These calls follow the module structure illustrated in Figure 3.2.

When the Scheduler has completed its task, APP calls the Device Manager update. The Device Manager then updates the device's state.

Once the application has finished the update loop, it calls the User Manager. The User Manager then decides what new data to send to the client-side application.

3.2 Device Manager

The Device Manager is in charge of updating and sending commands to the devices. Furthermore, the Device Manager is the only part of the server that communicates with the devices.

Connection Started

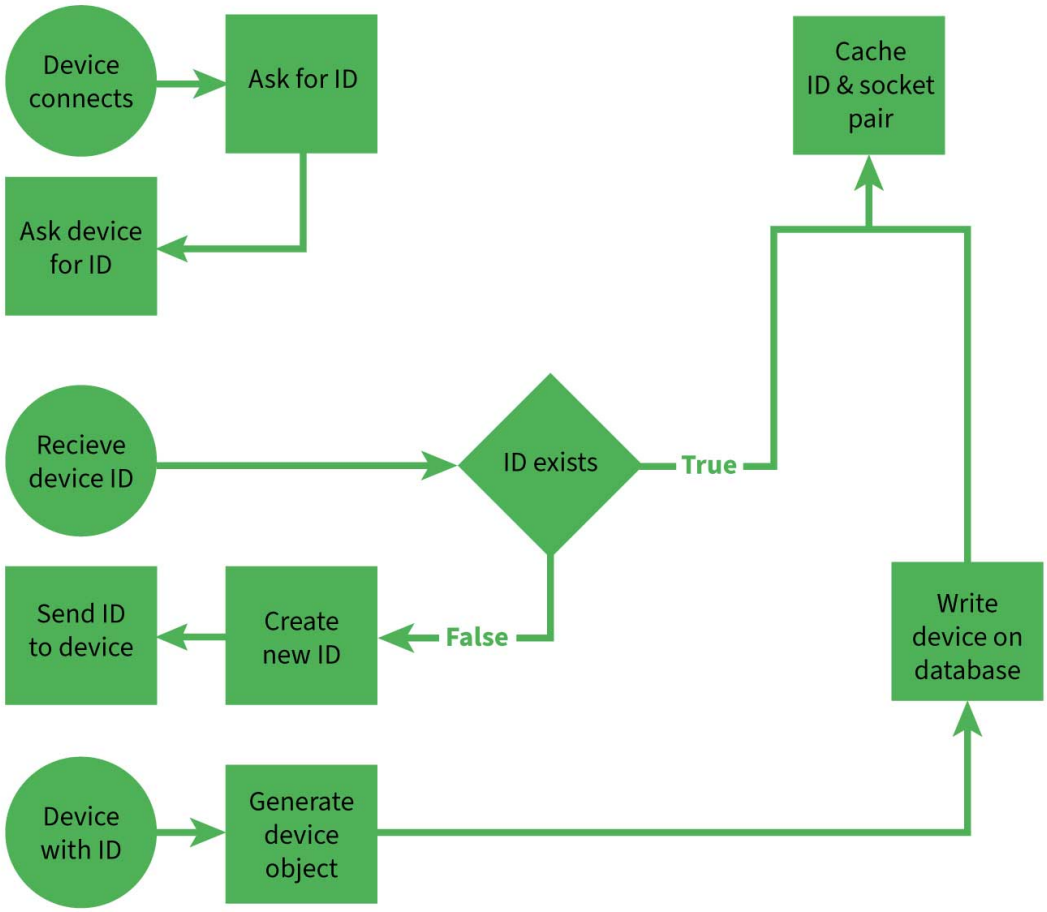


Figure 3.3: Model of the "Connection Started" functionality in the Device Manager.

When a device first connects, the server will request the device's ID. If the ID does not match any of device IDs on the database, then a new ID will be created and sent to the device. When the server receives confirmation that the ID has been assigned, the device will be created on the server and saved in the database.

When the device has been assigned an ID or connects with a valid ID, the server will cache the ID with its corresponding socket in a active connections array, for future communication.

When a device loses connection to the server, the ID socket pair is removed from the active connections array, as it is no longer an active connection.

Delete Device

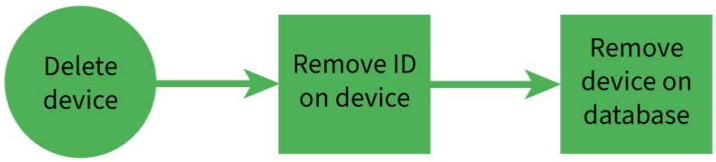


Figure 3.4: Model of the "Delete Device" functionality in the Device Manager.

The deleteDevice function calls deleteDevice on the Database Accessor with the ID of the device it should delete. The Database Accessor will then make sure the device is deleted from the database.

Device Manager Update

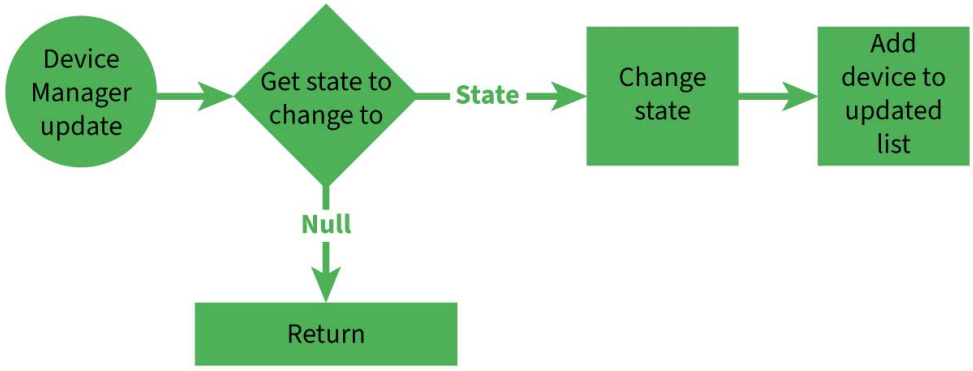


Figure 3.5: Model of the Update loop in the Device Manager.

The Device Manager has a main update loop, which is used to manage all devices as seen in Figure 3.5. This method is called from the main update loop in APP, where it receives a device to manage. It first checks to see if the device should change state. If this is the case, it will command the device to change state and add the device to the list of updated devices. If it should not change state, then it will simply return and do nothing. The list of updated devices is combined with the list of updated devices from the Scheduler, after which the combined list of updated devices will be sent to the User Manager, where they will be sent to the client.

Get State to Change To

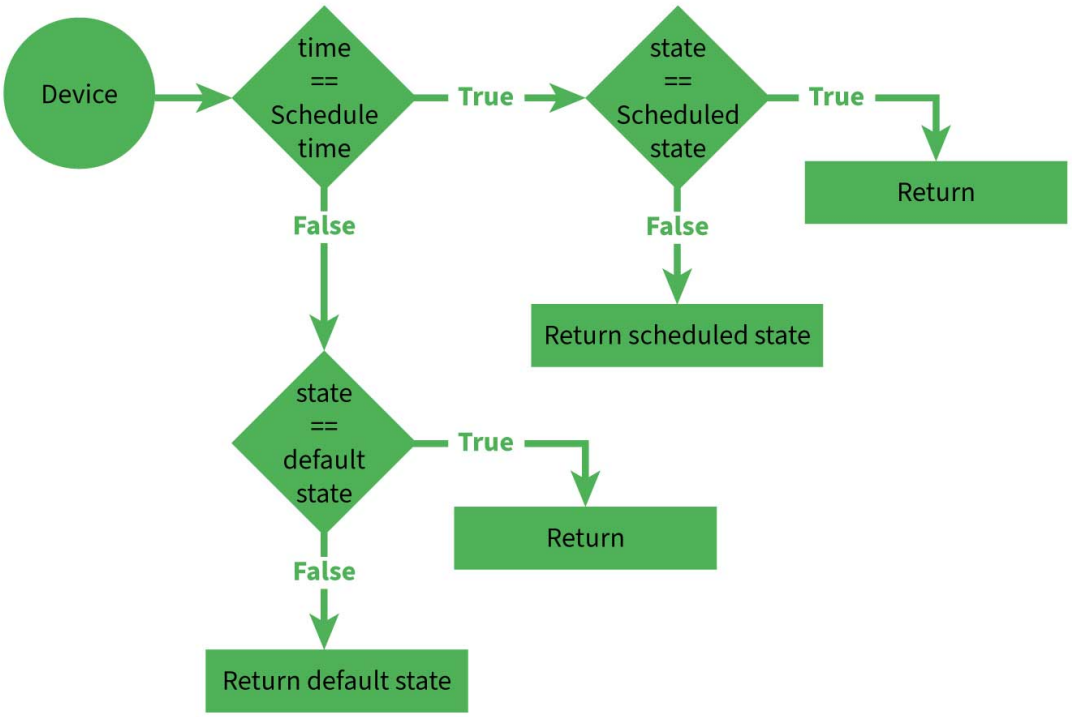


Figure 3.6: Model of the "Get State to Change To" functionality in the Device Manager.

The get state to change to function checks the device's current state and makes sure it is correct. The function compares the current time and the device's scheduled time, to see what state the device should be in. When this knowledge is established, the function checks the device's current state. If the device is in a wrong state, the function returns the state it is supposed to be in. If the device is in the correct state, it simply returns without a state.

Change State

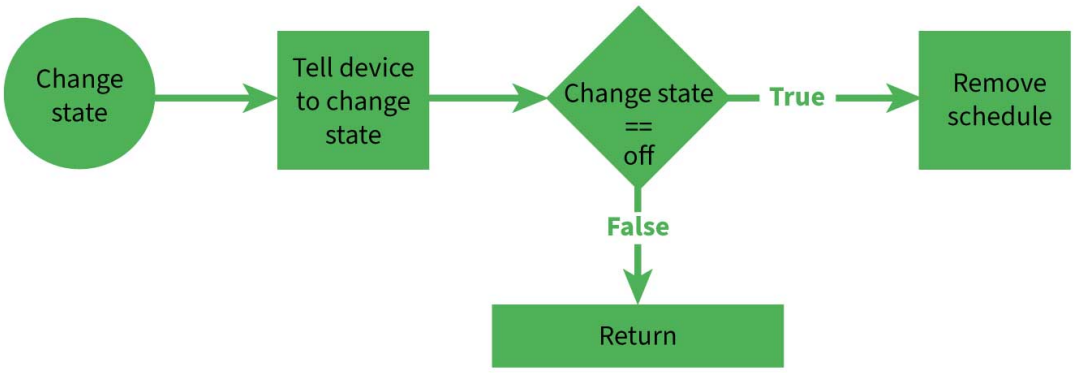


Figure 3.7: Model of the "Change State" functionality in the Device Manager.

The change state function tells a device that the server wants it to change state. If the device was turned on by a schedule, the schedule will be considered done and removed.

Update Device

Figure 3.8: Model of the "Update Device" functionality in the Device Manager.

When an update from a device is received by the server it will update the system accordingly. The update device function takes a device as an input parameter and uses its Device ID to locate and update the device in the database. Afterwards the device will be added to the updated devices array.

Communication

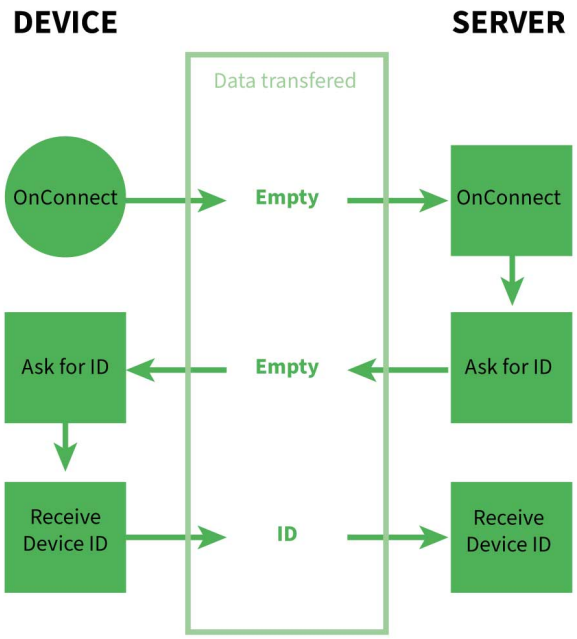


Figure 3.9: Model illustrating when a device connects to the server.

When a device connects to the server, the device emits "OnConnect" without sending any data to the server. The server then emits "Ask for ID" back to the device, where the device will respond with a "Receive Device ID" emit, sending its ID to the server.

If a device connects for the first time, the ID sent in "Receive Device ID" will be invalid, and the server emits "Set ID", sending a new ID to the device. Afterwards, the device emits "New Device with ID", sending its new ID and init info back to the server.

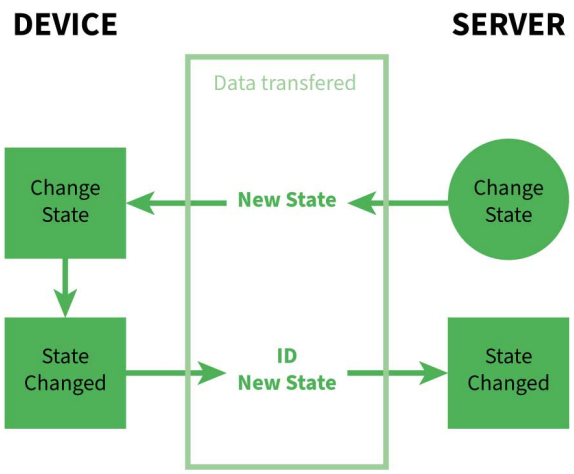


Figure 3.10: Model illustrating when a device state has been changed.

When the server wants to change the state of a device, the server sends a "Change State" emit with the new state to the device.

When a device has changed state, regardless of whether the device was told to do so by the server or not, it sends a "State Changed" emit with its ID and its new state to the server.

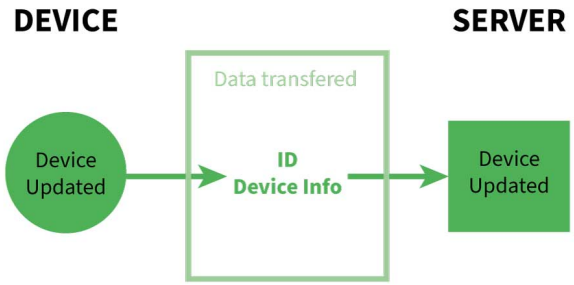


Figure 3.11: Model illustrating when a device is updated.

When a device has been updated, it sends its ID as well as the device info to the server in a "Device Updated" emit.

3.3 Scheduler

As mentioned in Section 3.1, the Scheduler handles everything that is related to the scheduling of devices. This includes, but it not limited to; scheduling a device, rescheduling a device and removing device schedules.

Schedule Device

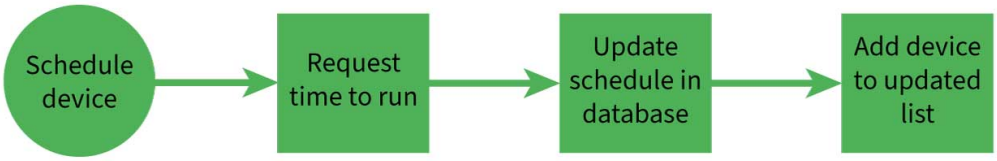


Figure 3.12: Model of the "Schedule Device" functionality in the Scheduler.

When a device needs scheduling, the Scheduler receives the device it should schedule from the update loop in APP.

After receiving the Device object, the Scheduler retrieves a time interval and a demand graph from the device, which it then uses to call the function requestTimeToRun from the Request Manager. This function will be explained in greater depth in section 3.3.1, which is the following section.

After calling the requestTimeToRun function, which returns the time interval where the device should run, the Scheduler updates the device schedule in the database with the new information. Lastly, the Scheduler adds the newly scheduled device to a list of updated devices called updatedDevices.

Schedule Update

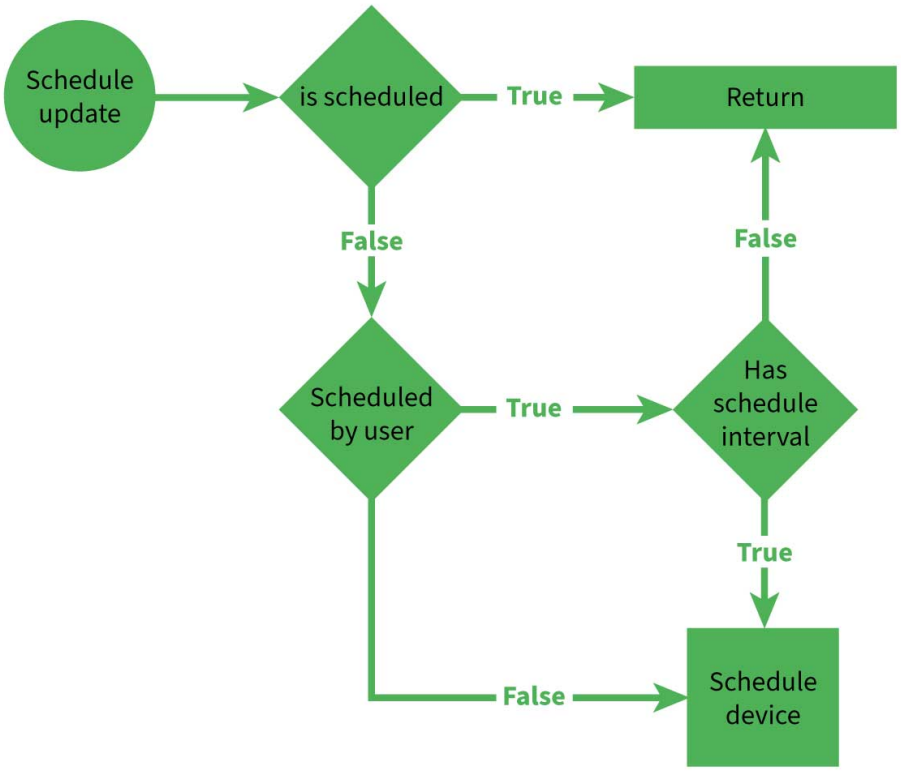


Figure 3.13: Model of the "Schedule Update" functionality in the Scheduler.

If a device schedule needs to be updated, the Scheduler calls the scheduleUpdate function. It takes the Device object as input and checks whether or not the device already is scheduled.

If it is scheduled, then the function returns, since a device that is already scheduled should not be scheduled again by this function (See the "Reschedule Interval" functionality below instead). If the device is not scheduled, then the function checks if the device should be scheduled by a user or not.

If it should not be scheduled by a user, the device will be scheduled. However, if it should be scheduled by a user, then the device will only be scheduled if it already has been assigned an interval by the user.

Reschedule Interval

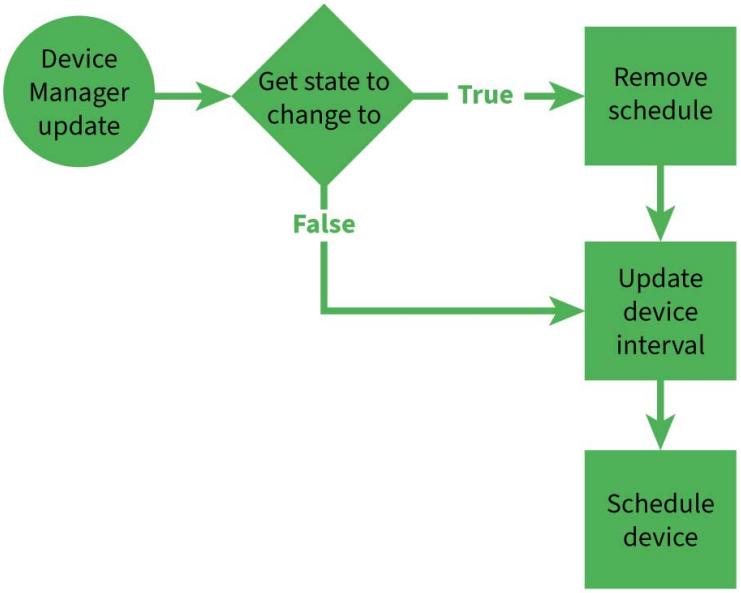


Figure 3.14: Model of the "Reschedule Interval" functionality in the Scheduler.

If a device needs to be scheduled by a user the rescheduleInterval function will be called. This function takes a Device object, the new interval and the program which should be run as input.

The first thing this function does is to check whether the device is already scheduled or not. If it is scheduled, then the function removeSchedule from the Scheduler is called (See the following subsection). After the schedule is removed, the new time interval and program will be updated in the database.

However, if the device is not already scheduled, the function removeSchedule does not need to be called, and the database can be updated with a new time interval and program immediately.

After updating the database, regardless of whether the device was already scheduled or not, the scheduleDevice function is called.

Remove Schedule

Figure 3.15: Model of the "Remove Schedule" functionality in the Scheduler.

If a device schedule needs to be removed completely, removeSchedule receives the device whose schedule should be removed.

Before removing the schedule from the device, the scheduled demand should be removed from database. This is done by calling removeCurrentDemand in Request Manager.

After the demand has been removed successfully, removeSchedule removes the schedule from the device and updates the database accordingly.

3.3.1 Request Manager

The Request Manager is the part of the Scheduler that handles requests. This module has two primary functions; finding the optimal time to start a device in a given time interval and removing demand when a device is rescheduled.

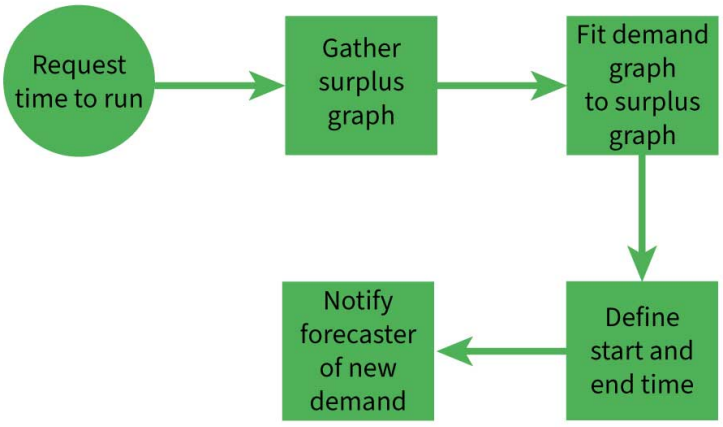


Figure 3.16: Model of the "Request a time to Run" functionality in the Request Manager.

The requestTimeToRun function takes a demand graph and an interval as input. The function uses this input to find the optimal time to start the device. Firstly, the function

retrieves the current surplus graph from the Forecaster, which gathers the surplus graph from the database. Next, the demand graph is fitted to the newly retrieved surplus graph.

After fitting the graphs, the start and end times of the interval are defined, which will be the start time and end time for the device. When the start and end times have been defined, the Request Manager notifies the Forecaster of the new demand and start time.

Finally, the function returns a time interval; namely the start and end times that were just defined.

The Request Manager's removeCurrentDemand function is quite simple and is therefore not illustrated here: The function takes a demand graph and a start time as input. It then calls the Forecaster's removeDemand function, which removes the device demand graph from the Forecaster.

3.3.2 Forecaster

The purpose of the Forecaster is to handle incoming demand and production graphs, both from the service itself, and from the third party API calls. This will be done with three primary functions, namely the updateSurplus function, addDemand function, and removeDemand function.

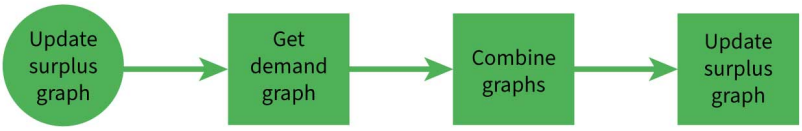


Figure 3.17: Model of the "Update Surplus" functionality in the Forecaster.

The updateSurplus function will be pulling the third party API information, namely the third party RE production and third party demand, as well as pulling data on the IoT network energy demand from the database. This data will be used to create an overall RE surplus graph by combining the other graphs, which will be written to the database for other parts of the service to use.

The addDemand and removeDemand functions are pretty simple, which is why they have not been illustrated in the report. The addDemand function takes demand and start time data from a device, and adds this data to the related demand graph in the database.

The removeDemand function takes demand and start time data from a device, and removes this data from the related demand graph in the database.

3.4 User Manager

The User Manager is the link between the server and the client-side application. It keeps track of the connected users, and provides updates on graphs and user-owned devices. The

User Manager also provides a way for the client-side application to set the schedules of, removing the schedule of, and immediately starting, non-autonomous devices.

User Connected

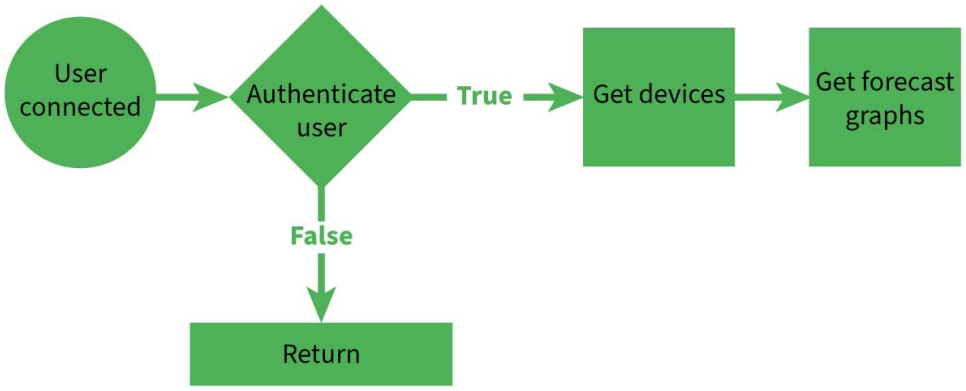


Figure 3.18: Model of the "User Connected" functionality in the User Manager.

When a user connects to the server, an initial setup is run. This initial setup will first authenticate the connecting user. Note that if the user is not authenticated the connection is dropped by returning. If a user connects and is authenticated, all devices that are connected, and the graphs from the forecaster will be sent to the client.

Scheduling

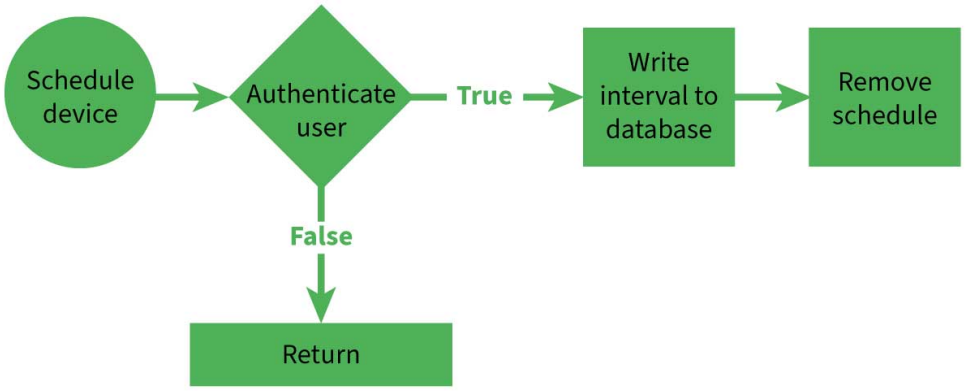


Figure 3.19: Model of the "Scheduling" functionality in the User Manager.

The scheduling function will run if a user decides to manually set a schedule for a device. When a user sets a schedule for a device, the user will first be authenticated and then the interval will be written to the database. If a schedule already exists on the database it will be removed and replaced with the user created interval.

User Manager Update

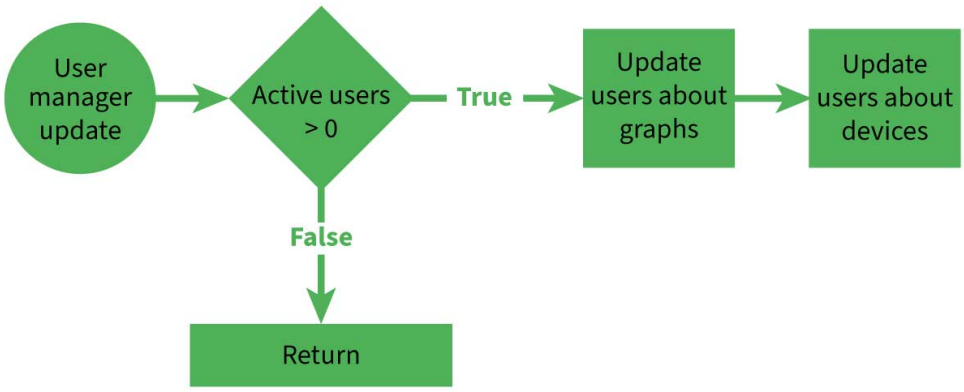


Figure 3.20: Model of the "User Manager Update" functionality in the User Manager.

The user manager update will send new data to the user when a device or a graph has been updated. There are two main functions that are called from the user manager update, the "update users about graphs" function and the "update users about devices" function. These functions are called in each update iteration of the APP if there is a user connected. The two functions will be described in figure 3.21 and figure 3.22.

Update users about graphs

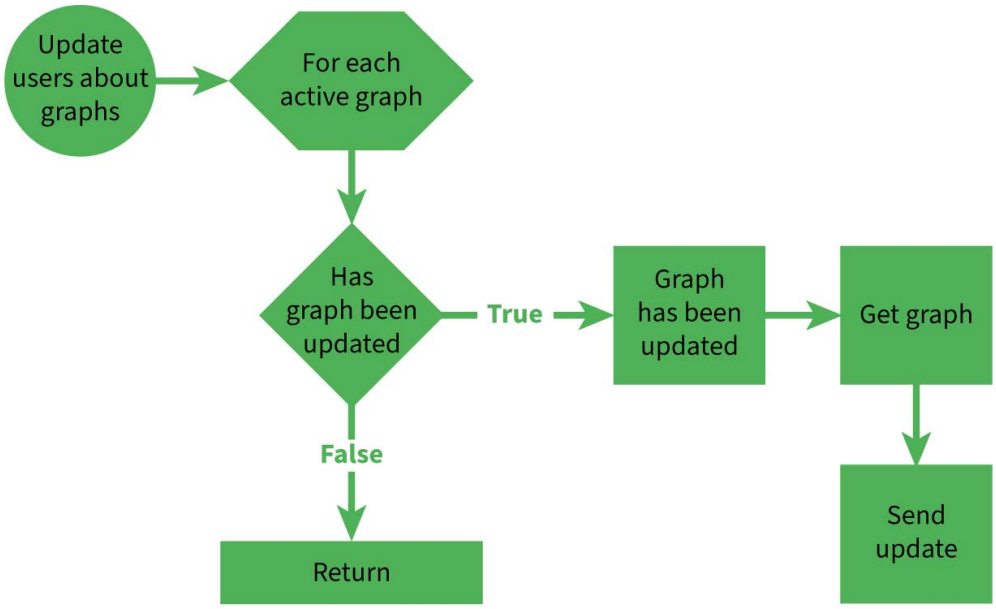


Figure 3.21: Model of the "update users about graphs" functionality in the User Manager.

Update users about graphs will return updates for every active graph to the frontend. The function checks if the active graph has changes since last updated, and if this is the case it sends an update. If the graph has not been changed the function does nothing.

Update Users about Devices

Figure 3.22: Model of the "Update Users about Devices" functionality in the User Manager.

"Update Users about Devices" will iterate through a list of all updated devices, and send each updated device to the frontend. The updated devices list is made in the device manager.

Communication

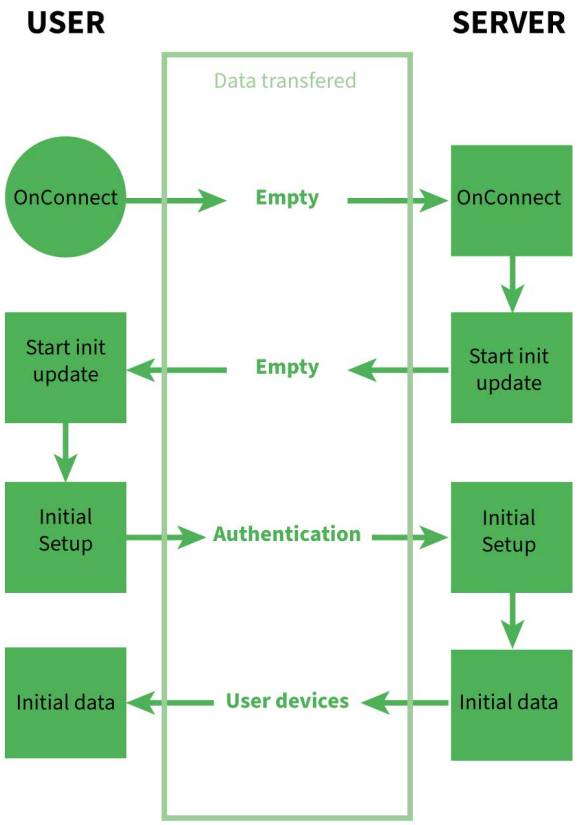


Figure 3.23: Model illustrating when a user connects to the server.

When a user connects, the user emits "OnConnect" without any data. When the server receives the "OnConnect" it will emit "Start init update" to the user, this is also an empty emit. Upon receiving the "Start init update", the user will emit "Initial Setup" to the server. When the user is authenticated, the server will emit "Initial Data" which includes all the user's devices that are connected, as well as all the user's graphs belonging to those devices.

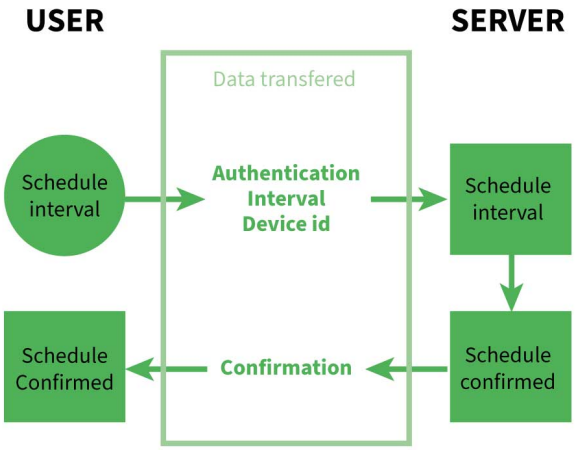


Figure 3.24: Model illustrating when a user schedules a device.

When a user schedules an interval, "Schedule Interval" will be emitted. This emit contains the wanted interval, user authentication and the device's ID. When the server receives the "Schedule Interval" emit, it will confirm the schedule by emitting "Schedule Confirmed" which includes a Boolean that indicates if the scheduling succeeded.

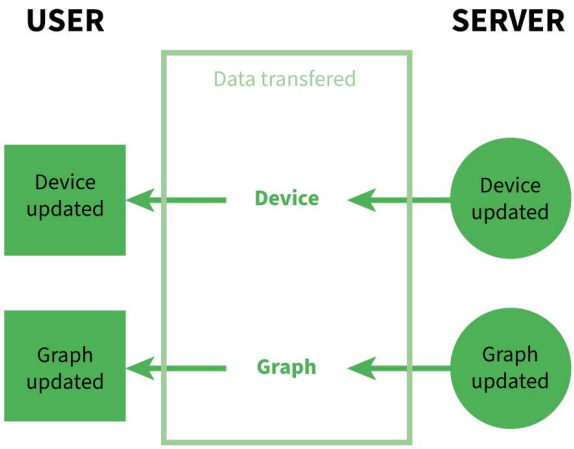


Figure 3.25: Model illustrating when a device or graph gets updated.

When a device is updated, the server will emit "Device Updated" and include the device that has been updated. The updated device can then be displayed on the frontend. The same applies for when a graph is updated, although the emit will instead be "Graph Updated", and the data sent with the emit will be an updated graph.

3.5 State Machines

As a part of modelling, two state machines have been made; one for the water heater device and one for the washing machine device. In short, a state machine is a behaviour model that has a finite number of states [30], all depicted as a square box. The arrows between the states depict state transitions. Based on the current state the machine is in and a given input, the machine performs a certain transition into another state. [30] This means that a state machine can change state depending on its current state and a given condition.

The state machines have been made to visualize how the two devices should behave. The state machines show how the devices switch between different states. After modelling the state machines, it will be easier to implement, verify and test the devices in Chapter 4.

The following subsections describe the state machines for the water heater and the washing machine, respectively.

3.5.1 Water Heater

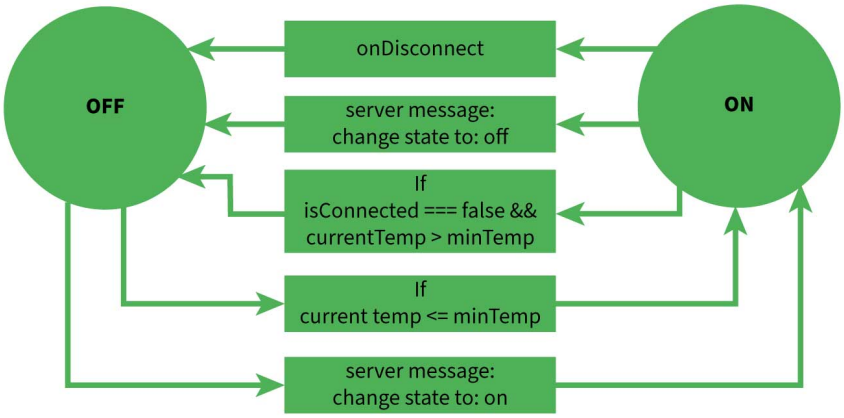


Figure 3.26: Model illustrating the state machine for the Water Heater.

The state machine for the water heater shown in Figure 3.26 consists of two states; "Off" and "On". Here, the state "Off" implies that the water heater is shut off and therefore is not heating, and the state "On" implies that the water heater is currently heating up water.

If the water heater's current state is "Off" and its current temperature is less than or equal to the minimum temperature of the water heater device, e.g. 55 °C, it should be turned on, regardless of whether the device is connected to the server or not. This is done to ensure that the water heater does not fall below its minimum temperature. Additionally, if the water heater's current state is "Off" and the device receives a command to change state to "On", the device switches state to "On".

If the current state of the water heater is "On", it is possible to turn the machine off in

three different ways: Firstly, the water heater's current state will be switched from "On" to "Off" if the device suddenly disconnects from the server. This is done to make sure that the system always will be able to control that the water heater's current temperature never exceeds the maximum temperature of the device, e.g. being 90 °C.

Secondly, the water heater can also change state from "On" to "Off" if the device receives a command to turn off.

Lastly, if the device is "On" and the current temperature exceeds the maximum temperature, then it will change to "Off".

3.5.2 Washing Machine

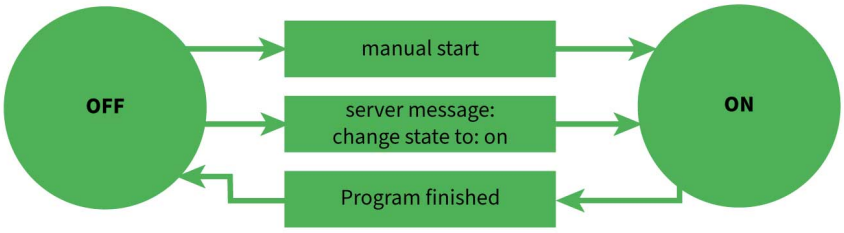


Figure 3.27: Model illustrating the the state machine for the Washing Machine.

The state machine for the washing machine shown in Figure 3.27 is simple: It consists of two states, "Off" and "On". Note that in this case, "On" means that the washing machine is running a program, but the term "On" is used to simplify the model.

If the washing machine's current state is "Off" and the user presses the start button on the machine (which is called "manual start" on the figure), then the machine should turn on.

Additionally, if the washing machine's current state is "Off" and the device receives a command to change its state to "On", the machine turns on.

Lastly, if the washing machine is "On" and the current program finishes, it should change state to "Off", since it no longer needs to run.

Chapter 4: Implementation

This chapter presents a description of the development phase of the project. It will go into detail on how different parts of the system have been made and in what order they were implemented. The goal of this chapter is to give the reader insight on the process of implementation, as well as a clearer view on how each part of the system works individually.

Even though the top down approach was used to design the model, this approach was not chosen for the implementation, because in the implementation phase, it can be beneficial to implement everything from the bottom up. The bottom up approach means starting from the bottom and then building upon that base until the entire system has been built. However, this approach is only viable if everything has been modelled in great detail, so that every module is described well. Another great thing that comes with a system that is defined well through modelling, is that it is fairly easy to test.

To enable multiples teams to work on the system at the same time, an execution plan is needed. This plan should sketch out when different parts of the system can be worked on, and which parts depend on each other. Therefore, an execution plan has been made for the implementation of this project. The plan can be seen in Figure 4.1, and this plan will be used as the general outline of the Implementation chapter.

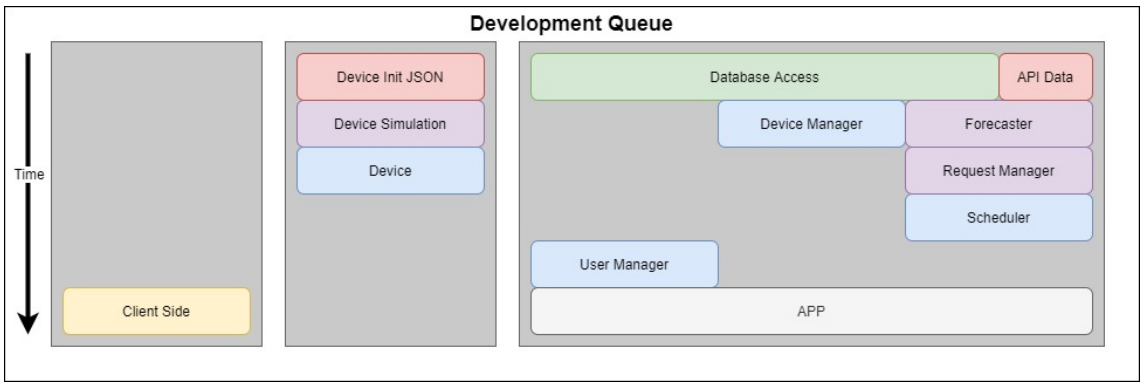


Figure 4.1: Illustrates the Development Queue. Every box can be worked on when all of the boxes above it have been implemented. For example, the *Device Manager* can be worked on when *Database Acces* has been implemented, even if *Device Init JSON* is not yet completed.

This chapter is split into three parts corresponding to the main components of the system:

- Devices
- Server
- User Interface

Additionally, a short Testing section will be included later in this chapter.

If the reader wishes to see some of the functions that have not been included as code examples, or if they wish to read the code in a more continuous way, they are encouraged to see the project's GitHub page where all the code can be found under the Implementation folder:

`https://github.com/theodor349/Central-Energy-Stabilizer`

4.1 Devices

This section describes the development of the device side of the system.

Initially, the project looked to include two devices in the final product: The washing machine and the water heater. However, the minimum viable product only needed to include the water heater, which is why the water heater has been implemented first.

4.1.1 Device Setup Files

When looking at the code, it is seen that the water heater device consists of two different JSON files: DeviceId and DeviceInfo.

The DeviceId file is a file with one line of code that contains the device ID. It was decided early on in the project that the ID should be in a separate file, since this property is going to be overwritten quite often.

However, the device ID is not just any ordinary string; the device ID is generated by creating a Version 4 UUID[31]. A Universally Unique Identifier (UUID) is a unique identifier that is generated by using random numbers.

The DeviceInfo file contains the rest of the information on the device which is the name of the device, max power (measured in W), device type and a Boolean value called isAutomatic. Additionally, the file also contains some unique properties for the device, these being minimum temperature, maximum temperature and its current temperature.

4.1.2 Device Implementation

The water heater has been implemented following the state machine diagram in Section 3.5.1. The implementation has been fairly straightforward because of the well-described diagram.

The setState function from APP, seen in Listing 7.1 in the Appendix, is the function in which the state machine is incorporated. The function consists of an if else chain that checks if the water heater should change state depending on what values its properties currently are. The water heater object is passed into the function so that the function can access its properties.

As mentioned above, the setState function is a direct implementation of the state machine into the code. Therefore, further explanation of the code will not be added. The reader is encouraged to look up the code in the Appendix if they need to confirm that it is a direct implementation of the water heater state machine.

The washing machine device would be more complex to implement, since the washing machine is not a fully automatic device like the water heater. However, this device has not been implemented during the original time frame of this project. This will be discussed in further depth in Chapter 6, Section 6.2.

4.2 Server

This section describes the development of the server side part of the system. It encompasses:

- Database Access
- API Data
- Forecaster
- Request Manager
- Device Manager
- Scheduler
- User Manager
- APP

Every part of the server side system will be described in the following subsections in the above order.

4.2.1 Database Access

The database is accessed for two different types of data: Device data and graph data. Therefore, the Database Accessor is split up into two parts; one that handles device data and one that handles graphs. This means that there will be two different scripts that handle communication with the database, so that all the functionality regarding graphs and devices are separated. This is done to ensure readability of the database access code, as well as making sure each script does not have multiple responsibilities.

The following subsections describe the Device Accessor and Graph Accessor, respectively. These subsections should give the reader insight on how the entire database is structured and how to access the different types of data found in the database.

Device Accessor

The Device Accessor's job is to create, get and update devices in the database. Additionally, the Device Accessor also has to be able to delete a device, if necessary.

On the server, a device consists of the device's own parameters, as well as some server-side parameters, shown in Listing 4.1.

```
let Device = mongoose.model("Devices", new mongoose.Schema({
    // From Server
    scheduledByUser: Boolean,
    isScheduled: Boolean,
    nextState: String,
    schedule: String, // Object with Start and End Time (actual Running time)
    scheduledInterval: String, // Object with Start and End Time (Scheduled Time)

    // From Device
    deviceId: String,
    name: String,
    maxPower: Number,
    isAutomatic: Boolean,
    currentPower: Number,
    currentState: String,
    deviceType: String,
    isConnected: Boolean,
    programs: String,
    uniqueProperties: String
}));
```

Listing 4.1: The Mongoose Model of a Device on the server.

The Device Accessor has four exported functions, as seen in Listing 4.2 below:

```
const functions = {
    createDevice: (device) => createDevice(device),
    getDevice: (id) => getDevice(id),
    deleteDevice: (id) => deleteDevice(id),
    updateDevice: (id, field, value) => updateDevice(id, field, value),
}
```

Listing 4.2: The exported functions from *DbAccessorDevice*

These four functions are everything other scripts need to call if they want to create, get, update or delete a device.

The createDevice function takes in a device with all its parameters and inserts that into the database.

The getDevice and deleteDevice functions both only need the ID of the device in order to find the device and either return or delete it.

The updateDevice function takes in the ID of the device it should update, as well as the field and corresponding value it should update. It then uses a switch and the field value to create the correct query before sending it off to the database for updating.

Graph Accessor

The Graph Accessor's job is similar to the one of the Device Accessor, although it handles graphs instead of devices. The Graph Accessor creates, gets and updates graphs in the database. The system will need to store four types of data in these graphs: Scheduled demand, forecasted RE production, forecasted demand and the corresponding surplus.

The following functions are the exported functions from DatabaseAccessorGraphs:

```
const functions = {
    createGraph: (graph) => createGraph(graph),
    getGraph: (id) => getGraph(id),
    updateGraph: (id, values, shouldAdd) => updateGraph(id, values, shouldAdd),
}
```

Listing 4.3: The exported functions from *DatabaseAccessorGraphs*.

Graph IDs and the dateToId function

Before explaining the primary functions of the Database Accessor, this subsection gives an in-depth explanation on how graph IDs are generated: A graph consists of an ID and a list of 60 numbers, where each of the 60 numbers resembles a value corresponding to one minute. This means that there is one number for every minute in an hour. If one graph consists of 60 numbers, then the system can represent 24 hours of forecasted demand by creating 24 graphs with 60 numbers each.

```
function dateToId(prefix, date) {
    return prefix +
        "-Y" + date.getFullYear() +
        "-M" + date.getMonth() +
        "-D" + date.getDate() +
        "-H" + date.getHours();
}
```

Listing 4.4: The "dateToId" function from *Utilities*.

The dateToId function shown in Listing 4.4 is a utility function from Utilities.js. It is used to generate graph IDs. It takes a prefix and a date as input, the prefix being the graph type and the date being the current date and hour.

The ID of a graph is a string generated with both a prefix and a postfix. The prefix is set to be the type of graph that is currently being created, and the postfix of the ID

consists of the date and time that was passed into the function through the Date object. Additionally, the postfix includes letters Y-, M-, D- and H- as prefixes in front of the ID's year, month, day and hour, respectively. The date and time of the ID is retrieved by using the getFullYear(), getMonth(), getDate() and getHours() JavaScript methods on the Date object. The function returns the full ID.

An example of an ID for a scheduled demand between 16:00 and 17:00 on January 24, 2020 would be "scheduledDemand-Y2020-M0-D24-H16".

Keep in mind that the "Month" part of the graph ID is zero indexed, which is why "M0" corresponds to the first month in the year, this being January. February would be "M1", and so on and so forth. The rest of the ID postfix is not zero indexed.

Additionally, the "Hour" part of the graph ID is set to include the starting hour of the schedule. In this example, the scheduled demand is set between 16:00 and 17:00, which is why the starting hour, "H16", is written as part of the ID.

The createGraph function

The createGraph function is called with a Graph object containing an ID and an array of 60 numbers. This function will insert the Graph object into the database, if the graph is valid. A graph is classified as valid if the number of values in the array is equal to 60, as seen in Listing 4.5

```
1 function isGraphValid(graph) {
2     return graph.values.length === 60
3 }
```

Listing 4.5: The function "isGraphValid" from *DbAccessorGraph*.

The getGraph function

The getGraph function is called with the ID of the graph which it should return from the database. When getGraph asks the database for a graph, the database will simply return the graph if it is found and getGraph will return that result. However, if the ID does not exist on the database and the graph is not found, the database will return null instead.

If the graph does not exist, getGraph will create a new graph consisting of 60 zero values and return it after it has been created. This is done to make sure that it is always possible to get a graph using the getGraph function. Additionally, this is done because of the assumption that if a graph does not exist, a new one is needed, and this is most likely going to be a graph with all values being zero.

The updateGraph function

The updateGraph function takes the ID of the graph to update, the points which will be used to update the graph and a Boolean named shouldSum, that determines whether it should swap all values in the graph or sum them.

Firstly, the function checks if the values are valid. If this is not the case, it will return false. When the values have been validated, the function retrieves the graph from the database using the graph ID. If it should sum the values together, it will go through each value from the database and add the new values, otherwise it will just overwrite the old values completely.

When the points have been updated, the function sends the updated graph to the database and it returns true if the update was successful.

4.2.2 API Data

Due to a lack of API accessibility during the implementation of the project, the decision to simulate necessary data about the electricity network was made, namely simulating the graphs for apiDemand and apiProduction for RE. These two datapoints are used to calculate the RE surplus at any given time, which can be used by other elements of the program, such as the frontend graph drawing or the Forecaster.

The following functions are the exported functions of the Forecaster API:

```
1 const functions = {
2     updateApiGraphs: (date) => updateApiGraphs(date),
3     updateApiDemand: (date) => updateApiDemand(date),
4     updateApiProduction: (date) => updateApiProduction(date),
5 }
```

Listing 4.6: The exported functions of *ForecasterAPI*.

The first of the exported functions on the list is the updateApiGraphs function, in which the rest of the graph generation functions are called. This is done through a for loop that generates the relevant graphs for each hour, over the next 48 hours. This for loop is shown in the code example below:

```
1     ...
2     for (var i = 0; i < 48; i++) {
3         updateApiProduction(date);
4         updateApiDemand(date);
5         updateApiSurplus(date);
6         date.setTime(date.getTime() + 60 * 60 * 1000);
7     }
8     ...
```

Listing 4.7: The "updateApiGraphs" function of *ForecasterAPI*.

The updateApiDemand and updateApiProduction functions are very similiar to each other, as their functionality essentially is the same. Therefore, only one of these functions will be included as a code example, this being the the updateApiDemand function, which is shown in Listing 4.8 below. The only difference between the two functions is that in line 7

inside the for loop of updateApiDemand, the getDemandAt function is called, but inside the for loop of the updateApiProduction, the getProductionAt function is called instead. Additionally, in line 10, the updateApiProduction uses the string "apiProduction" as input instead of "apiDemand" that is used in function updateApiDemand.

These two functions are the bridge between the mathematical equations in getDemandAt and getProductionAt, and the database graph storage. getDemandAt and getProductionAt can be seen in Listing 7.5 in the Appendix.

```
1  async function updateApiDemand(date) {
2      return new Promise(async (resolve, reject) => {
3          let values = new Array(60);
4          let d = date.getDate();
5          let h = date.getHours();
6          for (var m = 0; m < 60; m++) {
7              values[m] = getDemandAt(d * 24 + h + m / 60);
8          }
9
10         let id = util.dateToId("apiDemand", date);
11         let res = await da.updateGraph(id, values, false);
12         resolve(res);
13     })
14 }
```

Listing 4.8: The "updateApiDemand" function of *ForecasterAPI*.

4.2.3 Forecaster

As mentioned in Chapter 3, the Forecaster is split into three primary functions, with a series of helper functions that reduce the repetition of code. The primary functions are defined as:

```
1  const functions = {
2      updateSurplus: (interval) => updateSurplus(interval),
3      addDemand: (startTime, graph) => addDemand(startTime, graph),
4      removeDemand: (startTime, graph) => removeDemand(startTime, graph),
5  }
```

Listing 4.9: The primary functions of *Forecaster*.

The updateSurplus function

Firstly, we have the updateSurplus function, which gets an interval as an input parameter. This interval is used to define the interval in which the updateSurplus needs to update the surplus graphs. The function returns all the values of the surplus graph within the interval for later usage.

```
1  async function updateSurplus(interval) {
2      return new Promise(async (resolve, reject) => {
3          ...
4          let graph = [];
5
6          let hoursInInterval = (interval.finish.getTime() -
7              interval.start.getTime()) / (60 * 60 * 1000);
8
9          for (let i = 0; i < hoursInInterval + 1; i++) {
10             await updateSurplusGraph(surplusStartTime);
11
12             let updatedSurplus = await
                   da.getGraph(utility.dateToId("surplusGraph", surplusStartTime));
13             for (let t = 0; t < 60; t++) {
14                 graph.push(updatedSurplus.values[t]);
15             }
16             surplusStartTime.setTime(surplusStartTime.getTime() + 60 * 60 * 1000);
17         }
18         resolve(graph);
19     });
20 }
```

Listing 4.10: The "updateSurplus" function in the *Forecaster*.

The core of the updateSurplus functionality is within the for loop on line 9, which runs once for every hour in the interval. The function calls a helper function updateSurplusGraph, in which the surplus graph of the current time frame of the loop is calculated and written to the database.

Once the current surplus graph is in the database, updateSurplus retrieves the updated graph and pushes the values unto the graph array. At the end of each loop iteration, the surplusStartTime is incremented by an hour to prepare for the next loop iteration. After the loop iterations have finished, the function returns the graph of the entire interval.

The addDemand and removeDemand functions

The addDemand function is called when a new device is being scheduled, and the demand graph has to be updated.

The primary functionality of the addDemand function consists of a for loop, which handles demand graphs that are larger than 60 values, and a section of code which handles the remaining graphs of less than 60 values. The addDemand function can be seen in Listing 4.11 on the following page.

```
    ...
    for (i = 0; graph.length > 60; i++) {
        graphToSplit = await createGraphToSplit(graph);

        demandGraphs = splitGraph(startTime, graphToSplit);
        lowerGraph.graphId = await utility.dateToId("demandGraph", startTime);
        upperGraph.graphId = await utility.dateToId("demandGraph",
            secondGraphStartTime);
        lowerGraph.values = demandGraphs.demandGraphLower;
        upperGraph.values = demandGraphs.demandGraphUpper;

        await updateDemandGraph(lowerGraph);
        await updateDemandGraph(upperGraph);
        await updateSurplusGraph(startTime);

        startTime.setTime(startTime.getTime() + 60 * 60 * 1000);
        secondGraphStartTime.setTime(startTime.getTime() + 60 * 60 * 1000);
    }
    ...
```

Listing 4.11: The main functionality from the "addDemand" function in the *Forecaster*.

The for loop calls the createGraphToSplit function, which takes the first 60 values from the demand graph and sends them to the splitGraph helper function. The splitGraph helper function takes a graph with a length between 0 and 60, and fits it into two graphs, according to the specific time a demand starts. These two graphs are then updated in the database, as well as the surplus graph, with the initial start time of the earliest of the graphs. The for loop then prepares the initial start times of the next loop iteration by adding another hour to the Date objects.

```
    ...
    demandGraphs = splitGraph(startTime, graph);
    // Puts the graphId and values into the lower and upper bound graphs
    lowerGraph.graphId = await utility.dateToId("demandGraph", startTime);
    upperGraph.graphId = await utility.dateToId("demandGraph",
        secondGraphStartTime);
    lowerGraph.values = demandGraphs.demandGraphLower;
    upperGraph.values = demandGraphs.demandGraphUpper;

    await updateDemandGraph(lowerGraph);
    await updateDemandGraph(upperGraph);
    await updateSurplusGraph(startTime);
    await updateSurplusGraph(secondGraphStartTime);
    ...
```

Listing 4.12: The main functionality from the "addDemand" function in the *Forecaster*.

The end of the addDemand function echoes much of what is inside the for loop, with the exceptions that the createGraphToSplit is unnecessary, as the graph length is at most 60, and the surplus graph for the final hour is updated.

Finally, the removeDemand function is the simplest of the bunch, as it merely calls the invertValues helper function to invert the values of the incoming graph, after which it passes the inverted graph to the addDemand function, as seen in Listing 4.13.

```
async function removeDemand(startTime, graph) {
    return new Promise(async (resolve, reject) => {
        graph = invertValues(graph);
        await addDemand(startTime, graph);

        resolve(true);
    });
}
```

Listing 4.13: The "removeDemand" function in the *Forecaster*.

4.2.4 Request Manager

The Request Manager's primary function is to handle request for when a device should run. Additionally, it has the function removeCurrentDemand that is also exported. All the exported functions can be seen below:

```
const functions = {
    requestTimeToRun: (graph, timeIntervalObject) => requestTimeToRun(graph,
        timeIntervalObject),
    removeCurrentDemand: (currentSchedule, graph) =>
        removeCurrentDemand(currentSchedule, graph),
}
```

Listing 4.14: Exported functions from *Request Manager*.

The requestTimeToRun function

```
function requestTimeToRun(graph, timeIntervalObject) {
    return new Promise(async (resolve, reject) => {
        let outputIntervalObject = {
            start: undefined,
            end: undefined
        };

        fitDemandToSurplus(outputIntervalObject, timeIntervalObject, graph);

        await forecaster.addDemand(outputIntervalObject.start, graph);

        resolve(outputIntervalObject);
    });
```

Listing 4.15: The "requestTimeToRun" function from *RequestManager*.

The function shown in 4.15 is called when the scheduler requests a time a device should run, it takes a graph and a time interval object as input. The graph is a array of points resembeling how much power the device uses. The time interval object is the time interval in which the device can run. Say, if a device is scheduled by a user to run between 16:00 and 19:00, the time interval's start property will be 16:00, and the end property will be 19:00.

Firstly, the requestTimeToRun function creates the output object the function will return. This consists of a start and an end time, which starts out as being undefined. Next, the function fitDemandToSurplus is called. This function will take the output object and the other parameters and find the optimal time for the device to run. When the time to run is found, the forecasted demand is added to the database and the output object is returned.

4.2.5 Device Manager

The first thing that the Device Manager does when a device connects is to ask its ID. This is done by sending a command. A command is defined as an object that consists of a socket, a command and a payload. This structure can be seen in the createCommand function shown in Listing 4.16:

```
function createCommand(socket, command, payload) {
    let commandObj = {
        socket: socket,
        command: command,
        payload: payload,
    }
    commandQueue.push(commandObj);
}
```

Listing 4.16: The "createCommand" function from *DeviceManager*.

When a command has been made, the command will be added to the commandQueue which will be executed by APP at the end of each update iteration. The socket is the link used to communicate with a device, the command is the message to emit to the socket and the payload is any data that has to be sent with the message. For example, the payload could include an ID or a device state.

When a device connects or disconnects

Shown below is the onConnect function. This is the first function that is called from the Device Manager when a new device connects.

```
function onConnect(socket) {
    createCommand(socket, "askForId");
}
```

Listing 4.17: The "onConnect" function from *DeviceManager*.

As seen in Listing 4.17, the createCommand function from APP is called, with a socket from the connecting device, and the command *askForId*. Note that if no payload is defined, then the function will create a command with the payload being undefined.

```
function onDisconnect(socket) {
    return removeConnectionWithSocket(socket);
}
```

Listing 4.18: The "onDisconnect" function from *DeviceManager*.

As seen in Listing 4.18, the onDisconnect function calls the local helper function removeConnectionWithSocket with the socket of the device disconnecting. This function returns either a true or a false value, depending on whether or not the index of the device disconnecting is found in the activeConnections array.

Firstly, the index of the device in the activeConnections is found by searching for the disconnecting device's socket, and is then saved in the "index" binding. If the index is found, the device is removed from the activeConnections array by using the splice() method on the array, and the function returns a true value back to onDisconnect. If the index is not found, the function returns a false value.

Updating devices

In Listing 4.19, the updateDevice function is shown. This is one of the more important functions in the Device Manager.

```
async function updateDevice(deviceInfo, serverCheck) {
    let dbDevice = await db.getDevice(deviceInfo.deviceId);
    if (dbDevice === null) {
        return 0;
    }
    let fieldsToUpdate = getFieldsToUpdate(deviceInfo, dbDevice, serverCheck);
    for (let i = 0; i < fieldsToUpdate.length; i++) {
        await db.updateDevice(deviceInfo.deviceId,
            fieldsToUpdate[i].field,
            fieldsToUpdate[i].value);
    }
    if (fieldsToUpdate.length > 0) {
        updatedDevices.push(deviceInfo.deviceId);
    }
    return fieldsToUpdate.length;
}
```

Listing 4.19: The "updateDevice" function from *DeviceManager*.

The updateDevice function is a key function of the Device Manager. It is an asynchronous function that takes the deviceInfo and a Boolean value serverCheck as input. Simply

explained, if serverCheck is true, then it means that the device object is from another part of the server which means there are extra variables to check for changes. If the value is false, it probably means that it came from a device or somewhere outside the server.

When the updateDevice function is called, it gets the device from the database by using the getDevice function, saving this device locally in the dbDevice binding. If this binding is null, the device was not found, and the function returns and does nothing.

After ensuring that the device was found on the database, the getFieldsToUpdate function is called on the device and saved in a local binding, fieldsToUpdate. This function finds out which device fields should be updated by checking every field for changes. Every time it finds a field that should be updated, the field name and value are saved in a binding called fieldsToUpdate. After checking through all the fields, the function returns the fieldsToUpdate to the updateDevice function.

After the fieldsToUpdate have been found, a for loop runs through the entries in the array and updates the fields in the database with their respective values by calling the updateDevice database function.

Lastly, the function then checks if there were any actual changes. If so, it adds the device ID to the list of updatedDevies. Next, the function returns the number of fields that were updated.

Managing devices

Another important Device Manager function, the manageDevice function, is shown in listing 4.20:

```
1 function manageDevice(deviceInfo) {
2     let time = new Date();
3     let nextState = getScheduledState(deviceInfo, time);
4     if (nextState === null) {
5         return false;
6     }
7     changeState(deviceInfo.deviceId, nextState);
8     return true;
9 }
```

Listing 4.20: The "manageDevice" function from *DeviceManager*.

In this function, the Device Manager decides which command to send to the device, if any.

To decide this it needs to know what state it should be in, therefore it calls getScheduledState. This function returns null if the device has not been scheduled or is already in the correct state. Otherwise, the function returns the state the device should be in.

When manageDevice has the right information, it will change the state if the nextState is not null. The function returns true if it has changed the state of the device and false if it did nothing.

When a device state has been changed

The stateChanged function is shown in Listing 7.7 in the Appendix.

This function will update the database with the new state of the device.

It does so by retrieving the device from the database and changing the state. If the new state is off, the function also resets some values to their default values. These are:

- isScheduled is set to false
- nextState is set to null
- schedule is set to null
- scheduledInterval is set to null

After having changed the device state, the function updates the database and returns.

4.2.6 Scheduler

The Scheduler is, as mentioned in Chapter 3, in charge of scheduling devices as its primary function. Its secondary functionality is adding devices to a list of updated devices, when a device is changed.

All of the exported functions from the Scheduler are shown below:

```
1 const functions = {
2     scheduleDevice: (device) => scheduleDevice(device),
3     getCommandQueue: () => getCommandQueue(),
4     clearUpdatedDevices: () => clearUpdatedDevices(),
5     getUpdatedDevices: () => getUpdatedDevices(),
6 }
```

Listing 4.21: Exported scheduling functions from *Scheduler*.

The scheduleDevice function

The Scheduler's most important function is scheduleDevice, which takes a device as input. The function can be seen in Listing 7.6 in the Appendix.

The scheduleDevice function that has been implemented is designed to only handle fully autonomous devices, which can turn on/off at a moment's notice. This means that the scheduling process becomes very simple, as a device can only be in one of two states: On or off. As the device can switch between the states on and off instantly, the system only needs to check if there is RE surplus to decide whether or not to change state.

The first thing the scheduleDevice function does is that it retrieves the surplus graph from the databaseAccessor.

Next, it checks if there is surplus of RE, and if there is, the device should be turned on. However, if there is no RE surplus, then the device should be turned off.

If the device's current state is the same as the scheduledState, the device will not be turned on, and nothing happens when scheduleDevice is called. But if the device should change state, a new command with the new state is sent to the device and the device ID is saved in a list of updated devices for later use.

4.2.7 User Manager

The User Manager is, as mentioned in Chapter 3, the link between the server and the client-side application.

Below are the exported functions from the User Manager:

```
const functions = {
    onConnect: (socket, connectedDevices) => onConnect(socket, connectedDevices),
    onConnectDevice: (socket, connectedDevices) => onConnectDevice(socket,
        connectedDevices),
    onConnectGraph: (socket) => onConnectGraph(socket),
    sendUpdatedDevices: (updatedDevices) => sendUpdatedDevices(updatedDevices),
    graphUpdate: () => graphUpdate(),
    sendAPISurplusGraph: (socket, date) => sendAPISurplusGraph(socket, date),
    sendSurplusGraph: (socket, date) => sendSurplusGraph(socket, date),
}
```

Listing 4.22: Exported functions from *UserManager*.

Initially, the User Manager was meant to follow the diagrams closely, but due to the scope being reduced, much of the original functionality has been removed, which results in a simplified version. First, we have the onConnect function, which calls the onConnectDevice and onConnectGraph functions. The onConnect function is shown in Listing 4.23.

```
async function onConnect(socket, connectedDevices) {
    await onConnectDevice(socket, connectedDevices);
    await onConnectGraph(socket);
}
```

Listing 4.23: The "onConnect" function from *UserManager*.

The onConnectDevice and onConnectGraph functions are in charge of requesting the device and graph data from the database, after which the onConnectGraph adds the command "updateDevice" to the command queue together with the corresponding device data, and the onConnectGraph calls the two functions sendAPISurplusGraph and sendSurplusGraph. These two functions are in charge of issuing the commands "createGraphValues" with the relevant Graph object as a payload. These functions are shown in Listings 7.2 and 7.3 in the Appendix.

The commands are issued through a call to the createCommand helper function, in which an object with a destination in this case "userSpace"(all users), command and payload are defined. This command is pushed into a command queue, which the APP handles.

Updating the User Interface

Besides the previously described functions, there are two functions for updating the data on the User Interface, which are also defined in the User Manager.

The first of these is the sendUpdatedDevices function. This function receives a list of all the updated device IDs and requests these devices from the databaseAccessor before creating a command with the updated device info.

The sendUpdatedDevices function is shown in Listing 4.24 below:

```
async function sendUpdatedDevices(updatedDevices) {
    return new Promise(async (resolve, reject) => {
        if (updatedDevices.length === 0) {
            resolve(true);
        } else {
            for (let i = 0; i < updatedDevices.length; i++) {
                let device = await dbD.getDevice(updatedDevices[i]);
                createCommand("userSpace", "updateDevice", device);
            }
            resolve(true);
        }
    });
}
```

Listing 4.24: The "sendUpdatedDevices" function from *UserManager*.

The second function is the graphUpdate function, in which the latest interval of the graph is requested from the database, and similarly passed onto the createCommand helper function. The graphUpdate function is shown in listing 7.4 in the Appendix.

After running these two functions, an up-to-date frontend user interface is provided to the users.

4.2.8 APP

The APP is the primary file that is run; it either directly or indirectly calls every other file on the server. This is done through an update loop, which runs at a given iteration interval, in which all active devices are retrieved from the Device Manager, scheduled through the Scheduler, managed by the Device Manager and the User Manager is called to update the User Interface.

Listing 4.25 shows the main update loop of APP:

```
async function startServer() {
    print("Starting server");
    setInterval(() => {
        update();
    }, updateInterval);
}
```

Listing 4.25: The "startServer" function from *APP*.

This function will run the update function every updateInterval which is set to 1000. Meaning every second the update function will be run. The update function is described in listing 4.26

```
async function update() {
    //print("Update started")
    let devices = dm.getActiveConnections();

    for (let i = 0; i < devices.length; i++) {
        let device = await dd.getDevice(devices[i].deviceId);
        if (device !== null) {
            ...
            await sd.scheduleDevice(device);
        }
        device = await dd.getDevice(devices[i].deviceId);
        if (device !== null) {
            dm.manageDevice(device);
        }
    }

    handleCommands();
    await updateUserManager();
    //print("Update finished")
}
```

Listing 4.26: The "update" function from *APP*.

The update function starts by getting a list of all the connected devices. Next, the for loop will iterate over all the device IDs from the activeConnections array, and get the matching device from the database. After checking if the device exists, the device will be scheduled by using the scheduleDevice function from the Scheduler. After getting scheduled, the device will be managed, meaning it will make a command that will change it to the state that it has been scheduled to, when the command is executed. The handleCommands function will execute all commands from the Device Manager, Scheduler and User Manager, and the updateUserManager function will then send all the updated devices, as well as the updated graphs, to the client.

Within the Device Manager, Scheduler and User Manager, lists of commands are created, which are retrieved by APP, and executed through the executeCommand function which is essentially called from handleCommands.

```
function executeCommand(command) {
    let socket = command.socket;
    let payload = command.payload;
    command = command.command;
    if (socket === 'userSpace') {
        userSpace.emit(command, payload);
    } else {
        socket.emit(command, payload);
    }
}
```

Listing 4.27: The "executeCommand" function from *APP*.

The executeCommand function will take a command as an input and emit the command to the corresponding socket, with the attached data. If the socket is userSpace, the command will be emitted to the user space, otherwise it will go to the device space. The userSpace has to do with displaying information to the user. Everything relating to the devices, e.g. updating a device's state, is handled on the deviceSpace.

4.3 User Interface

This section describes the interface a user will use to interact with the system. Additionally, this section will include access limitations, structure, design choices for the interface and how information from the server is processed before being displayed for the user.

In short, the User Interface (UI) is the part of the program that connects the user and the system through an interface. For this application, a Graphical User Interface (GUI) has been developed. This has been chosen because of its flexibility. In a GUI, the developer has unlimited access to develop user actions and they can be changed, updated and expanded in real time. This makes a GUI attractive in dynamic environments that must be able to adjust when changes happen in the system.

Another great benefit with GUIs is the improved information flow. In a GUI environment, the developer is able to use typographical laws and design approaches to help the user navigate the interface.

Access

When designing UIs, access limitations is very important. A user should only be able to access information and the parts of the system relevant for that user. This improves the quality of the service by limiting the things that can go wrong.

In the implementation, there are two main systems that ensure the limited access for users. Native in Node, server files are not shared with any user trying to get access. Files that

should be accessible must be declared within the server script itself. This makes sure that the developers actively choose the wanted files for sharing. To make it clear for any developer what is publicly accessible in the system, a folder named Public was created on the server and all public content is contained inside it.

When a user wants to access information on the server outside the Public folder, another system makes sure of the access limitation. This is the Socket.io library.

Socket.io allows the user to send a request to the server. When this request is received by the server, it will then look through a list of allowed requests to figure out how to handle the request. If a user requests something not defined or out of its access limitation, the server will simply ignore the request.

```
1 const userSpace = io.of('/user');
2 userSpace.on('connection', (socket) => {
3
4     socket.on('getMainGraphs', () => {
5         um.onConnect(socket, dm.getActiveConnections());
6     });
7 });
8 }
```

Listing 4.28: The implementation of user requests allowed by the server.

Listing 4.28 shows how user requests has been implemented. In the first line, the scope of the user requests are defined to be userSpace. Line 2-8 then encapsulate all allowed requests. In this case the only allowed request is 'getMainGraphs'.

Structure

The GUI is seperated into three different types of structure files: HTML, CSS and JavaScript files.

The HTML files are the containers for the displayed information and are styled and manipulated by the other file types. Inside HTML files, static structure is also defined. These definitions simplify the other file types, that can exclude these definitions and simply reference the static structure in the HTML.

CSS files handle the interface's appearance. These files handle how the HTML is displayed for the user. There are plenty of benefits from separating appearance from the static structure: Firstly, it will simplify changes to the HTML because appearance can be ignored. It also makes it possible to define rules for the appearance in a global scope, instead of defining rules for each HTML element.

JavaScript files are used to manipulate the HTML files and send requests to the server. These files allow the interface to be dynamic and display content in real time.

The combination of these three file types makes it easy to maintain the program, fix errors and implement new functionality to the interface, because each file type has its own purpose and contains unique error types.

```
<body onresize="reloadPageElements()">
    ...
    <section id="mainGraphContainer"> </section>
    ...
</body>
```

Listing 4.29: HTML structure for the mainGraphContainer. Defines the order and relation between content within the interface.

```
...
.graphPathRed {
    stroke: rgba(240, 40, 40, 0.71);
    stroke-width: 2;
    fill: rgba(240, 40, 40, 0);
    transition-duration: 2s;
}
...
```

Listing 4.30: Part of CSS style for graphs. Defines how a graph called graphPathRed should look.

```
function buildProperty(propertyItem, device, list) {
    let listItem = document.createElement("li");
    let nameContainer = document.createElement("p");
    let valueContainer = document.createElement("p");
    let unitContainer = document.createElement("p");

    let property = getPropertyInformation(device, propertyItem);

    nameContainer.innerHTML = property.name + ": ";
    valueContainer.innerHTML = property.value;
    unitContainer.innerHTML = " " + property.unit;

    ...

    listItem.appendChild(unitContainer);
    list.appendChild(listItem);
}
```

Listing 4.31: Javascript function for generating device properties.

Design choices

When the GUI was designed, some steps were taken to improve the experience for the user and make it easier to navigate it. First of all, the page begins with a header to display initial information for the user that indicates the user is within the interface. It includes the app's name and some symbolic illustrations.

Underneath, the header of the main graph the systems performance is shown. The graph is placed immediately under the header for two reasons: Firstly, it gives the user information relevant for the users next actions in the interface. Secondly, it improves the performance of the page. This is because the content below the main graph can expand the page. If the graph was underneath, it would then force the interface to move the graph and all its data when the page was expanded.

In the main graph, the color theme has been chosen in regards to colorblindness to ensure the best possible user experience regardless of colorblindness.

Directly underneath the main graph, the connected devices are shown. They are displayed as a header and can be expanded to show detailed information.

Displaying Devices Connected to the Server

Displaying devices in HTML is handled by JavaScript functions that manipulate the HTML documents. The functions will locate the wanted HTML elements and add or manipulate them. The functions are generated dynamically and with unique parameters. This behavior is obtained by using predefined templates that specify how to handle different parameters.

```
function addDevice(serverDevice) {
    let devicesContainer = document.getElementById("serverDevices");
    let device = buildDeviceInHTML(serverDevice);
    devicesContainer.appendChild(device);
}

function buildDeviceInHTML(device) {
    let deviceContainer = document.createElement("section");
    deviceContainer.id = device.deviceId;
    deviceContainer.setAttribute("onClick", "displayDeviceSetting('" +
        device.deviceId + "')");

    let header = buildDeviceHeader(device);
    deviceContainer.appendChild(header);
    let settings = buildDeviceSettings(device);
    deviceContainer.appendChild(settings);

    return deviceContainer;
}
```

Listing 4.32: Functions that adds and displays a device.

The addDevice function seen in Listing 4.32 takes a device sent from the server and creates a frontend HTML element to display it and finally inserts it on the page by the appendChild function.

buildDeviceInHtml is the sub function to addDevice that creates the HTML structure for the device and calls two sub functions for the HTML generation. It also creates a reference to that specific device for easy maintenance of the device's properties.

The subfunctions that take care of generating the HTML use predefined templates to mark up and style correctly. There are two types of templates; one for a device type and one to mark up properties. This is done because different device types will share properties and by splitting the templates up, property definitions can be reused for multiple device types.

```
let waterHeater = {
    imageSrc: "./Images/waterheaterIcon.svg",
    headerProperties: ["currentTemp", "currentPower", "currentStatus"],
    settingsProperties: ["tempInterval", "maxPower", "deviceId"]
};
```

Listing 4.33: Device type definition.

```
function getPropertyInformation(device, propertyItem) {
    let property = {};

    switch (propertyItem) {
        case "currentTemp":
            property.name = "Temp";
            if (device.uniqueProperties.currentTemp !== undefined) {
                property.value = device.uniqueProperties.currentTemp.toFixed(2);
            }
            property.value2 = null;
            property.unit = "C";
            break;
    ...

    default:
            console.warn("Warning: build for property " + propertyItem + " Not
                defined");
            break;
    }
}
```

Listing 4.34: Device property definition.

When the system tells the UI about a device update, the function seen in Listing 4.35 below takes care of updating the correct HTML. This is faster than rebuilding the device from scratch with the addDevice function.

```
function updateDevice(serverDevice) {
    switch (serverDevice.deviceType) {
        case "Water Heater":
            waterHeater.headerProperties.forEach((propertyItem, propertyIdex) => {
                updateHeaderProperty(propertyItem, propertyIdex, serverDevice);
            });
            break;

        default:
            console.warn("Warning: Can't find header properties for " +
                serverDevice.deviceType);
            break;
    }
}
```

Listing 4.35: Function that updates a device.

Displaying graphs received from the server

Graphs also have templates. These templates' main goal is to make it easy to implement new graphs to the interface by reusing generation functions that take a template as input parameter. The interface's main graph template is seen in Listing 4.36.

```
let mainGraph = {
    htmlElement: null,
    svgWidth: "95vw",
    svgHeight: "450px",
    svgId: "mainGraph",
    height: "450", //px
    width: null, //pw
    horizotalTextheight: 22, //px
    verticalTextWidth: 55, //px
    innerHeight: null,
    innerWidth: null,
    horizontalLines: 20,
    horizontalValue: "Mw",
    horizontalOrigin: 12500, // where on the vertical axis should the zero point
        be
    horizontalAmount: 20000,
    verticalLines: 48,
    verticalValue: "time interval",
    verticalAmount: 24,
    lowerLimit: 1000, // When should the numbers on the horizontal axis show
        shorter text
    verticalTextOffset: 20, // how far back should the text be taken to be
        aligned center with line
    indexDeviderIntervalHorizontal: 4,
    indexDeviderIntervalVertical: 3,
    maxPoints: 1440 // max amount of value points for the graph
};
```

Listing 4.36: Graph definition.

When the interface is told to draw a graph for the user, it uses the drawGraph function seen in Listing 4.37. This function asks a lot of sub functions to take care of the drawing. The drawing system used is called Scalable Vector Graphics, or SVG for short. This system is based on vectors, which makes it attractive for displaying graphs because it is very precise and can be scaled without any loss of precision. Another benefit from the format is the speed: It draws quickly.

Before the actual graphs are drawn, the coordinate system is generated. This is done based on the graph template.

After the coodirnate system has been drawn, the function asks the server for the values for the graph.

```
1  function drawGraph(graph) {
2
3      drawSvgContainer(graph);
4      getGraphSize(graph);
5      drawGraphBackground(graph);
6      drawGraphHorizontalLines(graph);
7      drawGraphVerticalLines(graph);
8      activeGraphs.push(graph.htmlElement);
9
10     socket.emit('getMainGraphs');
11 }
```

Listing 4.37: The "drawGraph" function.

When the interface receives the values for the graph from the server, it calls the drawGraphValues function which can be seen in 4.38. This function builds the initial point of the graph and calls the recursive function displayNextValue seen in line 6. This function takes the current plotted graph and adds the next value to it. Choosing a recursive approach to solve this problem is beneficial for updating the graph with new data in real time. This is because the interface simply is able to call the displayNextValue when it receives a new value for the graph and then the recursive function will take care of updating the graph without the need of an update function.

```
1
2  function drawGraphValues(name, graphValues, style, graph) {
3
4      ...
5
6      displayNextValue(graphValues, 0, verticalOrigin, path, pathWidth, style,
           previousPath, previousVerticalValue, previousHorizontalValue, graph,
           valuesToSkip, name);
7  }
```

Listing 4.38: Section of the "drawGraphValues" function.

4.4 Quality Control

Multiple things have been done throughout the project to ensure that the developed program is of high quality.

4.4.1 Testing

To ensure that a program is of high quality, the program must be tested well to prevent any errors occurring. Additionally, testing is also a smart way to see if the software responds correctly to all kinds of input.

Another reason as to why testing is so important when it comes to software is that a well-tested program is highly sustainable. When a new team of software developers sit down to continue working on a program made by the previous development team, it is easy to implement new code if the modules have been tested well, because it is easy to see where the mistake is when a test fails. In this way, the program becomes sustainable, which also ensures a high quality product.

In this project, Test Driven Development (TDD) has been used. TDD is a way of coding where the programmer writes the tests before writing the code itself. Next, they write the code that will make the test pass, after which they run the tests [32]. This is a good way to write code, since you ensure that the code can be tested in a way that makes sense before you start writing the code. In this way, you will not end up in a situation where your code cannot be tested after spending hours writing the code.

The type of testing that has been used in this project is unit testing. A unit is the smallest testable part of a program[32], which is why unit testing is usually done on function level. Unit testing makes it easy to spot mistakes, since the test usually only overlooks one function at a time.

For this project, 100 tests have been written to ensure that the program works as expected. During the coding of the tests, tests that were not done by the end of the day or tests that did not pass were always commented out before pushing the code to the GitHub repository. This was done to ensure there would be no confusion between the team members when pulling code others had written. By the time of project turn in, all of the tests pass.

Example of Test Code

Only one example of test code will be shown in this report. The rest of the tests are available on the project's GitHub page in the Test folder. The report will look into one of the tests written for the Device Manager, because it gives a clear general view of how the tests have been constructed. The code for this test can be found in the Appendix under Listing 7.8.

This particular test checks if the device initialization is done correctly and that a device with the correct ID is added to the database. As the comments in lines 6, 10 and 12 in the code example state, the test is split into three parts: Setup, running the test and checking that the test outputs the correct values. Most of the tests written for this project have been constructed in this way.

Firstly, in lines 7-9 everything is set up so that the test environment is ready. Here, the database is dropped to make sure nothing is on there before running the test; this ensures that no other code can have impact on the test. In line 11, the test is run by calling the deviceInit function from the Device Manager with the testDevice that was created in the setup part of the test. After running the test, lines 13 and 14 retrieve the command queue and the device, respectively. These new bindings will be used to check that the test works as intended. In lines 15-25, the test asserts on different logical statements, checking that the device is initialized and uploaded with the correct values.

The goal of the test was to check that the device was added to the database with the correct ID. If the ID was not correct, the device would not be found in the database when calling the getDevice function from the database in line 14. Therefore, the test passes if the ID is correct and the assert resolves true.

4.4.2 Version Control

The distributed version control system Git has been used to ensure that multiple people were able to work on the project simultaneously. When multiple people are working on a project simultaneously a set of challenges arise; for example pushing code that does not work properly, or multiple people working on the same file at the same time. To counter these challenges, some rules were made: These rules include only pushing code that passes all tests, and coordinating who does what in which files before starting to code.

4.4.3 REST

To make sure the implementation is of high quality the RESTfull architecture has been implemented. [33]

Client-Server

The architecture of the platform is Client-Server, because there is a single server which connects all devices and clients together and clients do not interact directly with devices and vise versa. This means that if the platform should be scaled, it is only the server part that should be scaled and all clients and devices can stay the same.

Stateless

The fact that the server is stateless means that the server should not contain states and should not behave according to a state.

All the clients connected to the server handle their states themselves. This can be seen by the device's states that is not controlled by the server. The server will simply calculate a wanted state and then request the device to change state.

The users' states are also handled by the client itself. Whenever a user resizes the browser, the client will ask for an updated graph that fits the browser's new size.

Cacheability

The platform does not use caches. However, this is due to the fact that the platform uses and manipulates data in real time, and therefore there is little to no information which can be cached.

Layered System

Instead of using caches, this means that the application is split into layers, where each layer only can interact with the layer directly above or directly under it. A good example of this in the application is how the Device Manager does not directly contact the database, but calls the DatabaseAccessor instead. This means that if the Database where to be swapped, only the DatabaseAccessor would need rewriting.

Uniform Interface

Uniform Interfaces are interfaces that a very general and help simplify the architecture of the application.

The implementation of this can be seen in the small amount of interfaces a device can call which are:

- onConnect
- receiveDeviceId
- newDeviceWithId
- stateChanged
- deviceUpdate
- disconnect

The implementation of the interfaces can be seen in Listing 7.9 in the Appendix.

This simplifies the structure of the applications because what would have been many interface functions that do mostly the same has been fused together and made more general.

For example, the deviceUpdate is used when the device wants to send an update to the server. Instead of sending a small message when something changes, it sends everything once a tick. This also means that if a message was lost in transit, the server's information about the device will only be wrong until the next update comes through, which might only be a few ticks and therefore not a big issue.

4.4.4 Clean Code

While writing the code, there was a huge emphasis on naming of variables and functions. When writing code that includes clear variable and function names, the code becomes more

readable, and comments are therefore only needed for structural part of the code. Adding too many comments to a program with good naming will result in describing everything twice. The good naming makes the program easier to maintain, because it is only the code that needs changing, whereas if there were many comments in the code, all of these should also be maintained to keep the code readable.

Furthermore, all functions are made as small as possible, to make sure they only have a single responsibility. This further improves the readability of the code because each function does not do multiple things at the same time.

The smaller functions makes it possible for each file to be split up into multiple sections: Interface functions and helper functions. Interface functions are all the functions that other files can access, and helper functions are functions each interface function can call. This means that the interface functions mostly are function calls to other functions inside of their respective files, which further increases the readability of the code.

Chapter 5: Discussion

This chapter includes a discussion of the project as a whole. Here, the project results will be presented and discussed in relation to the problem statement. Additionally, the adequacy of the end product will be discussed in detail.

5.1 Introduction to Results

This section includes a description of the end product. This description will lead to a discussion of the results in the coming sections.

The graph representation of energy surplus when running six water heaters in the system can be seen in Figure 5.1 on the next page. The coordinate system seen in this figure is a screenshot of the actual graphs drawn by the application. The coordinate system consists of two graphs: A red graph and a blue graph. When these two overlap each other, the color becomes purple.

The red graph shows a simulated RE surplus graph. Simply explained: This is the difference between how much RE production there is and the total demand for electricity at a given moment. The blue graph shows how much surplus there is when six water heaters are connected to the application.

When looking at the implementation of the water heater, it is seen that the water heater has two different stages: On and off. Initially, when the simulated water heater is set-up, it starts with a temperature of 55 °C, which is its minimum temperature. Additionally, the water heater has a maximum temperature of 90 °C. When the water heater is turned on, its temperature will increase by 0.0095 °C per second. When it is turned off, its temperature decreases by 0.000 159 °C per second.

The temperature changes mentioned above were calculated using the max wattage of the simulated water heater and the wattage usage of the simulated water heater when it needs to stay at a certain temperature. These numbers were based on a specific type of water heater, namely the "Atlantic Zeneo el vandvarmer 75 liter" [34].

The final product shows both the amount of power used when there is an RE surplus and the amount of avoided peaker plant production. These numbers are calculated based on the amount of connected devices. The values of the calculations are displayed on the frontend in the "Renewable energy stored" and the "Avoided Peaker plant production" containers, respectively. Both of these numbers are measured in kWh.

5.2 Results

In this section, the results gathered from the simulation will be presented. For the simulation, six water heaters have been used.

Figure 5.1 shows the results of running six water heaters simultaneously with a starting temperature of 55 °C. During the time between 23:00 and 24:00 the day before the screenshot in Figure 5.1 was taken, there was an RE deficit, meaning that the water heaters needed to keep their temperature at 55 °C. After running the simulation from 23:00 until 10:45 the following day, the results were captured and can be seen in Figures 5.1, 5.2, 5.3, 5.4, 5.6 and 5.7.

At around 10:45, all the water heaters were restarted with a starting temperature of 15 °C. The result of this action can be seen in Figure 5.5 on the following page and in the overall Figure 5.1:

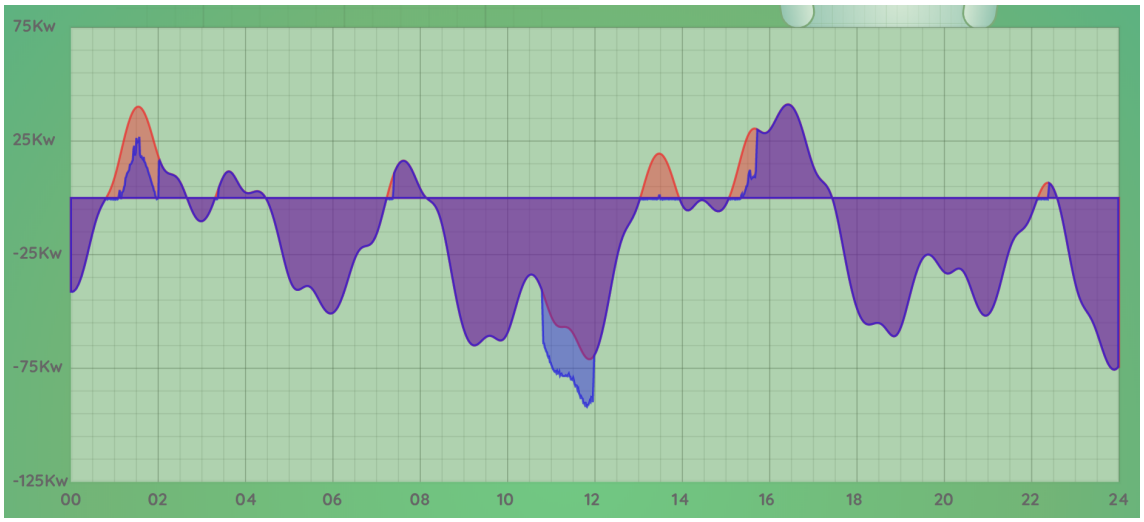


Figure 5.1: A picture of the frontend showing the impact of running all six water heaters. All of the water heaters were restarted at around 10:45.

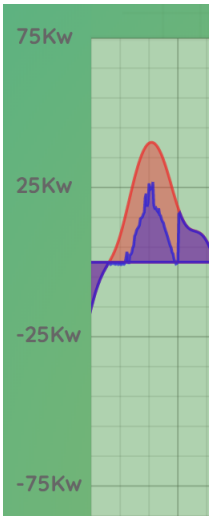


Figure 5.2: A zoomed in picture of the RE surplus between 1:00 and 2:45 from Figure 5.1.

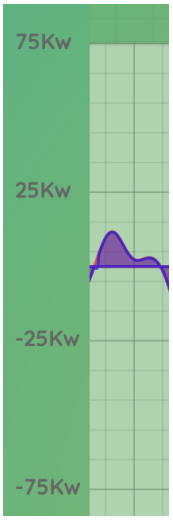


Figure 5.3: A zoomed in picture of the RE surplus between 3:15 and 4:30 from Figure 5.1.

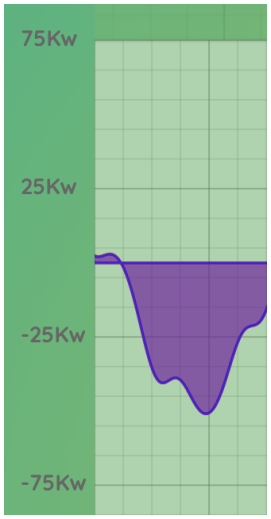


Figure 5.4: A zoomed in picture of the RE deficit between 4:30 and 7:00 from Figure 5.1.

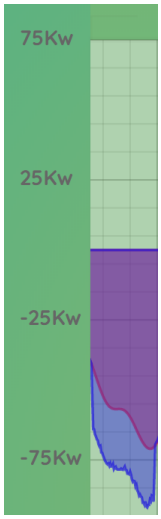


Figure 5.5: A zoomed in picture of the RE deficit between 10:45 and 12:00 from Figure 5.1.

Figure 5.6: A picture of the frontend showing the "Avoided Peaker plant production" captured at 10:45.

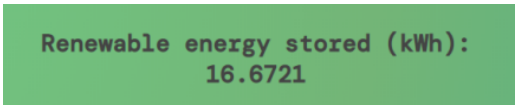


Figure 5.7: A picture of the frontend showing the "Renewable energy stored" captured at 10:45.

Sources of Inaccuracies

When simulating the real world, there is a certain amount of inaccuracies. Despite this, the project uses a lot of simulations. The reason for this choice is to save time and materials. Even with the inaccuracies, some general tendencies can be seen and a proof of concept can be evaluated.

Two of the systems that have been simulated in this project are the API data and the power demand graph. Getting access to accurate data observations for these categories requires cooperation with power regulating companies. To benefit from the precision of API data and power demand, the system would have to simulate all water heating devices within the power regulating company's network, and the devices would all need their own unique properties. Therefore, observed API data and power demand were excluded from the project.

5.3 Discussion of Results

This section aims to give a discussion of the results presented in the previous subsections. As mentioned previously, the aim of this project is to find out if a centralized IoT network can be used to stabilize the energy grid. The problem statement in Chapter 2, Section 2.5, presents the following question:

How can a centralized IoT network use flexible appliances to move energy consumption to when RE is produced and thereby help stabilize the energy grid?

The results gained from running the application for a prolonged period of time show that when there is an RE surplus, the network will try to use as much of this excess energy as possible, as seen in Figure 5.2. However, as seen in Figure 5.3, when the application has already heated up the water heaters to their maximum temperature, they cannot consume any more energy. Therefore, the blue graph starts following the red graph, indicating that the system is unable to change the current energy consumption.

When there is an RE deficit, the network uses the increased temperature to postpone turning on the water heaters. This means that if the water heater's current temperature is sufficiently high when the RE surplus graph enters a deficit, the water heater can be turned off until it reaches its minimum temperature, resulting in a lowered need for peaker plant energy. This can be seen in Figure 5.4. However, if the water heater's temperature is below the minimum temperature, then it will consume energy to heat the water up to the minimum temperature, regardless of the amount of RE surplus. This results in an increased energy consumption. An example of this can be seen in Figure 5.5.

Running the simulation for around 11 hours resulted in 2.16 kWh avoided peaker plant production and 16.67 kWh stored RE.

The *avoided peaker plant production* number means that if the six water heaters were set to keep a constant temperature, they would have consumed 2.16 kWh in a RE deficit period, resulting in a need for a peaker plants to produce that amount of energy. However, this consumption of peaker plant energy was avoided, because the water heaters were scheduled by the application.

The other number, *RE stored*, is a measure of how much of the excess RE surplus was used by the six water heaters, in comparison to the amount they would normally have consumed.

This means that as long as both of these measures are positive, this application could have helped stabilize the energy grid by using energy when there is too much RE, and not using it when there is not enough RE.

The numbers gained from the 11 hour simulation show that six water heaters can have a small impact on the energy grid. Additionally, if this application was scaled up to the national level of 1 000 000 water heaters, then the 16.67 kWh of stored RE would become 2.78 GWh and 2.16 kWh of avoided peaker plant production becomes 360 MWh. These numbers show that when the application is scaled up, it could have a large impact on the energy grid.

However, this calculation assumes that there would be enough energy surplus to enable the heating of 1 000 000 houses, which might not be the case, as this data was simulated and therefore is not a 1:1 representation of the real world.

Another inaccuracy that would have an effect on the real world are baths, showers and other types of warm water usage. When someone takes a shower, the water heater would have to heat up new water. Therefore, this extra usage results in a new need for energy consumption.

If this extra usage of warm water happens at a time of a surplus, it would benefit the energy grid, as that surplus would be used to reheat the water. However, if it happens at a time of a deficit, then it would result in an increased consumption at a time where there is not enough energy, thereby putting a bigger strain on the grid. This would also happen without the system, meaning that the extra strain on the grid happens in both cases.

It can be concluded that the system, in the worst case scenario, will have the same impact on the energy grid as when keeping the water heaters at a constant temperature. In all other cases, the application will help stabilize the energy grid.

To answer the problem statement, this project concludes that a centralized IoT system that is in control of many flexible appliances can help stabilize the energy grid. Furthermore, this can be done by turning on appliances when there is an RE surplus and turning them off when there is an RE deficit. However, if the appliance must consume energy because it has fallen below a certain threshold, the appliance will be turned on to consume as little energy as possible while the RE deficit persists.

Chapter 6: Conclusion

In this chapter, a conclusion on the project as a whole will be made. The conclusion will be formulated on the basis of the Discussion chapter, where the project's end product has been discussed in detail.

After the conclusion, a section describing future work follows. This sections aims to give some examples of different expansions that can be added to current program.

6.1 Conclusion on the Project

This report aimed to look at how the energy grid could be stabilized by reducing the impact of when RE production takes place. Which becomes an increasing part of the future energy grid due to the increased focus on RE and sustainability.

The problem statement presents the following question:

How can a centralized IoT network use flexible appliances to move energy consumption to when RE is produced and thereby help stabilize the energy grid?.

To answer the problem statement, an IoT network has been developed. With the help of the program, a simulation was run, and the results gained from this simulation were discussed in Chapter 5. In the Discussion Chapter, it was found that a centralized IoT system would be able to help stabilize the energy grid by turning on appliances when there is an RE surplus, and turning them off when there is an RE deficit. However, if the appliance must consume energy because it has fallen below a certain threshold, the appliance will be turned on to consume as little energy as possible while the RE deficit persists.

6.2 Future Work

In this section, all future work on the project will be described. This is the work that would been done if the project time frame had been extended. If the same project group or a new group of developers were to further develop this project, they should look at this section to get insight on the extensions that need to be implemented.

6.2.1 Implementation of the Washing Machine Device

As mentioned in Chapter 4, Section 4.1.2, the washing machine device has not been implemented into the program by the time of project turn-in. This was due to a lack of time in the project, meaning there was not enough time to fully implement the washing machine device with all its functionality. Therefore, the project group decided not to implement this device, and simply implementing the fully autonomous water heater device instead, since this device requires less complex functionalities.

If the project time frame had been extended, the washing machine device would have been implemented into the code. This would be done by following state machine of the washing machine, shown in Chapter 3, Section 3.5.2, as has been done with the water heater and its respective state machine.

As mentioned earlier, the washing machine device is a more complex device, because does not fall under the category of fully autonomous devices like the water heater does. Its higher complexity means that the washing machine will be handled differently by the program, and that it needs more functionalities than the water heater.

Some of the functionalities that a washing machine needs, that have not already been implemented within the original time frame, are the following:

- Scheduling and re-scheduling of a device
- Removing a device schedule

When looking at the functionalities that the washing machine needs to be implemented fully, it is seen that both scheduling and re-scheduling of a device as well as removing a device schedule are Scheduler functionalities. These functionalities have both been modelled and explained in Chapter 3, Section 3.3. They would be implemented into the program in accordance with the models.

6.2.2 Implementation of Additional Devices

Initially, as discussed in the Problem Analysis Chapter 2, the long-term goal for the project was that any standardized type of device should be able to connect to the program, although this was not possible within the original time frame. However, if this project was to be expanded further than just the inclusion of the washing machine device, the program should be programmed to include any standardized type of device. Examples of these devices are microwaves, ovens and robotic lawnmowers.

The devices would be handled differently by the program depending on whether they are autonomous or not. However, after having implemented one type of autonomous device into the program, it should be fairly easy to implement a different type of autonomous device. For example, if the water heater device is already implemented, it should not be difficult to implement a robotic lawnmower afterwards, since they are both autonomous and should be handled very similarly by the software.

6.2.3 Security and Authentication

Authentication is an important part of a consumer-ready product, but it was quickly ruled out as a part of the project scope. Proper authentication of the program would require development of login functionalities, secure session cookies, proper database security, and hash encryption for passwords. On top of the login security issues, development of unique user pages on which the user only has access to their own devices, would be another important feature.

Unique user interfaces would also require a way for the user to add and remove their devices from the network through the the frontend user interface. Additionally, the unique user interfaces would also require a way for the database to keep track of who owns the different devices.

One security feature that has been used is the UUIDv4 system for device IDs, which is a randomly generated 128bit number[31], as explained in Chapter 4, Section 4.1.1. This makes it difficult for any potential attackers to guess the device ID of any devices through the use of brute force attacks.

To conclude on this subsection, it is noted that some minimum level of security and authentication would have been implemented into the program if the project time period had been extended. However, because it was considered out of scope for this project, security and authentication has not been a focus point, although it generally is important for any product that is consumer-ready.

6.2.4 Gathering of API Data from a Weather Forecasting API

As mentioned in Chapter 4, Section 4.2.2, it was not possible to get access to an API during the original time period of the project. Therefore, RE demand data and RE production data was simulated instead.

However, if the time period of the project had been extended, the original plan was to hook up the server to a weather forecasting API, such as the ones of Danish energy companies *Energinet*[14] and *DMI*[35]. This would be done by accessing the respective APIs and connecting these to the developed program, so that the program would be able to access the API data.

If the connection to one or both of the aforementioned APIs were established, the program would not need to simulate RE data any longer. Instead, this data would be available through the APIs, meaning the program would be able to access real weather data. This means that the graphs drawn on the frontend application would be drawn in accordance to the current weather data that has been gathered from the API.

It is seen that using APIs to forecast weather data would end up yielding results that are based on real data, instead of simulated data. Therefore, getting access to an API and implementing it in the program would be an important expansion to the implementation if the project time frame had been extended.

Chapter 7: Appendix

7.1 The setState function from App

```
function setState(waterHeater) {
    let uniqueProperties = waterHeater.uniqueProperties;

    if (uniqueProperties.currentTemp > uniqueProperties.maxTemp) {
        changeStateToOff(waterHeater);
    } else if (waterHeater.isConnected === false &&
        uniqueProperties.currentTemp > uniqueProperties.minTemp &&
        waterHeater.currentState === "on") {
        changeStateToOff(waterHeater);
    } else if (uniqueProperties.currentTemp <= uniqueProperties.minTemp) {
        changeStateToOn(waterHeater);
    } else if (waterHeater.isConnected === true &&
        waterHeater.serverMessage === "off" &&
        waterHeater.currentState === "on") {
        changeStateToOff(waterHeater);
        waterHeater.serverMessage = null;
    } else if (waterHeater.isConnected === true &&
        waterHeater.serverMessage === "on" &&
        waterHeater.currentState === "off") {
        changeStateToOn(waterHeater);
        waterHeater.serverMessage = null;
    } else if (waterHeater.onDisconnect === true &&
        waterHeater.currentState === "on") {
        changeStateToOff(waterHeater);
        waterHeater.serverMessage = null;
    }
}
```

Listing 7.1: The "setState" function from the water heater's *app*.

7.2 Functions from the User Manager

```
async function sendAPISurplusGraph(socket, date) {
    return new Promise(async (resolve, reject) => {
        if (process.env.PORT) {
            date.setHours(date.getHours());
        }
        for (var i = 0; i < 24; i++) {
            let id = util.dateToId("apiSurplusGraph", date);
            let graph = await dbG.getGraph(id);
            let payload = {
                name: "apiSurplusGraph",
                hour: i,
                values: graph.values
            }
            createCommand(socket, "createGraphValues", payload);
            date = incrementHour(date, 1);
        }

        let endOfGraphPayload = {
            name: "apiSurplusGraph",
            hour: -1,
            values: "done"
        }
        createCommand(socket, "createGraphValues", endOfGraphPayload);

        resolve(true);
    });
}
```

Listing 7.2: The sendAPISurplusGraph from the *UserManager*.

```
async function sendSurplusGraph(socket, date) {
    return new Promise(async (resolve, reject) => {
        lastGraphUpdate = new Date(date.getTime());
        let hours = date.getHours();
        date.setHours(0);

        for (var i = 0; i < hours; i++) {
            let id = util.dateToId("surplusGraph", date);
            let graph = await dbG.getGraph(id);
            let payload = {
                name: "surplusGraph",
                hour: i,
                values: graph.values
            }
            createCommand(socket, "createGraphValues", payload);
            date = incrementHour(date, 1);
        }

        let minutes = date.getMinutes();
        let id = util.dateToId("surplusGraph", date);
```

```
        let graph = await dbG.getGraph(id);
        let values = [];
        for (var i = 0; i < minutes; i++) {
            values.push(graph.values[i]);
        }
        let payload = {
            name: "surplusGraph",
            hour: hours,
            values: values
        }
        createCommand(socket, "createGraphValues", payload)

        let endOfGraphPayload = {
            name: "surplusGraph",
            hour: -1,
            values: "done"
        }
        createCommand(socket, "createGraphValues", endOfGraphPayload);
        resolve(true);
    });
}
```

Listing 7.3: The sendSurplusGraph from the *UserManager*.

```
async function graphUpdate() {
    let date = new Date()
    if (lastGraphUpdate !== undefined &&
        lastGraphUpdate.getMinutes() === date.getMinutes()) {

        return false;
    }
    lastGraphUpdate = new Date();
    return new Promise(async (resolve, reject) => {
        date = new Date(); // 10:12
        date.setMinutes(date.getMinutes() - 1); // 10:11
        let graph = await dbG.getGraph(util.dateToId("surplusGraph", date));
        let minute = date.getMinutes();
        let point = [graph.values[minute]];
        let payload = {
            name: "surplusGraph",
            hour: date.getHours(),
            minute: minute,
            values: point
        }
        createCommand("userSpace", "graphValueUpdate", payload);
    });
}
```

Listing 7.4: The "graphUpdate" function from *UserManager*.

7.3 ForecasterAPI Mathematical Functions

```
function getDemandAt(x) {
    return (Math.sin(0.9 * x - 5) * 2.2 +
        Math.sin(1.4 * x - 0.2) * 1.3 +
        Math.sin(2.1 * x - 2) * 2.8 + lift) * scale + 1000000;
}

function getProdutionAt(x) {
    return (Math.sin(2.7 * x + 3.3) * 0.8 +
        Math.sin((-0.5) * x + 1) * 2.7 +
        Math.sin(Math.sin(2.1 * x - 2) * 2.8) + lift) * scale;
}
```

Listing 7.5: The mathematical functions of *ForecasterAPI*.

7.4 The scheduleDevice function from Scheduler

```
async function scheduleDevice(device) {
    return new Promise(async (resolve, reject) => {
        let date = new Date();
        graphId = util.dateToId("surplusGraph", date);
        surplusGraph = await daG.getGraph(graphId);
        let schedule = {
            start: date
        }
        // If there is surplus and the device is not scheduled to "on"
        if (surplusGraph.values[date.getMinutes()] > 0 && device.nextState !==
            "on") {
            await daD.updateDevice(device.deviceId, "nextState", "on");
            await daD.updateDevice(device.deviceId, "isScheduled", true);
            await daD.updateDevice(device.deviceId, "schedule", schedule);

            addUpdatedDevice(device.deviceId);

            resolve(true);
        }
        // If there is no surplus and the device is not scheduled to "off"
        else if (surplusGraph.values[date.getMinutes()] <= 0 && device.nextState
            !== "off") {
            await daD.updateDevice(device.deviceId, "nextState", "off");
            await daD.updateDevice(device.deviceId, "isScheduled", true);
            await daD.updateDevice(device.deviceId, "schedule", schedule);

            addUpdatedDevice(device.deviceId);

            resolve(true);
        } else {
            resolve(false);
        }
    });
}
```

Listing 7.6: The "scheduleDevice" function from *Scheduler*.

7.5 The stateChanged function from Device Manager

```
async function stateChanged(id, newState) {
    let device = await db.getDevice(id);
    if (device === null) {
        return false;
    }
    device.currentState = newState;

    if (newState === "off") {
        device.isScheduled = false;
        device.nextState = null;
        device.schedule = null;
        device.scheduledInterval = null;
    }

    return new Promise(async (resolve, reject) => {
        updateDevice(device, true)
            .then((val) => {
                resolve(true);
            })
            .catch((err) => {
                reject(err);
            })
    })
}
```

Listing 7.7: The "stateChanged" function from *DeviceManager*.

7.6 Test function from Device Manager

```
if (true) {
    describe('Device Manager', () => {
    ...
        // Device init
        it('deviceInit: Add device with correct id', async () => {
            // Setup
            await db.dropDatabase();
            let testDevice = createAutoTestDevice();
            let id = testDevice.deviceId;
            // Run
            let res = await dm.deviceInit(testDevice, "socket")
            // Get things to check
            let commandQueue = dm.getCommandQueue();
            let dbDevice = await db.getDevice(id);
            assert(res === true &&
                commandQueue !== undefined &&
                commandQueue.length === 0 &&
                dbDevice.scheduledByUser === false &&
                dbDevice.isScheduled === false &&
                dbDevice.nextState === null &&
                dbDevice.schedule === null &&
                dbDevice.scheduledInterval === null &&
                dbDevice.uniqueProperties.currentTemp ===
                    testDevice.uniqueProperties.currentTemp &&
                dbDevice.programs.length === testDevice.programs.length &&
                dbDevice.deviceType === testDevice.deviceType
            );
        })
        ...
    })
}
```

Listing 7.8: One of the many tests from *DeviceManager*.

7.7 Server Interface for Devices

```
const deviceSpace = io.of('/device');
deviceSpace.on('connection', (socket) => {
    dm.onConnect(socket);

    socket.on('receiveDeviceId', (id) => {
        console.log("---receiveDeviceId");
        console.log(id);
        dm.receiveId(id, socket);
    });

    socket.on('newDeviceWithId', (deviceInfo) => {
        console.log("---newDeviceWithId");
        dm.deviceInit(deviceInfo, socket);
    });

    socket.on('stateChanged', (state, id) => {
        console.log("---stateChanged to " + state);
        dm.stateChanged(id, state);
    });

    socket.on('deviceUpdate', (device) => {
        dm.updateDevice(device);
    });

    socket.on('disconnect', () => {
        console.log("---disconnect");
        console.log();
        dm.onDisconnect(socket);
    });
});
```

Listing 7.9: All functions a device can call.

Acronyms

API Application Programming Interface. 46, 76

APP Application File. 20, 21, 23, 34, 42, 51, 52, 56–58

EV Electric Vehicle. 4

GUI Graphical User Interface. 58–60

GWh Gigawatt Hours. 7

IoT Internet of Things. 1, 2, 5, 7, 9–11, 13–18, 32, 72–74

kWh Kilowatt Hours. 6, 7, 69

MWh Megawatt Hours. 6

RE Renewable Energy. 1, 2, 5–9, 12–14, 17, 18, 32, 44, 46, 54, 55, 69–74, 76

TDD Test Driven Development. 65

UI User Interface. 58, 62

UUID Universally Unique Identifier. 41, 76

References

[1] NASA. *Scientific Consensus: Earth's Climate is Warming*. Accessed on 21/05/2020. URL: https://climate.nasa.gov/scientific-consensus/.

[2] Sandra Laville & Jonathan Watts. *Across the globe, millions join biggest climate protest ever*. Accessed on 21/05/2020. URL: https://www.theguardian.com/environment/2019/sep/21/across-the-globe-millions-join-biggest-climate-protest-ever.

[3] Hannah Ritchie & Max Roser. *Renewable Energy*. Accessed on 21/05/2020. URL: https://ourworldindata.org/renewable-energy.

[4] ITU. *Internet of Things Global Standards Initiative*. Accessed on 26/02/2020. https://www.itu.int/en/ITU-T/gsi/iot/Pages/default.aspx, 2012.

[5] Dansk Energi Energinet. *TENDENSER OG FREMTIDSPERSPEKTIVER FOR ELSYSTEMET*. https://energinet.dk/Analyse-og-Forskning/Analyser/RS-Analyse-juni-2019-Tendenser-og-fremtidsperspektiver, 2019, (p. 1–7).

[6] Poul Sørensen et al. *Power Fluctuations From Large Wind Farms*. http://henrikmadsen.org/wp-content/uploads/2014/05/Journal_article_-_2007_-_Power_Fluctuations_From_Large_Wind_Farms.pdf, 2019, (p. 958).

[7] Joachim Geske and Diana Schumann. "Willing to participate in vehicle-to-grid (V2G)? Why not!" In: *Energy Policy* 120 (2018), pp. 392–401. ISSN: 0301-4215. DOI: https://doi.org/10.1016/j.enpol.2018.05.004. URL: http://www.sciencedirect.com/science/article/pii/S0301421518302982.

[8] M. Mancini et al. "High performance, environmentally friendly and low cost anodes for lithium-ion battery based on TiO2 anatase and water soluble binder carboxymethyl cellulose". In: *Journal of Power Sources* 196.22 (2011), pp. 9665–9671. DOI: 10.1016/j.jpowsour.2011.07.028. URL: https://www.sciencedirect.com/science/article/pii/S0378775311013668.

[9] Andrew W. Thompson. "Economic implications of lithium ion battery degradation for Vehicle-to-Grid (V2X) services". In: *Journal of Power Sources* 396 (2018), pp. 691–709. DOI: 10.1016/j.jpowsour.2018.06.053. URL: https://www.sciencedirect.com/science/article/pii/S0378775318306499.

[10] U.S. Energy Information Administration. "International Energy Outlook 2016". In: *Journal of Power Sources* (2016). Accessed on 27/05/2020, p. 113. URL: https://www.eia.gov/outlooks/ieo/pdf/industrial.pdf.

[11] safewise.com. *What Is Home Automation and How Does it Work?* Accessed on 08/04/2020. URL: https://www.safewise.com/home-security-faq/how-does-home-automation-work/.

[12] robots.net. *Top 10 Most Popular IoT Devices In 2020*. Accessed on 08/04/2020. URL: https://robots.net/tech-reviews/top-iot-devices/.

[13] Google.com. *Learn about Nest thermostats before you buy*. Accessed on 08/04/2020. URL: https://support.google.com/googlenest/answer/9244917?hl=en.

[14] energinet.dk. *Energinet*. Accessed on 20/05/2020. URL: https://energinet.dk/.

[15] The New York Times. *Do You Suffer From Decision Fatigue?* Accessed on 08/04/2020. URL: https://www.nytimes.com/2011/08/21/magazine/do-you-suffer-from-decision-fatigue.html.

[16] Metro Therm. *ELECTRICAL WATER HEATERS*. URL: https://www.metrotherm.dk/download/18.39f077dc16c43097ebb1bcc4/1568961412312/Electric-water-heaters-60-110-160-Smart-Control-manual-08013-1604.pdf.

[17] Bygningsreglement 2018. *§411*.

[18] Danmarks Statistik. *Savings Project: Lower Water Heating Temperature*. Accessed on 08/04/2020. URL: https://www.dst.dk/da/Statistik/emner/levevilkaar/boligforhold/boliger#.

[19] omnicalculator.com. *Specific Heat Calculator*. Accessed on 21/05/2020. URL: https://www.omnicalculator.com/physics/specific-heat#heat-capacity-formula.

[20] sparenergi.dk. *Sådan bruger du strømmen*. Accessed on 08/04/2020. URL: https://sparenergi.dk/forbruger/el/dit-elforbrug.

[21] simplyswitch.com. *Which household appliances use the most electricity?* Accessed on 08/04/2020. URL: https://www.simplyswitch.com/household-appliances-use-electricity/.

[22] bolius.dk. *Så meget el, vand og varme bruger en gennemsnitsfamilie*. Accessed on 08/04/2020. URL: https://www.bolius.dk/saa-meget-el-vand-og-varme-bruger-en-gennemsnitsfamilie-279.

[23] U.S. Department of Energy. *Savings Project: Lower Water Heating Temperature*. Accessed on 08/04/2020. URL: https://www.energy.gov/energysaver/services/do-it-yourself-energy-savings-projects/savings-project-lower-water-heating.

[24] Birgitte Merci Lund Per Holck Jens Kraaer. *Orbit B HTX*. Paperback ISBN: 978-87-616-2465-9. Systime, 2009, p. 44.

[25] Henrik W. Bindner Anders Thavlov. "Thermal Models for Intelligent Heating of Buildings". In: *Proceedings of the International Conference on Applied Energy, ICAE 2012* (2012), p5–9.

[26] Gate 21 and Region Hovedstaden. *ROADMAP 2025*. Tech. rep. 2018, p. 52. URL: https://www.gate21.dk/wp-content/uploads/2018/11/EPT_Roadmap-2025_WEB.pdf.

[27] propercloth.com. *How to Wash a Dress Shirt*. Accessed on 08/04/2020. URL: https://propercloth.com/reference/how-to-wash-a-dress-shirt/.

[28] Accenture. *The New Energy Consumer: Unleashing Business Value in a Digital World*. Accessed on 01/04/2020. https://www.accenture.com/_acnmedia/accenture/next-gen/insight-unlocking-value-of-digital-consumer/pdf/accenture-new-energy-consumer-final.pdf, 2015, p. 6, p. 25–26.

[29] gigamon.com. *Understanding Single Points of Failure (SPOF)*. Accessed on 08/04/2020. URL: https://blog.gigamon.com/2018/08/31/understanding-single-points-of-failure/.

[30] itemis.com. *What is a state machine?* Accessed on 27/04/2020. URL: https://www.itemis.com/en/yakindu/state-machine/documentation/user-guide/overview_what_are_state_machines.

[31] npmjs.com. *uuid - npm*. Accessed on 21/05/2020. URL: https://www.npmjs.com/package/uuid.

[32] smartbear.com. *What Is Unit Testing?* Accessed on 14/05/2020. URL: https://smartbear.com/learn/automated-testing/what-is-unit-testing/.

[33] restfulapi.net. *What is REST*. Accessed on 21/05/2020. URL: https://restfulapi.net/.

[34] atlantic-comfort.com. *Electric Water Heaters | Zeneo*. Accessed on 26/05/2020. Choose "75L Vertical wall mounted Turbo" type when looking at the Technical Data specifications. URL: https://www.atlantic-comfort.com/Our-products/Water-heating/Electric-water-heaters/Zeneo.

[35] dmi.dk. *Vejr fra DMI*. Accessed on 20/05/2020. URL: https://www.dmi.dk/.