Thomas Borge Skøien

An efficient and scalable software for the IoT management of multiple smart houses

Master's thesis in Cybernetics and Robotics
Supervisor: Sebastien Gros
June 2023

Thomas Borge Skøien

An efficient and scalable software for the IoT management of multiple smart houses

Master's thesis in Cybernetics and Robotics
Supervisor: Sebastien Gros
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Abstract

This thesis presents the development of a versatile and scalable infrastructure for the monitoring and control of various Internet-enabled Internet of Things (IoT) devices. The primary objective is to reduce energy consumption and costs through cost-effective thermal-response control in Norwegian homes and offices. The infrastructure, constructed using Kubernetes, Django, and InfluxDB, permits seamless integration of new devices, effectively logging their data and providing an easy-to-use platform for data retrieval. Users can engage with the system by logging device data from their homes, allowing comprehensive data analysis for researchers and providing users access to advanced control algorithms.

Historically, the integration of IoT devices from different vendors has been challenging due to diverse communication methods. However, the recent surge in user-friendly IoT devices has made remote data access and device control a reality. The system presented in this thesis has effectively collected data from a variety of IoT devices across multiple vendors and homes, thereby laying a robust foundation for the essential data gathering required for future research. The lack of data on domestic demand response for research is one of the main motivations for the system. While commercial vendors providing similar services exist, the drive to create this system is reinforced by factors including GDPR compliance, its intended sole use for research, and the advantageous prospect of the research group having its own dedicated tool.

The retrieved data has been used to complete the control loop by sending inputs back to some of these devices, and this has been successfully demonstrated with both Proportional-Integral (PI) control and Model Predictive Control (MPC). This accomplishment, a system made to be scalable using affordable, off-the-shelf hardware without an in-house central hub, is what sets it apart. Effective control algorithms also mean that users can realize substantial energy and cost savings. Enhanced performance can be achieved by integrating sensor data with real-time spot market prices and weather forecasts. Importantly, the system's adaptability extends beyond thermal-response control; it can be tailored to interact with other systems, such as ventilation and in-house battery systems. The collected data will also support the development of learning-based predictive control algorithms, contributing to further advancements in this field.

Sammendrag

Denne avhandlingen presenterer utviklingen av en allsidig og skalerbar infrastruktur for overvåkning og kontroll av forskjellige internettkoblede Tingenes Internett (IoT)-enheter. Hovedmålet er å redusere energiforbruket og kostnadene gjennom kostnadseffektiv termisk responskontroll i norske hjem og kontorer. Infrastrukturen, som er bygget på Kubernetes, Django og InfluxDB, tillater sømløs integrasjon av nye enheter, logger effektivt deres data og gir en brukervennlig plattform for datahenting. Brukere kan melde seg inn i systemet og deretter logge sensordata fra sine hjem, noe som tillater omfattende dataanalyse for forskning og gir brukerne tilgang til avanserte kontrollalgoritmer.

Historisk har integrasjonen av diverse IoT-enheter vært utfordrende grunnet varierte kommunikasjonsmetoder. Men ankomsten av flere brukervennlige IoT-enheter har gjort fjerntilgang av data og kontroll til en realitet. Systemet utviklet i denne avhandlingen har lykkes med å logge data fra et utvalg av IoT-enheter fra forskjellige leverandører og hjem, og dermed etablert grunnlaget for den nødvendige datainnsamling for fremtidig forskning på feltet. Denne mangelen på forskningsdata som omhandler etterspørselsrespons i vanlige hus er den største motivasjonen bak dette systemet. Til tross for tilstedeværelsen av lignende kommersielle løsninger, er behovet for dette systemet begrunnet med flere faktorer. Disse inkluderer systemets overensstemmelse med GDPR-retningslinjer, det faktum at det utelukkende vil bli brukt til forskningsformål, og fordelen ved å ha et dedikert, internt verktøy for datainnsamling innen forskningsgruppen.

Den innsamlede dataen har blitt brukt til å fullføre kontrollsløyfen ved å sende kontroll tilbake til noen av disse enhetene, noe som har blitt demonstrert med både Proporsjonal-Integrasjon (PI) kontroll og Modellprediktiv Styring (MPC). Det unike med dette systemet er dets evne til å skalere opp uten behov for en sentral kontroll-hub innad i huset, ved å bruke kostnadseffektive IoT enheter som forbrukere enkelt kan anskaffe. Effektive kontrollalgoritmer betyr også at brukerne kan oppnå betydelige energi- og kostnadsbesparelser. Forbedret ytelse kan oppnås ved å integrere sensordata med sanntids spotmarkedspriser og værprognoser. Systemets tilpasningsevne strekker seg også utover termisk responskontroll. Det kan tilpasses for å samhandle med andre systemer som ventilasjon og innebygde batterisystemer. Den innsamlede dataen vil også støtte utviklingen av læringsbaserte prediktive kontrollalgoritmer, noe som vil bidra til videre forskning på dette feltet.

Preface

This master's thesis signifies the finale of my enriching journey through the Master of Science degree in Cybernetics and Robotics at the Norwegian Institute of Science and Technology (NTNU).

Rooted in the ongoing research initiative *POWIOT*, under the guidance of Sebastien Gros, this exploration delved into the multifaceted realm of smart houses and energy conservation. It was a captivating study that broadened my perspectives and deepened my knowledge in this increasingly vital field.

The essence of this thesis takes inspiration from the diligent work of a previous master's student [1], as well as building upon my project report [2], with an envisioned goal of serving as a foundational infrastructure for subsequent students to continue working upon. The frameworks used, Django, Docker, Kubernetes, InfluxDB and PostgreSQL, are naturally not a part of the implementation in this thesis. However, all code on top of these frameworks, the code connecting the modules together, as well as the various device integrations, are a part of this thesis. Some ideas were however taken from the project thesis work in the autumn. For more information about this, look in Chapter 4.

My immersion into this project has been a learning experience, from understanding the intricate layers of the smart-house domain to developing software for it and efficiently executing control schemes on the established infrastructure. The journey was peppered with a mixture of successful decisions and missteps, all contributing to an invaluable learning curve.

I extend heartfelt thanks to Sebastien Gros, whose consistent support and mentorship shaped my thesis journey, going above and beyond by offering his own home as a practical experimentation site. I am also grateful to Dirk Peter Reinhardt for his valuable feedback that significantly improved my work. Equally, I owe a debt of gratitude to my mother, Ingeborg, for her generosity in allowing data collection from her home, and to my father, Thor Erling, whose quarterly visits and fridge refills kept me going throughout these years. I deeply appreciate my girlfriend, Kristianne, for her constant support and motivation. Finally, the Morning Coffee Group Meeting (MCGM) at Gamle Fysikk deserves special recognition for instilling a sense of routine, camaraderie, and serving as a much-needed daily wake-up call.

Contents

Figures

Tables

Acronyms

API Application Programming Interface. 1, 5–10, 13, 15, 18, 19, 21–24, 26–28, 30, 31, 33–37, 39, 43, 46, 48, 50, 51, 53, 54, 56–59, 61, 63, 75–79, 86, 87, 96, 98

DEC Department of Engineering Cybernetics. xi, 39, 74, 79

DRF Django Rest Framework. 15

EV Electric Vehicle. 8, 9

GDPR General Data Protection Regulation. iii, v, 18, 84, 91, 94, 95, 102

HTTP Hypertext Transfer Protocol. 29

HTTPS Hypertext Transfer Protocol Secure. 28

HVAC Heating, Ventilation, and Air Conditioning. 3, 9

IFTTT IF This Then That. 8

IoT Internet of Things. iii, v, 1–3, 7, 9, 10, 15, 18, 28, 97, 101

MCGM Morning Coffee Group Meeting. vii

MHE Moving Horizon Estimation. 11

MPC Model Predictive Control. iii, v, xi, 1, 3, 4, 9–11, 17, 40, 74, 79, 80, 96

MVC Model-View-Controller. 14

NTNU Norwegian Institute of Science and Technology. vii, 1, 9, 29, 71, 97

PI Proportional-Integral. iii, 40

PID Proportional-Integral-Derivative. 4, 10

SDGs United Nations Sustainable Development Goals. 97

SSH Secure Shell. 52

SSL Secure Sockets Layer. 14

TVOC Total Volatile Organic Compounds. 8, 33

VRM Victron Remote Management. 37

Glossary

JSON stands for JavaScript Object Notation, which is a lightweight, human-readable, and widely used data interchange format [3]. It is language-independent and easy for humans to read and write, as well as for machines to parse and generate. JSON is often used for asynchronous browser-server communication, and as a file format for storing and exchanging data between applications. 15, 28, 60, 115–124

MQTT stands for Message Queuing Telemetry Transport, which is a lightweight, open-source, and publish-subscribe messaging protocol designed for constrained environments and low-bandwidth, high-latency, or unreliable networks [4].. 10, 14, 23, 26, 28, 37, 57, 58, 63, 68

namespace in the context of Kubernetes, a namespace is a logical partition within a cluster that enables multiple teams or applications to share the same physical infrastructure while isolating their resources and policies [5]. It provides a way to organize and manage different environments, such as development, staging, and production. 53, 54

ORM is a programming technique that facilitates the conversion of data between incompatible, object-oriented programming languages and relational database systems [6]. ORM enables developers to interact with databases using the syntax and constructs of their programming language, abstracting away the underlying SQL code. This process simplifies data management and manipulation, allowing developers to work with databases more efficiently and with a higher level of abstraction. 14, 16, 30

proxy in computer networking, is an intermediary server that acts on behalf of clients seeking resources from other servers [7]. It acts as a gateway between the client and the target server, forwarding requests and responses between them. The primary purpose of a proxy is to enhance performance, security, and privacy. 29, 45, 69

XML stands for eXtensible Markup Language, which is a markup language designed to store, transport, and organize structured data in a human-readable format [8]. XML allows users to define their own tags and attributes to describe the structure and semantics of the data, making it highly versatile and applicable across various industries and use cases. It is often used for data exchange between systems and for the configuration of applications. 15

Chapter 1

Introduction

Venturing into the world of smart homes is akin to finding oneself in a choose-your-own-adventure story, brimming with choices of numerous gadgets and gizmos claiming to save you x amount on your energy bill. While many of these contenders offer notable benefits, some are pretty limited in their decision-making, and others cost a lot to implement. All of which have a varying degree of insight into your own data. However, imagine a scenario where these various IoT devices in homes could be integrated without adding a new gadget. This study offers a backstage pass to such a future, one that could not only be more intelligent but also more economical.

1.1 Description

Navigating the landscape of smart homes reveals a plethora of solutions geared towards augmenting home intelligence. These solutions exhibit a range of effectiveness and adopt diverse mechanisms. This study aims to harness the power of affordable equipment to interface with energy-centric appliances such as heat pumps and ventilation systems across multiple houses. Furthermore, it is to employ sophisticated algorithms that enhance operational efficiency without necessitating the installation of additional devices in the home. The approach proposed here takes advantage of the open Application Programming Interface (API) of existing energy-related home devices, performing the algorithmic computations remotely on an NTNU server. A high level overview of the total system can be found in Figure 1.1. Here hybrid digital twins as well as reinforcement learning is shown as ways of further improving Model Predictive Control (MPC) control.

The core emphasis of this study is on the design and development of a scalable software infrastructure capable of collecting data from a multitude of devices spread across various homes. It also proposes a backend that allows individuals to effortlessly log in, select their home devices, and initiate the data logging process for their devices. The study then evolves to demonstrate a practical control use case, further showcasing the utility and potential of this approach.

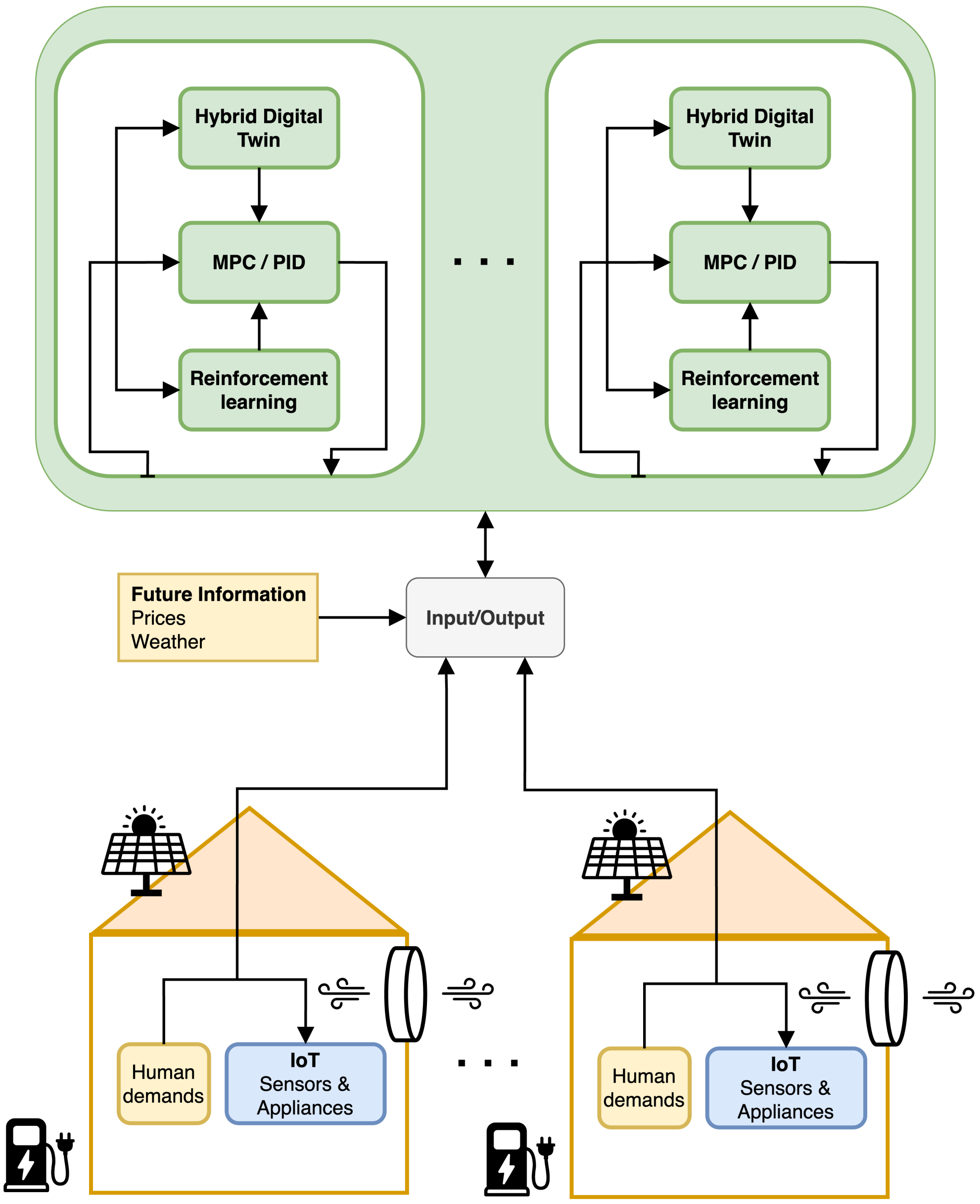


Figure 1.1: Showing a high level overview of the proposed control system. Multiple houses can connect to the same system, each with their own control algorithms. These control algorithms are running on some kind of server, doing calculations based on sensor data and sending actuations back to the corresponding IoT devices.

1.2 Motivation

This initiative seeks to build upon the work of Eric Törn [1], who, in his master's thesis, constructed a comprehensive system centred around logging data from a few publicly accessible IoT devices. This system aimed at closing the MPC heat pump control loop within a house, and its effectiveness was validated within a single house, showing promise for its adaptability to other homes.

This project aims to expand upon this foundation and provide an accessible and intelligent control solution capable of optimizing various aspects of a smart home using inexpensive, readily available IoT hardware.

One of the key observations from Törn's work was the potential for a more remote solution [1]. Despite being operational on a Raspberry Pi, the system was not tied to the house's geographical location, suggesting the system's potential to operate independently of the house's physical location. This study aims to explore this potential and develop a system that can log data and send control to IoT devices remotely.

Such an arrangement could allow the system to expand without the necessity of hiring additional technical staff, thereby providing scalability, which would, in turn, greatly increase the possibilities of more data for research. However, to facilitate this scalability, a re-engineering of the software from the ground up is required, incorporating principles that ensure it can cope with an expanding user base.

1.2.1 Norway: A Prime Location for Smart Homes and Energy Efficiency

Norway is an optimal location for the advancement and implementation of smart homes, given its well-connected society and favourable infrastructure. This is evidenced, for example, by the mandatory installation of smart power meters in all residences, which collect and transmit data on energy consumption [9]. Furthermore, in 2020, 61% of all energy used in Norway came from renewable sources [10]. Thus, using smart electrical devices for Heating, Ventilation, and Air Conditioning (HVAC) is a practical and viable solution for homes.

In addition, the demand for such solutions is greater in Norway due to the country's cold climate. With high energy costs and low temperatures, the benefits of these solutions could outweigh their development expenses. Over the years, energy consumption in Norwegian households has steadily risen. In 2017, residential buildings accounted for 22% of the country's total energy usage [11]. It is hoped that this figure will decrease with the implementation of smarter housing, including energy-efficient grids and individual homes that consume less energy while working in harmony.

1.2.2 Smart-home Control and the Gap in the State of the Art

One of the primary motivations for collecting data in smart homes is to utilize it to effectively control various devices and systems. The simplest form of control can be observed in thermostats, which regulate devices based on a set temperature by issuing either on or off signals. Moving forward in complexity, Proportional-Integral-Derivative (PID) controllers can be introduced to further refine the control process. The most advanced control systems involve predictive algorithms, such as Model Predictive Control (MPC), which can anticipate future conditions and adjust controls accordingly. However, these more advanced control schemes require more data, and the interoperability between these devices is not trivial. This research data, which there is a lack of on domestic response control, is needed to improve these control systems further.

This interoperability challenge is evident in most vendor-offered heating solutions, of which several will be detailed in Section 2.1, which vary significantly in their effectiveness and customizability. These systems often necessitate the exclusive use of devices within a specific ecosystem, thereby increasing costs for a comprehensive solution. This limitation highlights a notable gap in the market: the lack of advanced and customizable heating control systems that can operate across different ecosystems.

Though custom integrations are possible within some ecosystems, they are often limited to basic on/off controls based on the time of day or current utility price. Implementing and maintaining more advanced control algorithms, such as MPC, requires extensive technical expertise.

It has been shown by a previous master's student that heating control using cheaper devices and their data is possible, and it would therefore be interesting to see whether this is true also on a bigger scale [1]. The interconnected system offers an additional advantage: when multiple houses are connected, they can potentially work together.

Ecogrid, a project based in Denmark, made an attempt to unify several different houses into one smart system [12]. They claimed to control units in 800 homes in a connected system remotely. The project lasted between 2016 and 2019 and was more of a test concept to demonstrate how houses connected together in this smart grid could improve overall energy usage. Although they were able to integrate more green energy, reduce costs for customers, and maintain a balance between production and consumption, the project has been discontinued. This cessation of the Ecogrid project underscores the unmet need for comprehensive, interoperable, and user-friendly solutions in the smart-home sector.

Beyond heating control, this gap in the market extends to other aspects of smart homes, including ventilation systems and solar battery management. All-in-one solutions that address these issues exist, some of which are presented in Section 2.1.2. However, they often offer limited customization, and their control systems' overall impact can be difficult to discern. This again highlights the critical issue at hand: the absence of advanced, interoperable control systems that cater to varying user requirements without demanding excessive technical expertise.

To address this gap, future smart-home solutions could focus on the following aspects:

- **Interoperability**: Ensuring that smart home devices and systems can work seamlessly with each other, regardless of the manufacturer or ecosystem. This would enable users to mix and match devices to suit their specific needs and budgets.
- **Customization**: Developing advanced control systems that offer a higher degree of customization, allowing users to tailor their smart-home solutions to their unique requirements and preferences.
- **Usability**: Simplifying the process of setting up and maintaining advanced control algorithms, making it more accessible to users without extensive technical knowledge.
- **Transparency**: Providing users with clear insights into the performance of their smart-home systems, including quantifiable metrics that demonstrate the effectiveness of the control algorithms in use.

By adhering to these points, we can address the identified challenges and pave the way for the next generation of smart home solutions. This would allow home users to effortlessly access advanced control algorithms without investing in expensive ecosystems or proprietary smart hubs. This vision can be achieved by leveraging the open APIs of existing devices within their homes, thus granting them smart capabilities without any additional installations. As the necessary data and actuators are often already in place, unlocking these potential capabilities requires establishing the required infrastructure. This approach would empower individuals with control over their smart homes and simultaneously contribute valuable data for further research in the field.

1.2.3 Towards Economical and User-Friendly Smart Homes

The landscape of smart-home development presents a spectrum of solutions that vary significantly in sophistication and implementation costs, as discussed further in Section 2.1. The possibility of homeowners effortlessly installing readily available devices that may not inherently communicate with one another, but can be unified through non-local software, provides an economical and convenient avenue to decrease energy consumption. For instance, a homeowner could opt for a Sensibo Sky for heat pump control, a Tibber Pulse for monitoring electricity usage, and potentially a few Mill Sense sensors for assessing the indoor climate [13–15]. While these specific devices are just examples, their interconnectivity would supply all the necessary data for intelligent control, with the sole missing component being the system that unites them.

The system developed in this thesis aims to serve as that missing link, and its implementation is elaborated upon in Chapter 4. Despite the existence of similar solutions, the necessity for a solution that suits specific research requirements motivates the development of a new system. Additionally, this endeavour aligns with the UN sustainability goals, which are discussed further in Section 8.7.

1.3 Scope and Limitations

This project primarily focuses on developing the software infrastructure necessary for collecting data, which will serve as the foundation for more complex algorithms in smart-home control. As the research group *POWIOT* led by Sebastien Gros explores various aspects of smart home management, creating a scalable infrastructure for efficient data gathering is crucial for the team members.

The system is limited to devices that can be accessed remotely without any user intervention within the home. These devices must have a public API endpoint to which the system can connect. The currently supported devices are those in homes that have consented to participate in this research, as access to these devices is needed to test the integrations.
The main objectives of this project include:

- Developing a robust infrastructure for data collection and processing.
- Simultaneously gathering data from multiple houses.
- Facilitating easy access to data for research group members.
- Implementing a backend for serving collected data and providing information about active devices to users.
- Closing the control loop by running control algorithms on the server using the collected data.

1.4 Layout of this report

The report starts with an overview covering the current landscape of these solutions, what is available, and their limitations in Chapter 2. Then it will outline the software specification for the new system, going over its requirements in Chapter 3. After that, the entire implementation will be detailed in Chapter 4. To close the feedback loop, a test was conducted in Chapter 5, where control was used alongside the system. A complete system guide will be found in Chapter 6. After that, results are in Chapter 7, followed by the discussion in Chapter 8. Finally, the conclusion can be found in Chapter 9.

Chapter 2

Background

This section provides the necessary background information for this thesis. First, it explores the existing smart-home solutions in residential homes in Section 2.1. Then, in Section 2.2, it delves into the theory required for the implementation of the system in Chapter 4.

2.1 Current landscape

The landscape of IoT and smart homes is currently evolving at a rapid pace. With numerous vendors offering varying functionalities, navigating and implementing these technologies can be challenging and expensive. Furthermore, the degree of vendor lock-in varies significantly. Some manufacturers develop proprietary software that restricts device compatibility to their ecosystem, while others create open APIs for greater flexibility.

Historically, these systems have been associated with high costs and complex installation processes. However, the recent trend seems to have shifted towards more specialized, cloud-connected devices that excel in performing specific tasks. Some of these will be detailed in Section 2.1.1. This shift allows for greater adaptability and ease of integration within smart-home ecosystems.

2.1.1 Devices for smarter energy management

There is a wide range of energy-related IoT devices which fulfil various different tasks. Below is a list showing an assortment of IoT energy-related home devices:

- **Sensibo Sky and Air devices** enable Wi-Fi connectivity for most air-to-air heat pumps, allowing remote control through an app and their API [13]. These devices function by imitating the heat pump's remote, transmitting infrared signals in the same manner. They are equipped with built-in humidity and temperature sensors.
- **HAN meters** are mandatory in all Norwegian homes [16]. Connecting a device like Tibber Pulse [14] to this port enables the collection of real-time

energy consumption data, which can be used to optimize usage. Hourly data is accessible regardless of installed devices, but the ease of access varies depending on the electricity vendor.

- **Mill Heaters** are Wi-Fi enabled and integrate within their ecosystem for seamless automation based on sensor triggers [17]. Mill offers additional sensors like the Mill Sense that can be linked to heaters for improved temperature control in a room [15]. These sensors include humidity, temperature, eCO2 and Total Volatile Organic Compounds (TVOC) readings.
- **Aquarea Smart Cloud CZ-TAW1** facilitates remote control of a select few water-to-air heat pumps [18]. Once connected to the system, it is accessible through their smart cloud interface.
- **Easee Home** is a series of Electric Vehicle (EV) charging stations designed for residential use [19]. They are easy to install, compatible with various home environments, and Wi-Fi enabled for remote monitoring and control via a mobile app.
- **Nest Learning Thermostat** is a smart thermostat that supports Google Home and Apple Homekit [20]. It enables both manual control and independent operation based on learned user patterns. These thermostats are compatible with various heat pumps and provide intelligent temperature control through sensors located elsewhere in the home.
- **Tado Smart Thermostat** is a Wi-Fi-enabled device offering intelligent climate control for heating and cooling systems [21]. Compatible with various heat pumps and water boilers, Tado learns user habits and employs geofencing technology to optimize energy usage. Additionally, it integrates weather forecasts to adapt energy consumption to external conditions.
- **Similar solutions** often adopt cloud-based platforms with publicly available APIs, simplifying access and integration for users.

2.1.2 Smart-home solutions

To unify the variety of smart-home devices into a single solution, several vendors offer all-in-one systems that consolidate these devices into one app or ecosystem. Below is a list of some of the most popular smart-home solutions, though it is not exhaustive. They are listed in no particular order:

- **Homey** is a smart hub designed to support a wide range of vendors [22]. It is a centralized solution to manage and control smart devices, though it is not primarily geared towards integrating different devices. Homey provides flow automation for basic IF This Then That (IFTTT) automation that works even without an internet connection. Homey also offers an all-cloud solution that doesn't require a smart hub but integrates with other cloud-based systems.
- **Futurehome** provides a complete smart-home system. They sell their own hub along with separate sensors and devices for a smart home [23]. Although its compatibility list is shorter than Homey, the supported devices

work natively. Futurehome claims to reduce energy bills by up to 25% with its integrated solution that measures power and controls heating, boilers, and car charging based on spot market rates. To install the system, an electrician is needed.

- **Home Assistant** is the tinkerers choice. It is an open-source smart hub software focusing primarily on local control [24]. With it being open source and easily available they have a huge supported device list, it is only the software, and thus it has to be run on some kind of local hardware. For most, this could be a home server or some smaller, more dedicated unit like a Raspberry Pi. However, this can only communicate with other cloud-based devices as it needs additional hardware to connect to and control through Zigbee and other protocols commonly used for low-power smart home devices. It is also not out of the box possible to control these devices remotely, but it can be done using custom integrations.
  The pros of Home Assistant come in its total control. As it is made for the tinkerers, they can easily create their own integrations and automation, and there are really no limits on what can be done, although it may take time and some know-how. More complicated automation like Model Predictive Control (MPC) is not easily implemented, but workarounds are possible.
- **Google Home and Apple Home** have their own ecosystems for controlling and seeing data from various IoT devices [25, 26]. Their supported list is, however, smaller than Home Assistant as they need native compatibility. Automations are also limited in functionality. These don't usually require a central hub, although having one adds more features, such as remote control.
- **Tibber** functions as an electricity provider [27]. Besides selling proprietary devices like the Tibber Pulse, which enables live power monitoring, they also offer a dedicated app. This app integrates with a range of other devices to facilitate energy-saving automation by intelligently monitoring electricity usage in relation to spot market trends [14].
- **Enode** is a startup from NTNU that doesn't provide the smart features necessarily, but they allow for the unification of several different APIs [28]. Be it different Heating, Ventilation, and Air Conditioning (HVAC) systems, EV car chargers, solar inverters etc. This system maps API calls through their system to make it seem like the calls to different providers are the same. In this way as the user of their API, you wouldn't need to know the difference between a Sensibo API or a Mitsubishi cloud API, as they would do the work of integrating this for you. This system is mostly meant as a baseline for other integrators to work on top of, as they take care of the grunt work of having these APIs work and up to date. However, if their system is down, your entire system will also be down. In addition, if a vendor updates their API you would need to wait for Enode to update their integration before it will work again.
- **Samsung SmartThings** is a smart home ecosystem that combines various

devices, allowing for centralized control and management [29]. With a wide range of compatible devices and an open platform for developers, SmartThings offers users flexibility in configuring their smart home. The system supports automation through the SmartThings app, enabling the creation of custom rules and routines for different devices to interact.
- **Various vendor apps**: Many different device makers today also create their own apps that can sometimes integrate with other vendors' devices. These vary greatly in functionality.
- **Custom software**: Since a lot of vendors today allow for open APIs to their devices or open local protocols like MQTT, creating custom software is also a possibility. This can be done by connecting to these various APIs, logging data or trying to control them through code.
- **Matter standard** aims to unify all IoT devices by creating a similar interface between them all [30]. This standard will hopefully make it much easier to connect the devices of different vendors together. It is open source and already pushed by big corporations such as Apple and Google. Although it will take time, this will most likely change the IoT landscape drastically.

2.1.3 The Role of Data and Advanced Controls in Smart-Home Systems

The effectiveness and efficiency of smart-home systems hinge significantly on data. High-quality sensor data enables appliances to make informed decisions based on various factors like temperature and occupancy, leading to enhanced performance and greater energy efficiency. A classic example is a heat pump outfitted with several temperature sensors, which can develop a nuanced understanding of the indoor climate and construct more accurate operational models. Through the intelligent utilization of this data, the pump and other devices can anticipate future conditions, such as changing weather patterns or fluctuations in energy spot prices.

Commercial off-the-shelf solutions often fall short as they limit control to built-in sensors, inhibiting scalability and overall effectiveness. Addressing this challenge calls for a more sophisticated solution, which will be explored further in Chapter 4.

Given their substantial energy usage, heating, ventilation, and battery management systems, when present, are key optimization targets within the myriad systems of a smart home. Although these systems often come with implemented control algorithms, there is room for further enhancement. For instance, while ventilation systems may use a Proportional-Integral-Derivative (PID) control approach, recent testing suggests that model predictive control systems can lead to more efficient energy use and better system control [31].

Model Predictive Control (MPC) is an advanced control technique that enhances the performance of dynamic systems through the use of a mathematical model to predict future behaviour and optimize control inputs based on desired performance criteria. This technique is widely applied in various fields, including robotics [32], process control, [33], and smart homes [34], for its ability to optimize control inputs based on future predictions.

In previous work, heat pumps were equipped with MPC combined with a Moving Horizon Estimation (MHE) approach for state estimation [1]. This combination led to improved efficiency and performance by accurately predicting future system behaviour based on its current state. These findings underline the potential for advanced control techniques to revolutionize smart-home systems, provided they're fueled by comprehensive and high-quality data.

As the complexity of control models increases, with strategies ranging from machine learning [35], to MPC, to reinforcement learning-enhanced MPC models [36], the importance of data becomes even more critical. Accurate models require more than a detailed floor plan; they need real-world data, capturing factors like insulation quality and occupants' usage patterns. Collecting such comprehensive data empowers superior control and opens avenues for research with a diverse set of houses. This data-driven approach would also facilitate the creation of digital twins, enabling accurate simulations to optimize smart home systems further.

2.2 Software and theory

This section introduces software and frameworks used for the implementation of a custom logging system in Chapter 4.

2.2.1 Docker

Docker is a widely-used open-source platform that has revolutionized the way software applications are developed, deployed, and managed [37]. Docker allows developers to package applications and their dependencies into self-contained, portable containers that can be easily deployed and run on any platform. By abstracting the underlying infrastructure, Docker enables applications to run consistently and reliably across different environments, including on-premise data centres, public and private clouds, and hybrid environments. Docker has become a critical tool in modern software development, enabling faster and more efficient application delivery and deployment, and helping organizations to achieve greater agility, scalability, and flexibility in their operations. As seen in Figure 2.1, Docker works by having its own Docker layer and running containers on top of it. This allows the different containers to share much of the same infrastructure, reducing overhead, storage and memory use. This is in contrast to normal virtual machines that come with everything necessary for running a complete operating system every time one is booted up. Docker containers are built using Dockerfiles, which contain information about what image they are based on and what packages will be installed in the container. This Dockerfile also contains information about what to load into the container and in what order.

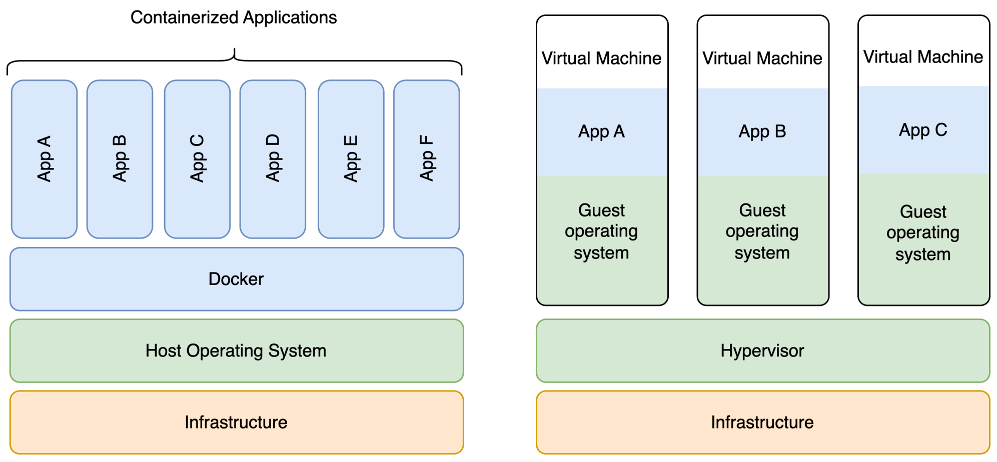


Figure 2.1: Overview of how docker works, Docker on the left and virtual machines on the right. The figure is based on the Docker documentation [38].

Docker Hub hosts many different images that can be used as the base for Docker containers [39]. These can be everything from small containers containing only the bare minimum to a full-fledged Ubuntu operating system. This makes it easier to create small microservice applications.

Docker Compose

Docker allows for quite a complex network creation [40]. To compose several communicating containers, separate virtual networks within the docker container clusters can be set up. To compose several containers at once that communicate, one can use Docker Compose [41]. It is a tool that simplifies the creation of different containers into a single .yaml file. An example of starting a Docker container using Docker Compose can be seen below.

```
version: "3.9"
services:
  postgres_db:
    image: postgres
    command: -p 5432
    expose:
      - 5432
    ports:
      - "5432:5432"
    volumes:
      - postgres_data:/var/lib/postgresql/data/
    restart: always

volumes:
  postgres_data:
```

By storing the above file in a*docker-compose.yaml* file and running:

```
  $ docker compose up
```

A Postgres database with port 5432 exposed will be started. The local folder postgres_data will also hold the local database.

2.2.2 Kubernetes

Kubernetes, colloquially referred to as K8s, is a sophisticated and open-source platform built specifically for managing containerized applications [42]. Such applications often come in the form of Docker containers. The platform automates the deployment, scaling, and management of these applications, thereby freeing developers from the concerns of underlying infrastructure complexities. Its robust functionality and versatility make it a popular choice for environments on-premise and in the cloud. It aids in the modern shift towards containerization and microservices architectures, providing an optimal platform for deploying and managing cloud-native applications. With potent tools, APIs, and built-in capabilities, Kubernetes enables load balancing, service discovery, and automated failover.

Kubernetes is a flexible platform with an array of features and extensibility options. Developers can tailor and extend it to meet their specific needs, adding to its appeal [42]. One significant feature includes the secure storage and utilization of confidential credentials, often termed *secrets* [43]. By default, Kubernetes encodes these secrets and offers additional encryption at rest for bolstered security. These secrets are seamlessly integrated into pods as environment variables, simplifying their usage within the containers.

Architecture and Components

The architecture of Kubernetes is a meshwork of several components, illustrated in Figure 2.2. At its core are clusters, facilitating communication across different machines [44]. A single machine can operate as an individual cluster or be part of a multi-machine cluster. Each cluster comprises nodes, which serve as the running environment for containerized applications, i.e., Docker containers. The control plane, a vital part of the Kubernetes architecture, oversees the operation of worker nodes and pods within the cluster. Pods represent the smallest deployable units of computing within Kubernetes. They host containerized applications and can scale to run multiple instances of the same application simultaneously. Deployments, defined within nodes, consist of a specified number of pods. Kubernetes has an inherent capability for auto-scaling these deployments to meet fluctuations in demand, allowing for an elastic and responsive system.

Kubernetes provides clients for multiple programming languages, including Python. The Python client, for instance, interfaces with the control plane, enabling actions on the cluster [45]. It supports various operations, from querying running pods, and initiating new deployments, to deleting existing ones. The comprehensive API ensures all actions on the clusters can be seamlessly executed, reinforcing Kubernetes as a robust tool for container orchestration.

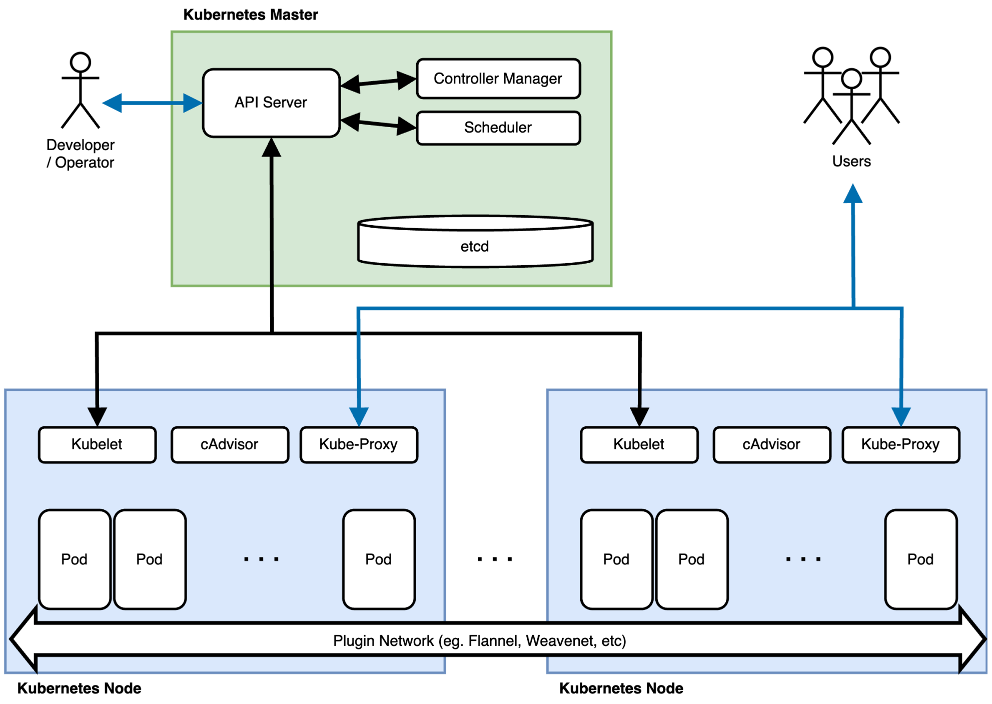


Figure 2.2: Overview of different Kubernetes components. The figure is based on the Kubernetes documentation [44].

2.2.3 MQTT

MQTT (Message Queuing Telemetry Transport) is a messaging system that is well-suited for constrained devices or networks with limited bandwidth [4]. MQTT operates on a publish/subscribe model, where all connected nodes can publish and subscribe to various topics. To enable this communication model, a broker is required to act as a post office, receiving and redistributing messages to the appropriate subscribing nodes. The broker can be hosted locally or in the cloud, and various brokers offer different service qualities, such as node reconnection and encryption. MQTT also provide node authentication to ensure that only eligible nodes can connect to the broker. While MQTT does not have built-in encryption, it does support encryption over the Secure Sockets Layer (SSL) protocol if configured on both ends of the communication [46].

2.2.4 Django

Django is a popular open-source web framework written in Python that is designed for the rapid development of robust, scalable, and maintainable web applications [47]. Django follows the Model-View-Controller (MVC) architectural pattern, which promotes code organization and separation of concerns. Django offers a rich set of built-in features and tools, including an ORM system, a powerful templating engine, an administrative interface for managing application data, and support for user authentication, session management, and other common web development tasks. Django also provides a robust ecosystem of third-party packages and libraries that extend its functionality and enable developers to build complex applications with ease [48]. With its emphasis on simplicity, flexibility, and reusability, Django has become a popular choice among developers for building web applications, from small personal projects to large-scale enterprise applications.

Django rest framework

The Django Rest Framework (DRF) is a powerful and flexible toolkit for building Web APIs [49]. It is an extension of the Django web framework, providing a set of additional tools and libraries specifically for building RESTful APIs. RESTful APIs, or Representational State Transfer APIs, are a type of web-based interface that follows certain principles to enable communication and data exchange between different systems or applications. In simpler terms, RESTful APIs provide a standardized way for systems to talk to each other over the Internet. They use common HTTP methods, such as GET, POST, PUT, and DELETE, to perform actions on resources (such as retrieving, creating, updating, or deleting data) in a predictable and consistent manner. DRF includes a range of features and capabilities, including support for serialization and deserialization of complex data types, authentication and permissions, content negotiation, pagination, filtering, and more. DRF also provides built-in support for popular serialization formats, such as JSON and XML, and enables developers to easily customize the output of their APIs. With DRF, developers can rapidly build robust and scalable APIs that can be easily consumed by clients across a variety of platforms and programming languages. DRF has become a popular choice among developers for building APIs that power web and mobile applications, IoT devices, and other systems that require seamless data exchange over the web.

2.2.5 InfluxDB

InfluxDB has emerged as a powerful and efficient time series database designed for high throughput and high demand [50]. It is specifically designed for time series data and has thus been used mostly in IoT data retrieval, industrial automation and monitoring systems. Its most highly viewed features are its data compression, scalability, and data retention policies, which give it quite the edge in this market. It can be used through their influx cloud, or it can also be self-hosted on company premises.

In conjunction with InfluxDB, Telegraf, another product developed by InfluxData, provides an extensive plugin system that complements the robustness of InfluxDB [51]. Telegraf is a server agent designed for collecting and reporting metrics and data, with more than 200 plugins available to gather various types of metrics from a myriad of sources. These plugins can be categorized into four types: input, output, aggregator, and processor plugins. Input plugins fetch metrics from specified sources, and output plugins are used to send these metrics to various other data stores, including InfluxDB. Aggregator plugins create aggregate metrics (like sums, averages, minimums, etc.), while processor plugins transform, decorate, and/or filter metrics. This extensibility allows users to integrate InfluxDB and Telegraf into existing systems or tailor it to their needs. Telegraf's plugin-centric design thus enhances InfluxDB's versatility, making it an even more powerful tool for time series data management.

Flux query language

The database can be queried using InfluxQL, an SQL-like language. However, this is not native in the latest version. Instead, the flux query language has emerged with its own Python integration [52]. The flux language resembles a functional programming style where data is piped into each other, and the server can handle much of the heavy lifting. Inside this language is the support for various statistical tools that can make the server do the work, making the amount of data needed to be transferred to the client smaller. For examples of what this query can look like, head to Section 6.3.

2.2.6 PostgreSQL

PostgreSQL is an advanced open-source relational database management system [53]. It adheres to the SQL standards and has support for more advanced types. In addition to this, it supports high workloads and a lot of customisation if necessary. It integrates well with Django using its ORM.

Chapter 3

Software Specification

This chapter details an overview of what the system wants to achieve and what parts it is built up from. First, the overview will be shown in Section 3.1. Then the requirements for the various parts of the system will be presented in Section 3.2, Section 3.3 and Section 3.4.

3.1 System overview

A simple illustration of the entire system can be observed in Figure 3.1. This system is predicated on the capability to log and store information from numerous devices within a residence in a designated database. Subsequently, the collected data can be utilized in a processing node for various purposes, ranging from basic control measures to more sophisticated Model Predictive Control (MPC) schemes, contingent on the specific device being managed. The control is implemented based on multiple data points recorded for each residence within the database. Commands are then transmitted to the designated devices, completing the feedback loop. To facilitate this process, a backend infrastructure is required to manage user information, house details, and the devices currently used by residents.

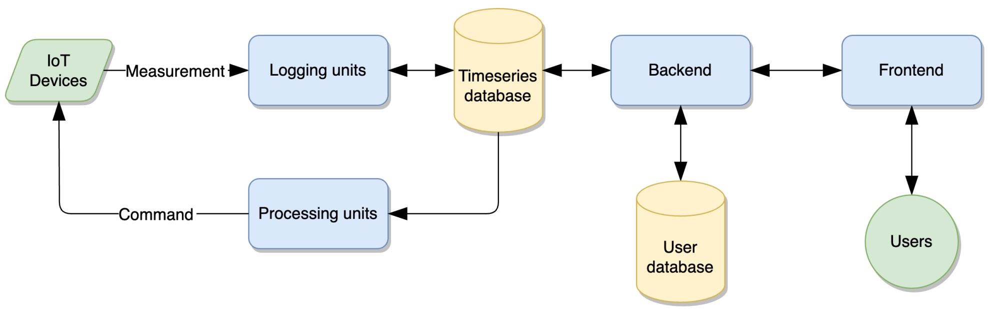


Figure 3.1: Simple overview of the proposed system.

Based on this overview, the system comprises a few main systems. And this chapter will outline a few requirements for the software. These requirements aren't necessarily every feature that is needed for the system to work, but all the features that would make this a complete product. The different systems are the following:

- Data logging
- Backend
- Processing
- Frontend

This project encompasses every aspect of the system, with the exception of the frontend. Therefore, its requirements will not be a part of this thesis.

Given the natural absence of standardized requirements for IoT logging systems, an effort has been made to establish a set of general requirements. Some requirements might be missing, but if it adheres to these, it should at least be capable of doing the minimum required by this type of system.

3.2 Data logging

The data logging module is an essential component of the system, as it enables the collection and storage of data from connected devices. The rest of the system can then distribute and use this data.

Data Logging Requirements

- **DLR-1:** High available resolution of measurements.
- **DLR-2:** High throughput.
- **DLR-3:** Resilient against system crashes and restarts.
  - **DLR-3.1:** Automatic restart on crashes.
  - **DLR-3.2:** Credentials saved through crash/restart.
  - **DLR-3.3:** Reconnection after internet failure.
- **DLR-4:** Persistent data storage for long-term access and analysis.
- **DLR-5:** Automatic backup and recovery in case of data loss.
- **DLR-6:** Scalable architecture for handling large amounts of data and users.
- **DLR-7:** Security and Compliance.
  - **DLR-7.1:**Compliance with relevant data privacy regulations (e.g. GDPR).
  - **DLR-7.2:** Robust data encryption and protection against unauthorized access.

3.3 Backend

- **BR-1:** Integration with other systems
  - **BR-1.1:** API access for developers to build custom applications and services on top of ours.

  - **BR-1.2:** Standard REST API to support multiple data formats for easy and seamless data exchange.
- **BR-2:** New device integrations
  - **BR-2.1:** Ensure the backend can work independently of new integrations.
  - **BR-2.2:** Have a template for creating new integrations to streamline the process.
  - **BR-2.3:** Provide support for custom credential inputs, so any device can be integrated.
- **BR-3:** User features
  - **BR-3.1:** Enable users to start devices based on what they have at home.
  - **BR-3.2:** Enable users to check the status of their devices.
  - **BR-3.3:** Enable users to stop devices on demand.
  - **BR-3.4:** Enable users to restart their devices on demand.
  - **BR-3.5:** Enable users to retrieve their own data.
  - **BR-3.6:** Remind users if their device is malfunctioning.
  - **BR-3.7:** Provide different tiers of use (just logging or logging and control) with configurable parameters for each tier.
- **BR-4:** Security
  - **BR-4.1:** Encrypt secret data.
  - **BR-4.2:** Enable user authentication.

3.4 Processing

- **PR-1:** Reliability
  - **PR-1.1:** Resilient against network connectivity issues.
  - **PR-1.2:** Can send commands to the various actuators.
- **PR-2:**Configuration
  - **PR-2.1:** Optional customisation of control algorithms.
  - **PR-2.2:** Variety of supported devices for control.

Chapter 4

Implementation

In this chapter, we delve into the software's implementation. We begin with Section 4.1, providing a basic overview of the system. Following this, Section 4.2 presents insights into our choices of software and the subsequent creations. Finally, in Section 4.3, we enumerate the devices that have been implemented within the system.

4.1 Overview of the system

The system comprises a main server, which talks to different vendors APIs to log data and control devices. Data is stored in a local database instance which can be queried locally and remotely through the internet for development purposes.

A backend API is available for normal users to log in, select which devices they currently have in their home, send in their credentials, and immediately start up the respecting logging nodes for their home devices. To start logging, they would only need one of the devices supported, all of which are listed in Section 4.4. Some of these devices do not even need a physical unit in-house, for instance, the weather forecast logger.

Based on this data, a control loop can be made, e.g. retrieving temperature data from a room and sending actuations to a heat pump. This is possible as some of the integrated devices also allow for control to be sent back through their APIs. This could be heat pumps, ventilation systems or a battery inverter. With data about power consumption and the spot marked, the power consumption can be optimized. This processing (control) node can also be run directly on the same server or remotely from an external computing unit when debugging.

4.2 Modules

A comprehensive overview of the entire system is illustrated in Figure 4.1. The system comprises multiple modules, all managed by the Kubernetes framework. The different modules are all booted up as deployments within the Kubernetes

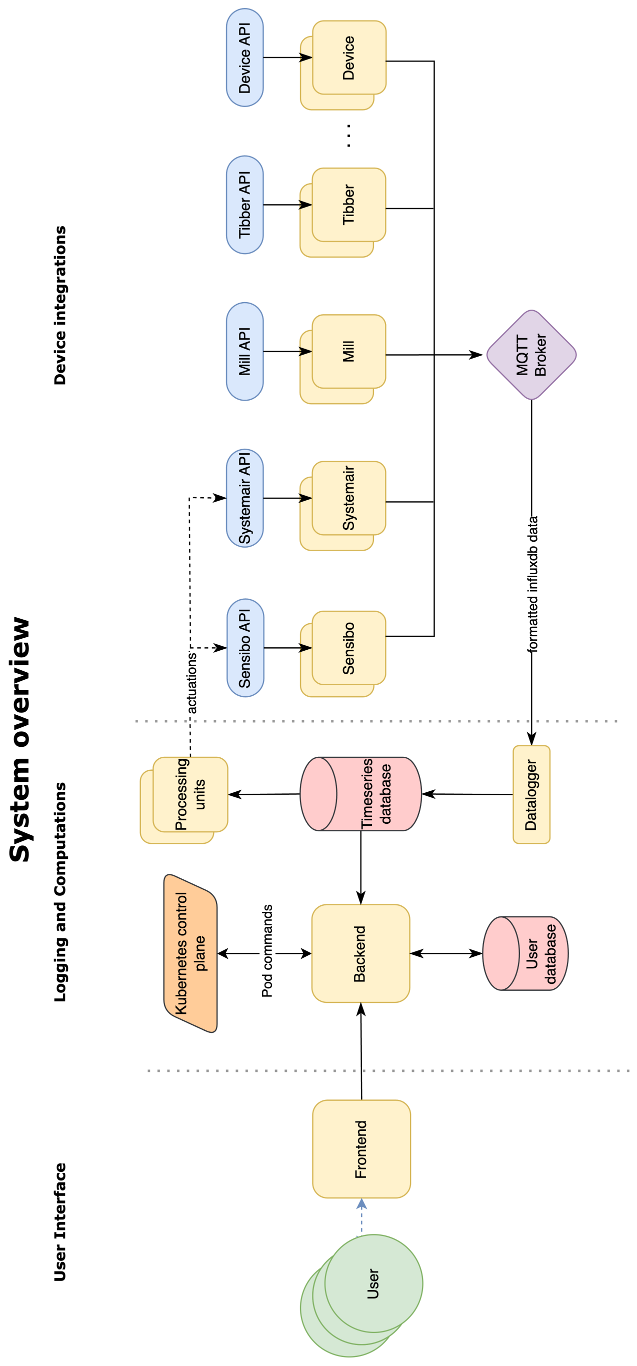


Figure 4.1: The system's architecture comprises yellow rectangles and pink cylinders, all of which are pods within Kubernetes and are run as Docker containers. Cylinders are storage instances. In instances where there are overlapping squares, multiple can boot up simultaneously. A few device integrations have been picked out as examples, but more are available, and more actuations can also be set up from the processing units. Currently, Sensibo and Systemair are among the APIs with actuation as well.

cluster. Local inter-container communication between device integrations and the Datalogger is facilitated through MQTT for seamless interaction between different programs. Devices can therefore be developed in any language with an MQTT interface and run within their respective containers. Data generated by these devices are sent to the Datalogger node that filters and validates the data before forwarding it to the InfluxDB database. Users can query the data directly from the database with developer access or interact with the Django-based backend and PostgreSQL user database to manage devices for their homes. Endpoints exist to retrieve data, and a comprehensive list of all of them can be found in Appendix B.

4.2.1 InfluxDB Time Series Database

The InfluxDB time series database is utilized for data storage and runs within the Kubernetes cluster. A dashboard for the database can be accessed at `https://influx.powiot.no`. Developers can query the database using custom accounts through the online dashboard interface or programmatically with API keys. A Datalogger node works alongside the database to collect and validate data sent from other devices via MQTT before storing it in the database. An overview of the structure of the InfluxDB database can be seen in Figure 4.2. The illustration depicts the method of data storage within the database, which comprises distinct components termed as *buckets*. These *buckets* essentially represent the databases themselves. Inside each bucket, we store the *measurements*. Each *measurement* has a set of associated *fields* and *tags*, as well as a *timestamp* that marks the moment when the *measurement* was taken. To better understand this concept, consider the Mill Sense *airSensor* on the right of the diagram. It demonstrates the different *fields* and *tags* attached to it. Within the system, the *Mill Sense* is within the *airSensor* group. Crucially, all the fields, tags, and timestamps can be described as key-value pairs. In this construct, every key (field, tag, or timestamp) has a corresponding value associated with it. This structure ensures that each piece of data can be identified and accessed easily within the database. For all measurements in the system, the *house_id* and *device_name* tag has to be included. Additional information can, for instance, be the location within the house. The available dashboard should be for administrative usage, as this has complete access to all houses. For normal user access to devices, the backend API should be used.

The database itself is based on the latest InfluxDB image 2.6, and it is booted up within Kubernetes using a yaml deployment file, with secret data put in through the yaml file. More info about this procedure can be found in Section 6.10.

4.2.2 Datalogger

The Datalogger node is responsible for pushing data into the database, its placement in the system can be seen in Figure 4.1. It validates the data received over MQTT for correct formatting before injecting it into the InfluxDB database. In addition, the Datalogger serves as a monitor, tracking which nodes are transmitting

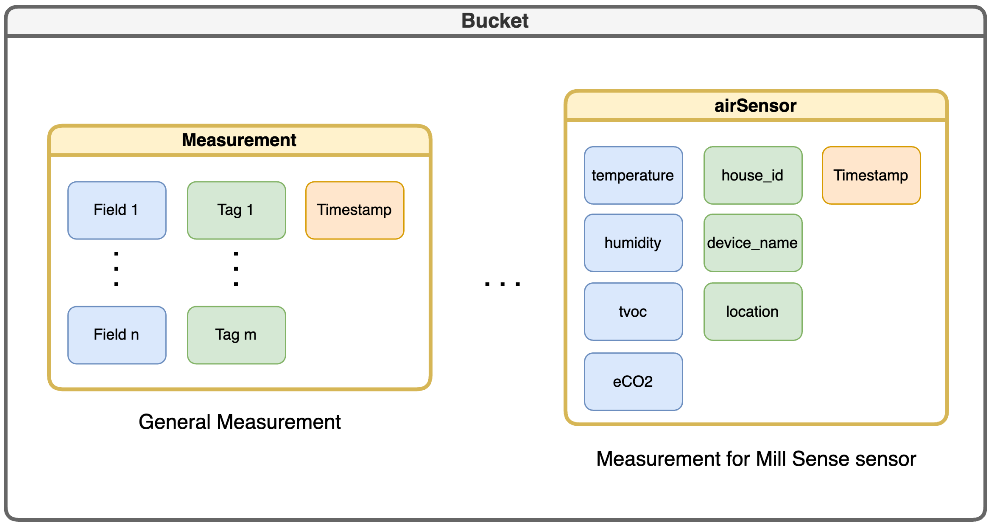


Figure 4.2: *Buckets* in InfluxDB are the names of the databases themselves. One bucket is created in this system, where the measurements for all devices within all the houses reside. This bucket has been named *HouseData*. *Measurements* have a certain amount of *fields* and *tags*, and also a *timestamp* of when the measurement was done. All of these *fields*, *tags* and *timestamps* are a key-value pair, and thus all have a corresponding value to them.

data and at what time. It also logs useful information related to data transmission, which proves invaluable when identifying issues with data sent during the development of new integrations. Although this introduces an additional layer of interaction, the enhanced clarity and support it provides to developers during the creation of integrations provides a justification for its inclusion. While it is possible to feed data directly into the InfluxDB database using Telegraf plugins, this approach requires pre-formatting of the data, complicating the troubleshooting process when developing new device integrations.

4.2.3 Backend

A Django-based backend, accompanied by a PostgreSQL database, is available to manage user credentials, devices, and houses associated with individual users. The overview of the models within the backend can be seen in Figure 4.3. Users can interact with endpoints for creating, restarting, or deleting devices and obtaining device status information. The implementation leverages the Django REST framework to facilitate easy endpoint implementation and automatically generate a browsable web API for convenient testing [49]. This backend stores info about the various integrated devices, and new integrations can be added through the admin panel.

Another part of the backend is the integration with the Kubernetes cluster. A custom interface has been made using the Kubernetes API for Python, which allows for checking the status of the different devices. This allows users to see the

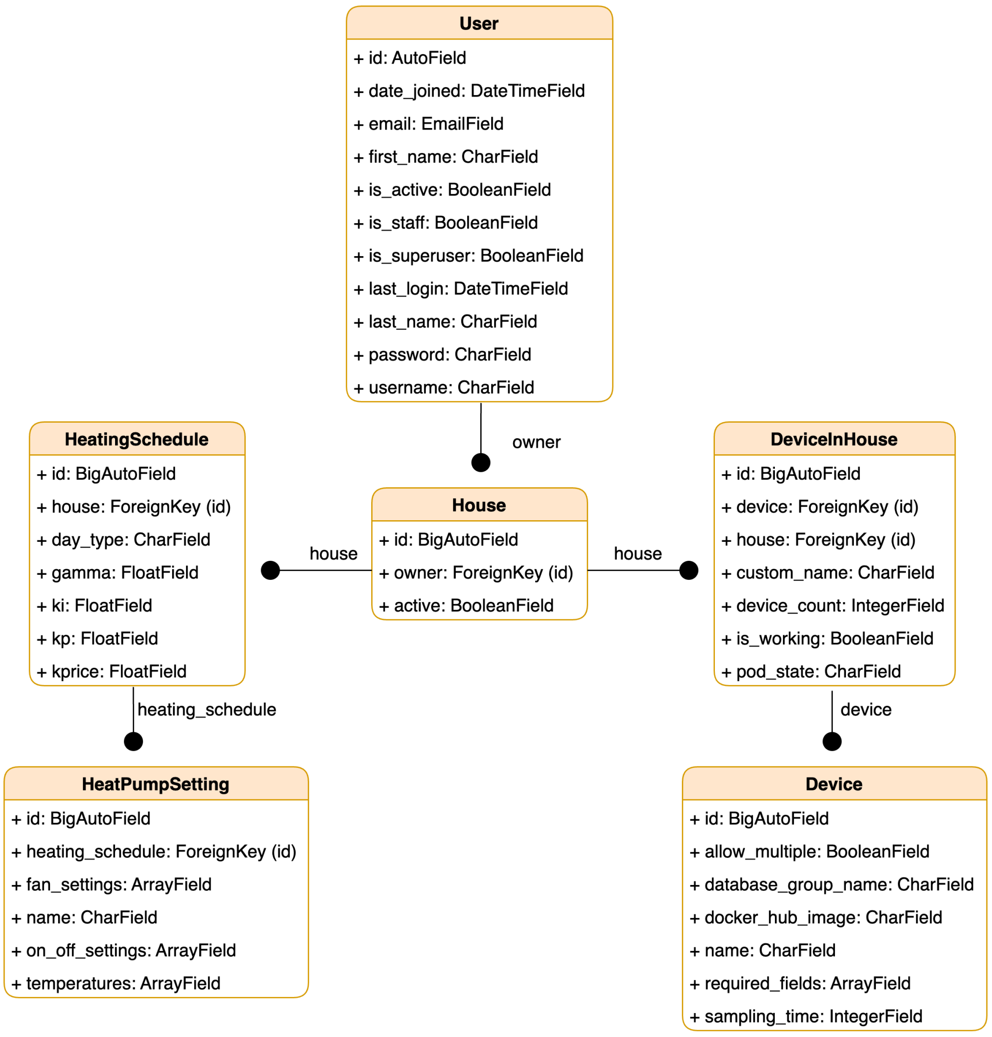


Figure 4.3: A table overview of the different models in the Django backend. It shows the types of the data stored in the PostgreSQL database.

reason why their integrations might be crashing, be it due to an API being down, or if invalid credentials were put in.

The Django backend is developed locally and is then built into a Docker container using the associated Dockerfile, and is then run within the Kubernetes cluster.

4.2.4 Backend API

The current API documentation is dynamically available at `https://powiot.no/api/docs`, where an example of this can be seen in Figure 4.4. Complete documentation with request body examples can be found in Appendix B.

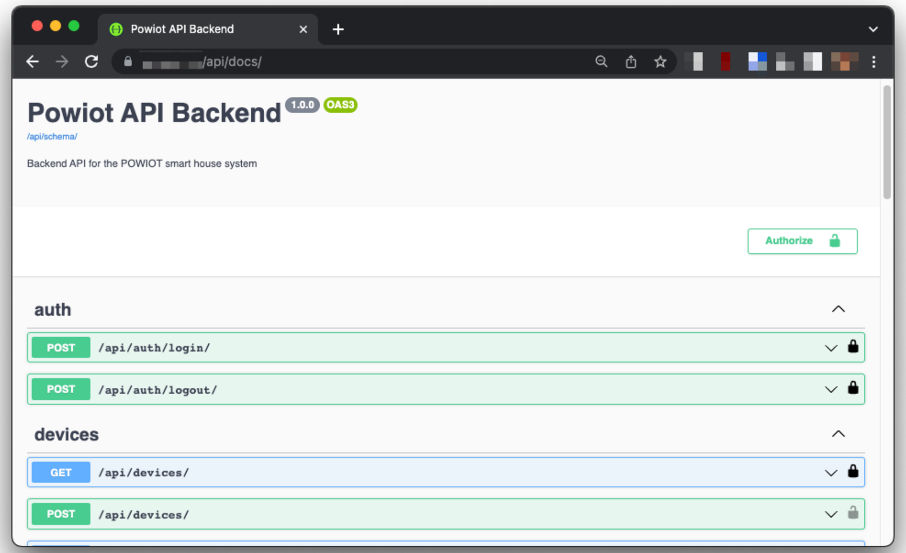


Figure 4.4: Swagger documentation available at /api/docs.

4.2.5 Device Integrations

Device integrations are developed using Docker containers, which can be written in any language with an MQTT interface. Currently, all existing integrations are developed in Python using the Paho-MQTT library [54] for MQTT communication. A template has been created to expedite the development of new devices. The program flow of one of these device integrations can be seen in Figure 4.5. This template includes the *Dockerfile* and an example Python device that sends dummy data. This template works out of the box, and can then be modified to suit the new device needs. For info about how to do this look in the user guide in Section 6.8.2. Device credentials are input through environment variables, which must be defined in the code and match those input through the backend API when starting a new device. The flow of this can be seen in Figure 4.6.

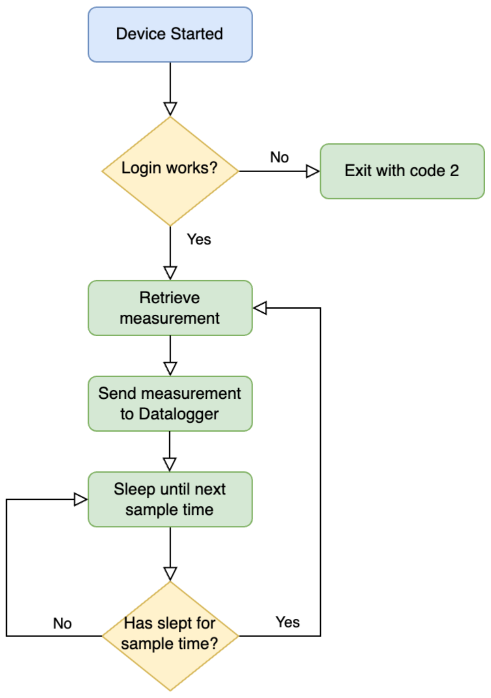

Device Started
Login works?
No
Exit with code 2
Yes
Retrieve measurement
Send measurement to Datalogger
Sleep until next sample time
Has slept for sample time?
No
Yes

Figure 4.5: The program flow for a device integration. The exit code for when a login does not work is to let Kubernetes know the cause of the exit.

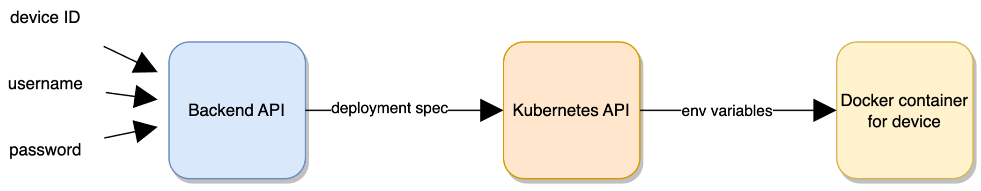


Figure 4.6: How environment variables are set into the respective docker containers all the way from the backend. A docker container has certain environment variables needed. This is then added to the device when creating it in the backend. Therefore when starting one of these devices, these fields are required, and through the Kubernetes API, a device is started with an accompanying secret with the environment variable data.

Devices can be tested independently before being incorporated into the system. All device integrations are pushed to Docker Hub and can be initiated from the backend immediately after being added to the list of devices. This ensures that the development of new devices remains independent of the running system. As long as the device functions correctly, it seamlessly integrates into the system without affecting other components. To organise device integrations, a grouping system has been created. All devices fall into some group, and if one does not already exist, a new one has to be made. The database groups try to generalize the data that the device retrieves. The database group name maps to the measurement name in the InfluxDB database. The current list of all database groups can be seen in Table 4.1.

Table 4.1: InfluxDB database name groupings.

Measurement	Description
airSensor	Temperature, humidity, and other climate sensor data.
powerConsumption	Power consumption for a house, realtime/hourly.
ventilationControl	Control of ventilation system.
heatPumpController	Control of heat pump.
weatherForecast	Weather forecasting data for location.
solarRadiation	Solar radiation forecasting for location.
solarBatterySystem	An integrated solar and battery inverter system.

This grouping is to maximise the interoperability of the different devices so that the same algorithms can use data from different devices without having to know which device it comes from. *airSensor* for example does not care if it is retrieved from a Mill Sense sensor or a Verisure smoke detector (which also has temperature and humidity). This list does not include every type of smart-home IoT device that can be included and will have to be extended with an appropriate name when a new type of device comes up.

4.2.6 Communication between components

To ensure interoperability, communication between device integrations and the data logger is facilitated through MQTT and its topics. Devices communicate externally with their respective vendor APIs using HTTPS calls.

Device data is transmitted via MQTT in the form of dictionaries, which are directly insertable into the InfluxDB database. This data is then encapsulated within a JSON object and sent over MQTT. Communication is unidirectional: devices transmit data to the Datalogger, which subsequently stores it in the database. This stored data can be accessed by other processing nodes or directly by users for further analysis or processing.

Sensor data is transmitted over the "sensors/<SENSOR-NAME>" topic, enabling the logger to collect all data published under the "sensors/#" topic and accommodate various sensor types. For instance, a Sensibo device would publish data on the "sensors/sensibo" topic. This approach is employed for local logging purposes to track data sources. Below is an example of the data structure sent over MQTT here as a Python dictionary.

```
data = {
    "measurement": "airSensor",
    "tags": {
        "house_id": "3",
        "location": "Soverom",
        "sensor_name": "mill-sense"},
    "fields": {"temperature": 19.7, "humidity": 40.3},
    "time": datetime.datetime.now().isoformat(),
}
```

The HTTP calls vary for each device, as they depend on the specific vendor. A comprehensive list of implemented devices can be found in Section 4.4.

Incoming communication to the pod is managed through a Cloudflare proxy. The configured domain is connected to Cloudflare, which in turn communicates with a pod running within the Kubernetes cluster. This is to open up the server to the Internet without port forwarding as this is not possible from the hosted NTNU server. As illustrated in Figure 4.7, this Cloudflare pod communicates with the Django backend service to access the system's backend. Subsequently, the backend interacts with various other services to access different modules within the cluster, such as the time series and user databases.

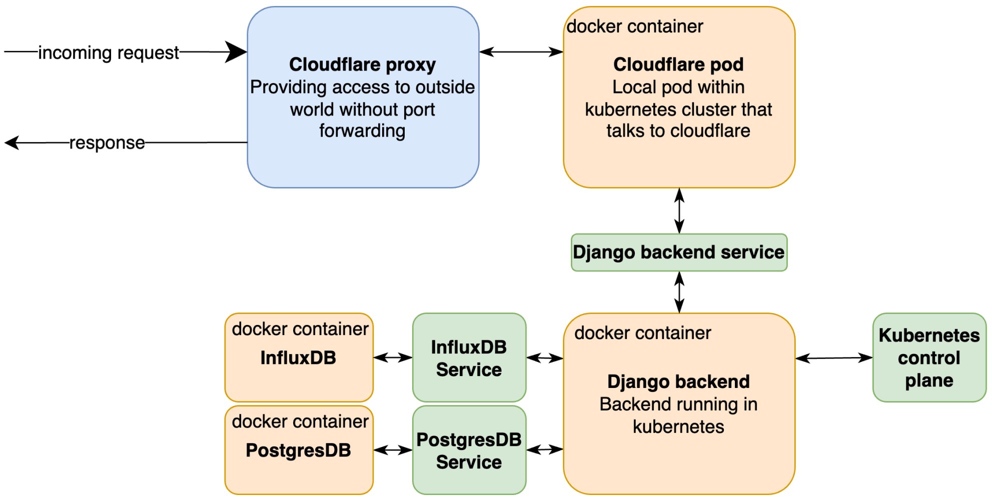


Figure 4.7: How a request is handled on the pod level inside the Kubernetes cluster. All orange squares are pods running within the Kubernetes cluster. The green squares are Kubernetes services, which act as the ports of the pods, allowing them to communicate with each other. The blue square is the Cloudflare proxy, which is hosted by Cloudflare.

4.3 Software design

The development decisions for the software, outlined in Section 4.1, were heavily influenced by a variety of goals. First and foremost, the software should be user-friendly and designed in such a way that future developers can easily pick up the mantle as the research project progresses. Another key consideration is scalability, anticipating the possibility of a significant increase in homeowners wishing to join the system. The software should also closely align with the requirements described in Chapter 3. Moreover, minimal maintenance requirements were prioritized, recognizing the limited resources available for continuous system upkeep. These overarching goals shaped the decision-making process, as discussed in the following section.

4.3.1 Software library: choices and considerations

Backend Selection

In the process of selecting a backend for this project, several systems were evaluated, including Django [55], FastAPI [56], and Flask [57], which are all major solutions for Python backend development. The predecessor of this system employed Flask as its web framework [1].

Flask offers ease of setup and initial use, but its expandability and database integration capabilities are limited [57]. Both FastAPI and Flask require developers to manually configure interactions with the database, increasing the risk of errors and necessitating further decisions about the database to use. However, the advantage is that only the necessary features need to be implemented, reducing overall system overhead.

FastAPI bridges the gap between Flask and Django by allowing minimal implementation while also providing useful built-in features like security and automatic documentation [56]. Despite FastAPI's advantages for small APIs and straightforward security, Django was selected for its maturity and user-friendliness. Django's features cater to a broad range of systems, making it familiar to developers with experience in similar projects, even without a deep understanding of the entire backend implementation.

The selection of an accompanying database was another smaller decision. Django's ORM directly links and maps objects to database storage [58]. Among the options, PostgresDB offered the most extensive feature set, including array fields, a valuable tool for storing an optional dynamic amount of required environment variable fields for device integration, and was therefore chosen.

Kubernetes

When confronted with the task of selecting a suitable framework for multi-device management, Kubernetes stood out as an unequivocal choice. The feasibility of developing a bespoke solution specific to this application was contemplated; however, the pre-existing advantages of Kubernetes presented a compelling argument in its favor. Other contenders, such as Docker Swarm [59], were duly considered, but they fell short of matching the robustness and flexibility afforded by Kubernetes, and they lacked a comparably extensive community and ecosystem.

The principal attributes that distinguish Kubernetes include its inherent support for rolling updates, self-healing capabilities, and scalability potential [42]. These, in tandem with an expansive community and a thriving ecosystem, position Kubernetes as a comprehensive solution in the realm of container management. Hence, the strategic decision was to leverage the extensive capabilities of Kubernetes, rather than pursuing the development of a custom solution.

Time Series Database - InfluxDB

Since most of the data would be time series data, a dedicated time series database was therefore considered. InfluxDB is highly specialised for a lot of time series data, both in storing and querying this well-suited data [50]. It is built for high write and query loads, which means that it will be well suited for the future if the system scales into more houses, with several sensors with high-time precision data. It also boasts horizontal scalability, allowing multiple servers to operate the same database concurrently, thereby eliminating the need for a single powerful server to run everything. The only thing to consider here is whether or not it will be overkill for its purpose. If the load of the system will not exceed millions of devices, it might have been easier to go with a normal SQL database. However, it is still interesting as a showcase of how it could scale to that amount, even if it is giving up a bit of usability. Since, this choice essentially means that two different databases are needed, as normal data can not be stored in the time series database.

4.3.2 Fail fast, a view on fault tolerance

A big scope of this system will be the fault tolerance, both for the individual devices as well as the control nodes when they are implemented. For this project, a fail-quick approach has been used. Since every single device can fail in a lot of different ways, it is easier to make things fail and then recover into a safe state. This is then based on the assumption that the device implementations are robust enough to be able to boot into such a safe state. This initialization state will then always be possible to achieve, if the vendor API is accessible, and a connection can be made. The fail-quick approach is aided by Kubernetes, if the individual Python device containers crash for some reason or another, it will try to boot up again immediately, and if it crashes again subsequently it will do so just with an increasing interval between reboots, so as to not overload the system. This interval caps out at 5 minutes. As a result, this ensures that all devices try to boot up again no matter what. As an example, in the event that an API goes down, the system will reboot the node until it works again.

This in turn means that there is no need for error-handling logic in the device nodes themselves, as they can crash, and the system as a whole takes care of it instead. This does, however, mean that the devices need to crash instead of getting stuck in an error-handling loop. If all errors are handled within the respective container, the node won't restart since the rest of the system cannot recognize its malfunction. For new device integrations this will therefore also have to be respected.

This approach has been chosen as the number of possible failure states is so massive that there is no way that all of them can be handled sufficiently. If the system instead expects the devices to crash and then handles it from there, it will be more future-proof as a result.

4.3.3 Performance considerations

Since the system introduced in Section 4.1 would be running on more powerful hardware than the previous system by Eric [1], performance is not as important. However, it will still be necessary if this is to be scalable. Keeping Docker containers small and efficient will help a lot with the scalability of the system. The Docker containers are created using the smallest possible Python image based on what is needed within it, and only the necessary packages are installed. If several containers run on the same images this will also decrease the amount of memory needed, as Docker handles this overlap. Currently, the Docker containers vary a bit in size but sit at around 300 MB. That is why all the devices that can, are using the same Python images as well. To further improve performance, statically typed languages can be used to further reduce memory and CPU usage, while also making the integrations more robust.

4.3.4 The modular structure of the system

The main idea behind the structure of the system has been modularity. If this system is to be reused by students who come after, it might not be that the entire system will be necessary, but if the parts that are needed can be easily reused, then this would be a success. In addition to this, it will make it easier to maintain the various parts, as well as develop new parts for it. It will also be a fact that some of the choices taken in a system on this scale will not be optimal, and the easier it is to remedy these changes, the better for the system in the long run. The message passing aids this modularity, making modules more interoperable, as changing out a piece of the system only has to respect the same input and output.

4.3.5 Measures in view of a large-scale deployment

There are various ideas put into practice in the infrastructure of the software implementation, to facilitate the large-scale deployment of this system in the future. Some considerations

- The entire system resides within a Kubernetes cluster, which makes the addition of redundancy a feasible task. This can be achieved by incorporating more clusters into the network. Furthermore, it may be worth considering the adoption of a multi-control plane architecture. In this design, the Kubernetes control plane operates from multiple locations, thereby mitigating the

risks associated with potential events such as power loss or hardware failure.
- Communication over MQTT allows for the use of practically any programming language for device integrations in the future, allowing more developers with different backgrounds to implement integrations.
- Integrations are made using docker containers of the smallest possible sizes, making them as atomic as possible only performing the task they are supposed to.
- Integrations are polling at random intervals, for instance, the Weather forecast integration, will sample as often as allowed but will also add a random offset to the sampling time to spread out the bandwidth use. This also helps when more devices are booted up within the same system, as the load of the integrations are naturally spread out in time, freeing up the CPU.

4.4 Available devices

At present, device support is confined to those enumerated in this section. It is important to underscore that only devices featuring some form of cloud integration can be integrated into the system. A template to facilitate the addition of new devices has been made, simplifying the integration process for devices that offer some kind of open API.

The data fields associated with devices aim to maintain as much generality as possible, with the ultimate objective of allowing a diverse set of devices to function within the same system. For instance, monitoring power consumption shouldn't necessitate a specific device as long as the devices provide the same data type. A case in point involves the Mill Sense and Verisure Smoke Detector sensors, both of which deliver airSensor measurements. Given that both sensors offer humidity and temperature measurements, it becomes irrelevant to discern which sensor provides this data. The sole distinguishing feature is that Mill Sense offers two additional data fields, TVOC and eco2. In practice, this shows up the same way in the database, just with two additional fields. Future integrations that use these fields must check that they are available.

Looking ahead, it is our hope that this list will continue to grow as other students onboard new houses equipped with novel devices into the system. This expansion will further enhance the versatility and robustness of this smart-home ecosystem.

4.4.1 Tibber / Tibber Pulse

Tibber [27], the power provider, makes hourly consumption available when used in their home. They also sell a Tibber Pulse unit [14] that connects to the HAN port of any mandatory smart meter in Norway [16]. This unit can then stream real-time data (in the order of every 2 seconds) through their open API. It requires an API key to start.

Database group(measurement name): **powerConsumption**
Available data for the *tibber-realtime* integration can be found in Table 4.2 and data from the *tibber-hourly* integration can be found in Table 4.3.

Table 4.2: Available data from the Tibber Pulse device.

Field	Unit	Explanation
accumulatedConsumption	kWh	Total consumption since midnight.
accumulatedCost	NOK	Cost of energy since midnight.
power	W	Current consumption

Table 4.3: Available data from the Tibber api.

Field	Unit	Explanation
hourlyCost	NOK	Price for consumption last hour.
hourlyPower	W	Watt consumed during last hour.

If the power provider for the house is also Tibber, historical data is also logged, allowing for the *tibber-hourly* device to be used, logging historical data as well, which can be more accurate, but is only accessible every hour.

4.4.2 Sensibo Sky

The Sensibo Sky device is a small device that mimics the remote controls of heat pumps. It also has a temperature and humidity sensor built in. In this way, it is possible to control more or less any type of heat pump by using this same device, which makes it ideal for a system that should work on as many types of heat pumps as possible.

The Sensibo Sky device integration connects to this API and logs the available data. However, this is also an API that allows for control, which means that the target temperatures of the system can be set through code. This device requires only an API key to start.
Database group(measurement name): **heatPumpController**
Available data from the *sensibo-sky* integration can be seen in Table 4.4.

4.4.3 Systemair VSR-500

The supervisor's house had a ventilation system installed, and accompanied by this system was a SAVE CONNECT cloud unit for accessing and controlling the system from remote locations [60]. This cloud interface was then turned into a device integration. It is useful for logging data from it, but it is also possible to control the system through the same API. The node requires a username and password to boot up.
Database group(measurement name): **ventilationControl**
Available data from the *systemair-vsr500* integration can be found in Table 4.5.

Table 4.4: Available data from the Sensibo Sky device.

Field	Unit	Explanation
fanLevel	low/medium/high	Speed of fans.
humidity	%	Relative humidity.
mode	cool/heat/auto	Mode of the heat pump.
on	true/false	If the heat pump is on or off.
targetTemperature	°C	Target temperature of the heat pump.
temperature	°C	Temperature seen in unit.

Table 4.5: Available data from the Systemair VSR500 device.

Field	Unit	Explanation
extractAirSpeed	%	Speed of extract air fan.
extractAirTemperature	°C	Temperature of extracted air.
humidity	%	Relative humidity.
outdoorTemperature	°C	Temperature outdoors.
supplyAirFanSpeed	%	Speed of supply air fan.
supplyAirTemperature	°C	Supply air temperature
temperatureSetpoint	°C	Temperature setpoint for inflow air.

4.4.4 Mill Sense

Mill Sense sensors are easily acquired sensors for indoor climate. They supply useful data for better control of the indoor climate. The Mill Sense device nodes start up and log all data from all Mill Sense devices in the home, as there can be multiple of these devices, and stores their data based on their location information. The Mill Sense requires a username, password, API key and access key to boot up.
Database group(measurement name): **airSensor**
Available data from the *mill-sense* integration can be found in Table 4.6.

Table 4.6: Available data from the Mill Sense device.

Field	Unit	Explanation
humidity	%	Relative humidity.
tvoc	ppb	Total volatile organic compounds.
temperature	°C	Current temperature.
eco2	ppm	CO2 in the air.

4.4.5 MET

For control of heat or power distribution systems, the weather could be an important metric. That is why the free MET API accessing forecasts for weather in

Norway has been added as a device. By supplying coordinates, this device will start logging forecasts for that area. No further API keys or credentials are needed for this logger to boot up.
Database group(measurement name): **weatherForecast**
Available data from the *met* integration can be found in Table 4.7.

Table 4.7: Available data from MET.

Field	Unit	Explanation
cloudAreaFraction	%	Fraction of sky covered in clouds.
humidity	%	Forecasted relative humidity.
temperature	°C	Forecasted temperature.

4.4.6 Solcast

Solcast is an online API supplying solar forecasts based on location. This data can be especially useful in future research projects involving solar panels and optimal distribution of batteries and grid power. That is why this API has also been added as its own device. The device connects to the API based on an API key and location, and starts logging solar forecasts every 6 hours, to stay within the free quota of requests per user.

To use this API the user has to create their own account, to get their API key.
Database group(measurement name): **solarRadiation**
Available data for the *solcast* integration can be found in Table 4.8. And more info about their API, can be found in their documentation [61].

Table 4.8: Available data from Solcast.

Field	Unit	Explanation
azimuth	°	Solar Azimuth Angle
zenith	°	Solar Zenith Angle
diffuseHorizontalIrradiance	W/m^2	Diffuse Horizontal Irradiance.
directNormalIrradiance	W/m^2	Direct Normal Irradiance
globalHorizontalIrradiance	W/m^2	Global Horizontal Irradiance.
temperature	°C	Forecasted temperature.

4.4.7 Chainpro / Victron

Chainpro is a complete battery/solar system that includes solar panels, an in-house battery, and distribution between these and the grid. This allows for the system to offload the grid power when the prices are high and charge up the battery when the prices are low. This data is available through their vendor Victron, where users can log in through a user interface. However, all data is also available through their MQTT server, where write requests can also be sent to control the system. This device integration, therefore, retrieves data through their MQTT server.

To be able to use this interface, a username, password and a Victron Remote Management (VRM) ID are required.
Database group(measurement name): **solarBatterySystem**
A lot of data can be retrieved from their MQTT server, but the ones that have been chosen to be logged in the system by the *victron* integration can be found in Table 4.9.

Table 4.9: Available data from Chainpro battery system.

Field	Unit	Explanation
batteryCharge	%	Battery percentage on the battery unit.
batteryPower	W	Produced/Consumed power from the battery.
gridPower	W	Consumed power from the grid.
powerConsumption	W	Gridpower minus batteryPower.
solarPower	W	Power produced by solar panels.

4.4.8 Verisure Smoke Detectors

Verisure has several kinds of devices ranging from door locks, alarm systems and smoke detectors. The smoke detectors are of special interest as they also include temperature and humidity measurements. Since a lot of houses already have devices like this installed, it opens up the possibility of having temperature sensors in every room without having to buy special temperature sensors like the Mill Sense. This would allow even more houses to be susceptible to this system.

There is no currently Open API for Verisure, but they are saying that they are considering it. In the meantime, they are allowing third-party solutions that mimic normal user logins. All data can therefore be retrieved through code. This has been implemented to retrieve the climate data from the smoke sensors. The device requires a username and password to start.
Database group(measurement name): **airSensor**
Available data from the *verisure-smoke-detector* integration can be seen in Table 4.10.

Table 4.10: Available data from the Verisure smoke detectors.

Field	Unit	Explanation
humidity	%	Relative humidity.
temperature	°C	Temperature seen in unit.

Chapter 5

Use Case - Closing the feedback loop

This chapter introduces a use case for the system by closing the feedback loop between temperature retrieval and inputs to a set of heat pumps. It first describes the system's configuration in Section 5.1. Thereafter the theory and the actually implemented control can be found in Section 5.2. Lastly, how the actual control was run can be found in Section 5.3. For the results from this use case, head to Section 7.4.1.

5.1 System Configuration

This section illustrates the system's capacity to administer control and complete a feedback loop, demonstrated through real-world application in the supervisor's residence during the final stages of this thesis' composition. The server accomplished this by utilizing data gathered from the system's operations. As a part of the proof-of-concept, the control was initiated in a standalone Docker container on the server, although it can also be activated within the Kubernetes cluster. To automate this process and enable control via the backend API, some level of integration is required.

The supervisor of this thesis resides in the smart-home test house at the DEC, which is fitted with four heat pumps. All these pumps are cloud-enabled via Sensibo Sky units. These units log temperature data and act as actuators, relaying reference temperatures to the heat pumps. The heat pumps, colloquially known as *main*, *studio*, *living*, and *livingdown*, will be adhering to a pre-set heating schedule.

To incorporate spot market prices into this setup, the Nordpool API, in conjunction with a Python API wrapper [62], was used to access spot market prices from Trondheim, where the house is located. This information is then incorporated into the subsequent control algorithm.

5.2 Theory and Implemented Control

A straightforward control algorithm was employed for the system. The current temperature measurements of the rooms in the house and the spot market were utilized for control purposes. The code calculates a reference temperature for the system based on a predetermined schedule and adjusts this temperature according to the spot market. The mean price for the next few hours is determined using the hourly spot market, with a higher weight given to prices closer in time as they are more likely to be accurate and may require rapid adaptation. The calculation is based on a geometric mean and can be found in Equation (5.1).

$$\text{DiscountedMeanPrice} = \frac{\sum_{k=1}^{n-1} 1.25(\text{Price}_k + \text{NettleieTarrif}_k) \cdot \gamma^{k-1}}{\sum_{k=1}^{n-1} \gamma^{k-1}} \tag{5.1}$$

In Equation (5.1), NettleieTariff represents the tariff from the electricity provider, which typically varies between day and night and, thus, influences the price. The variable k represents the timesteps into the future, and Price denotes the spot price at hour k. γ is a tunable parameter that determines the value attributed to prices further into the future. This discount is then adjusted according to another parameter, K_{price}, which serves as a gain factor for the influence of the DiscountedMeanPrice on the final offset, as depicted in Equation (5.2).

$$\text{DeltaPrice} = -K_{price} \cdot \text{DiscountedMeanPrice} \tag{5.2}$$

The DeltaPrice is subsequently utilized in a PI controller to establish a desired temperature setpoint for the heat pumps. The DeltaPrice is added to the user-defined temperature reference. Due to the system's slow dynamics, as a lot of heat is stored inside the house, the integral value is essential and accounts for the system's extended heat dynamics.

This proof of concept demonstrates that the newly created system can be operational, suggesting adaptability to multiple houses. Additionally, it highlights how a simple control algorithm utilizing such data can, in some cases, potentially enhance the system's control without the need for more complex approaches, such as MPC.

The complete Proportional-Integral (PI) control loop implementation can be seen in Equation (5.3), with the parameters described below. The actual parameters used in the test are provided in Table 7.4.

- $e(t)$ is the error at time t.
- $T_{\text{measurement}}(t)$ is the temperature measurement at time t.
- $T_{\text{reference}}(t)$ is the temperature reference point at time t.
- $I_{\text{updated}}(t)$ is the updated integrator value at time t.
- $S_{\text{on}}(t)$ is the pump state at time t (1 if "on", 0 if "off").
- α is the integral scaling factor.
- K_p is the proportional gain.
- K_i is the integral gain.

$$e(t) = T_{\text{measurement}}(t) - T_{\text{reference}}(t) \tag{5.3a}$$

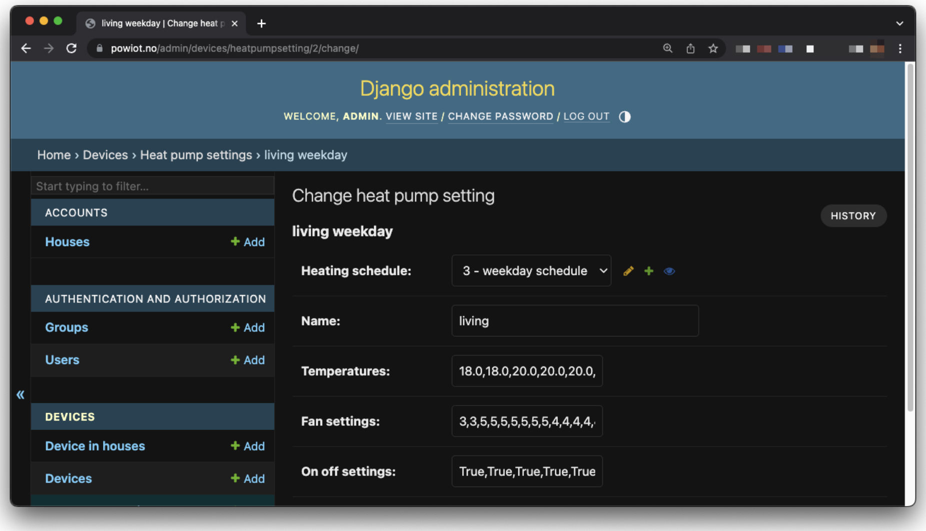
$$I_{\text{updated}}(t) = I(t-1) + S_{\text{on}}(t)(1+\alpha)e(t) - \alpha e(t-1) \tag{5.3b}$$

$$u(t) = T_{\text{reference}}(t) - K_i I_{\text{updated}}(t) - K_p e(t) \tag{5.3c}$$

5.3 Continuous Control Operation

The control script is launched using Docker Compose on the server with a configuration set to always restart in case of errors. To avoid running control on outdated values, the script only sends new signals to the heat pumps if there are updated measurements stored in the database. Consequently, the control operates autonomously alongside the rest of the system.

An endpoint was created in the Django backend to enable user control over temperature and fan settings. The structure of this data can be seen in Figure 4.3. This endpoint allows users to adjust the desired temperature and fan settings separately for weekdays and weekends. It also allows the adjustment of the parameters found in Table 7.4. This feature was implemented so the supervisor could manually modify settings while the control was running. An example of this functionality, accessed through the Django admin menu, can be seen in Figure 5.1. At each timestep, which was set to every 5 minutes, the control algorithm retrieves these user-defined settings and updates the temperature accordingly. The final result of this in action can be seen in Figure 7.4.


living weekday | Change heat p
powiot.no/admin/devices/heatpumpsetting/2/change/
Django administration
WELCOME, ADMIN. VIEW SITE / CHANGE PASSWORD / LOG OUT
Home › Devices › Heat pump settings › living weekday
Start typing to filter...
ACCOUNTS
Houses + Add
AUTHENTICATION AND AUTHORIZATION
Groups + Add
Users + Add
DEVICES
Device in houses + Add
Devices + Add
HISTORY

Change heat pump setting

living weekday

Heating schedule:	3 - weekday schedule
Name:	living
Temperatures:	18.0,18.0,20.0,20.0,20.0,
Fan settings:	3,3,5,5,5,5,5,5,4,4,4,4,
On off settings:	True,True,True,True,True

Figure 5.1: A look at how the heat-pump settings admin panel looks.

Chapter 6

Using, Managing and Developing the system: A Comprehensive Guide

In this section, you will find a thorough guide covering all aspects of the system. It begins with an introduction on how to use the system that is already running, followed by instructions on how to develop for it and how to set everything up from scratch. While the guide may seem lengthy, it is designed to be comprehensive rather than complicated. However, with everything like this it will be impossible to cover every minor detail, and thus it may require some additional research when delving deeper into some parts of the system. Instead of reading the entire chapter, skip to the section of interest. The following sections are included:

- Section 6.1: An intro guide to this guide.
- Section 6.2: An overview of the repository.
- Section 6.3: Shows how to access/modify and use the time series database directly.
- Section 6.4: Shows how to navigate and use/test the Django backend API.
- Section 6.5: Shows how to connect to the server for developer changes/-monitoring.
- Section 6.6: Shows how to check the status of running containers, secret management, and overall system changes.
- Section 6.7: Outlines the use of the development repository.
- Section 6.8: Details everything regarding device integrations.
- Section 6.9: Details developing the Django backend.
- Section 6.10: Introduces how to set up the production server from scratch.
- Section 6.11: Shows how to build docker containers.
- Section 6.12: Has some info about some issues that may arise and fixes for them.

6.1 A guide to the guide

For a lot of commands, the actual values have been replaced by <USERNAME-HERE> like values, to indicate that you should exchange that part with your own data. An example of how to do this would be to exchange the following:

```
$ ssh <USERNAME>@<PUBLIC-IP-HOST-MACHINE>
```

With a username: admin and an IP: 192.168.0.1, the result would be:

```
$ ssh admin@192.168.0.1
```

It's crucial to point out that in the terminal examples, new lines start with $.

Please refer to Section 6.10.2, particularly if you're executing commands on the server, because we've replaced *microk8s* with an alias to make the commands more general.

This guide is standalone, but it's beneficial to have some basic knowledge of Docker [37] and Kubernetes [63] to fully leverage it. If these technologies are new to you, consider doing some preliminary reading. At the very least, check out the theoretical background provided in Section 2.2.

6.2 Overview of the Repository Structure

The repository for all of the software created in this thesis can be found at `https://github.com/thomabsk/powiot-smart-home`. It is however private, and you would have to contact the group for access. The structure of the entire repository can be found in the tree below.

It consists of four folders. The *backend* holds the code for the Django backend. *k8s* holds the yaml files for booting up the various Kubernetes deployments. *src* has the source code for the different device integrations, and new ones are also added here in their own folders. *processing* holds the code for the control use case laid out in Chapter 5, as well as some testing regarding analytics, where it queries the house data and tries to correlate the outside temperatures with heating and draws some interesting analytics about this.

```
powiot-smart-home/
├── backend/
│   ├── backend/
│   ├── accounts/
│   ├── devices/
│   ├── powiot/
│   ├── .env.example
│   ├── .env.prod.example
│   ├── manage.py
│   └── .../
├── k8s/
│   ├── devices/
│   ├── django/
│   ├── logging/
│   ├── mqtt/
│   ├── postgres/
│   └── cloudflare-tunnel.yaml
├── src/
│   ├── device-template/
│   ├── datalogger/
│   ├── mill-sense/
│   └── .../
├── processing/
│   ├── Analytics/
│   ├── temperature-control/
│   ├── BasicDR/
│   └── casadi-test/
├── skaffold.yaml
├── docker-compose-django-database.yaml
├── docker-compose-device-logging.yaml
├── docker-compose-casadi.yaml
├── README.md
└── .env.example
```

6.3 Interacting with the InfluxDB Database

6.3.1 Using the User interface

The web user interface must be made available through port forwarding or some kind of proxy. If done, you can access the web interface. It is naturally forwarded to *localhost:8086*, but to make it accessible from anywhere the above has to be true. During this project, it was made available through a custom domain at `https://influx.powiot.no`. To log in you need a user account, a username and a password. For admin credentials contact the team.

Through the web interface, you can access data by querying it, see all current tokens and create new ones, and also add new plugins. It is not currently possible to delete data or create users through the web interface. To do this head to Section 6.3.2.

Accessing data

Follow the instruction seen in Figure 6.1 to query and visualize the data in the browser. Queries can be filtered in any order, and the time range can be chosen as you want. To display the data you have to choose an aggregate function. This just entails where to place data points on the time axis.

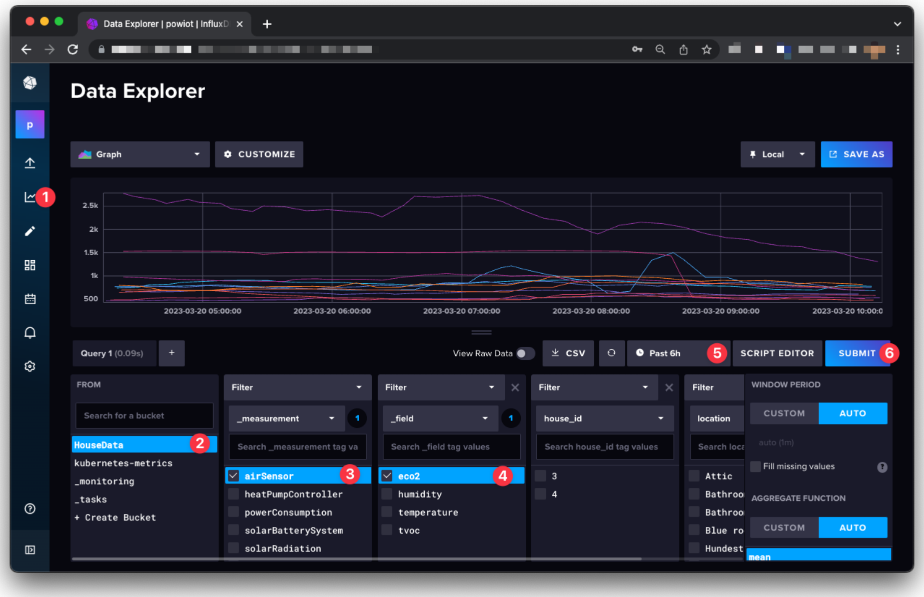


Figure 6.1: 1. Click to head to the data tab. 2. Choose a bucket. 3 and 4. filter the data. 5. Choose a time range. 6. Submit the query.

Creating tokens

To create a new token head to the API tokens tab as seen in Figure 6.2. Here an admin token can be made, or a token with reduced permissions. Always make the token with the least permissions needed to do the task, this makes it harder to accidentally delete or change data. Tokens can be deleted when they are no longer needed from this same interface.

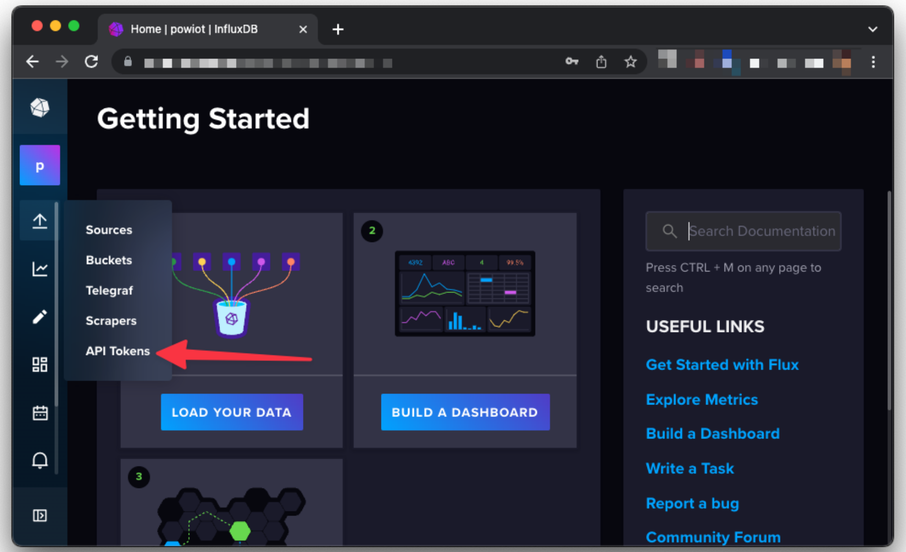


Figure 6.2: Where to find the InfluxDB token tab.

6.3.2 Utilizing the Command line interface

The database can be interfaced through a command line interface. To do this first install it using the latest instructions found in the InfluxDB documentation [64]. Then create your *influx config*, it requires a custom name for it, the web address for the database, a token with privileges needed for the action you are going to do, and the organization name within the database, in that order in the example below.

```
$ influx config create \
 -n powiot-config \
 -u https://influx.powiot.no \
 -t <TOKEN-HERE> \
 -o powiot
```

Then set the config as active.

```
 $ influx config set -n powiot-config --active
```

Ping it to see if it is working.

```
 $ influx ping
```

If the result of the ping is: *OK*, then you can move on.

Creating users

To create a new user to access the database directly, do the following.

```
$ influx user create -n <USERNAME> -p <PASSWORD> -o <ORGANIZATION_NAME>
```

But also notice that if someone should just have read access, for example, generating tokens for this is better and can be done through the user interface as found in Section 6.3.1.

Retrieving data

Data retrieval can be done using the web interface and downloading CSV files from there, or through the REST API, with example Python code found in Section 6.3.3.

Backing up data

To backup all the data in the database to your current folder, open a terminal in the folder you want the backup to be made in, and then run:

```
$ influx backup .
```

A complete database backup will then be written to the current folder.

Deleting data

CAREFUL: By using this, data will be removed forever, with no way of restoring without a backup. Therefore do the steps in Section 6.3.2 first so that the data is backed up in case of errors. An admin token with all privileges is also needed to perform this action.

To delete data from the bucket, a start time, end time and filters have to be specified. The filters are specified through the predicate as a string. In the below example *airSensor* measurements with the tags *house_id* = 3 and *location*="*Stue, sentrum*" will be deleted. Notice that there are strings within the strings.

```
$ influx delete --bucket HouseData \
--start 2023-02-13T16:50:00Z \
--stop 2023-03-03T00:00:00Z \
--predicate '_measurement="airSensor" \
AND house_id="3" AND location="Stue, sentrum"'
```

6.3.3 Accessing through Python code

Access can also naturally be made remotely or locally through Python code using the Python client for InfluxDB. The URL of the InfluxDB database will be *influxdb-service:8086* within the Kubernetes cluster. This can also be found by issuing

```
$ kubectl get service
```

And the influxdb-service will be one of the ones running. The URL of the InfluxDB database remotely is `https://influx.powiot.no`.

Simple example

```python
import influxdb_client

house_id = "3" # The ID of the house for which the data is being queried
bucket = "HouseData" # The bucket in which the data is stored in the database
org = "powiot" # The organization that manages the bucket
token = "TOKEN-HERE" # The API token used for authentication
url = "https://influx.powiot.no" # The URL where the database is hosted
measurement = "airSensor" # Database grouping
field = "temperature" # Field to query

client = influxdb_client.InfluxDBClient(
    url=url,
    token=token,
    org=org,
) # Creating an InfluxDB client

# Creating a Flux query. The query fetches data from the specified bucket,
# filters records based on the measurement, field and house_id values and finally
# gets the data from the last 20 minutes.
query = f"""
from(bucket:\"{bucket}\")
|> range(start: -20m)
|> filter(fn:(r) => r._measurement == \"{measurement}\")
|> filter(fn:(r) => r._field == \"{field}\")
|> filter(fn:(r) => r.house_id == \"{house_id}\")
"""
print(f"Query:\n{query}")

query_api = client.query_api()
result = query_api.query(org=org, query=query) # Querying the database

# Convert result into a dictionary
results = {}
results[measurement] = {}
for table in result:
    for record in table.records:
        location = record.values["location"] #The field that groups the data
        if location not in results[measurement]:
            # Initializing the dictionary for each location
            results[measurement][location] = {}
            results[measurement][location]["field"] = record.get_field()
            results[measurement][location]["time"] = []
            results[measurement][location]["value"] = []

        # Appending each record's time and value to the respective lists
        results[measurement][location]["time"].append(record.get_time())
        results[measurement][location]["value"].append(record.get_value())

print(results)
```

Code listing 6.1: Simple Python example for accessing the InfluxDB data. Note that the token has to be added for the example to work.

A simple example of Python access can be seen in Code listing 6.1. Make sure to change the token to one that has the right access. For an overview of the different measurements and their fields, go to Section 4.4. Notice, however that only houses with these devices enabled will have the corresponding data. The current houses and their devices can be seen in Table 7.3. The flux query which can be found in the query variable is the flux query sent to the database. Flux queries can do a lot, and for more information about them, visit the influxDB flux documentation [65].

This Python code uses the InfluxDBClient from the influxdb_client package to connect to an InfluxDB database and query it for specific data. The code is written to be specific to an airSensor's temperature measurements in a particular house over the past 20 minutes.

The script queries the database for data from a specific measurement (representing a set of data) and field (a particular aspect of the data) for a specific house. The query is written in InfluxDB's Flux query language.

The resulting data is then converted into a dictionary for easier handling in Python. The dictionary is structured such that each location's temperature measurement (recorded by the airSensor) is stored along with the respective times of measurement.

Longer example of access

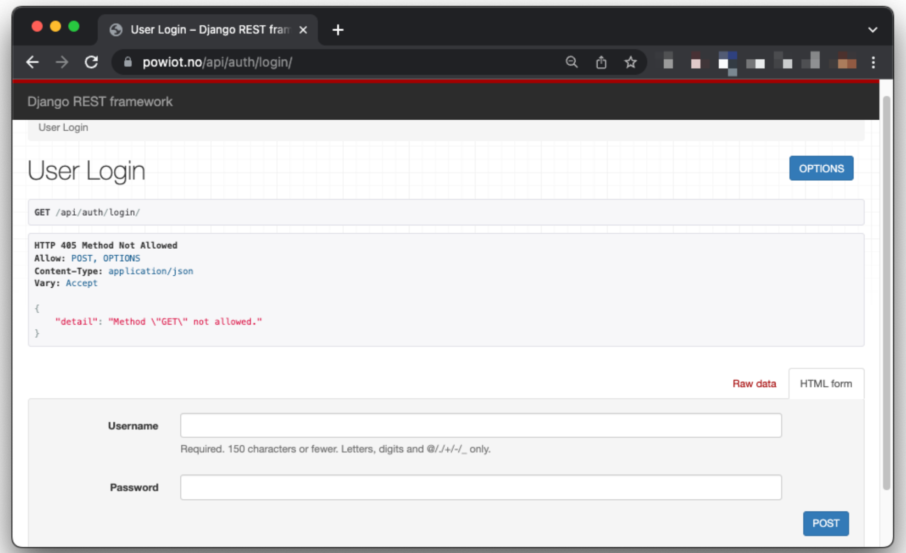
A more detailed example of accessing and querying which parameters are in the database can be found in Appendix A. This is a complete example, showing how to filter data, showing the structure of the data, and also how to turn the data structure into a Python dictionary. Notice that the API_KEY/TOKEN has to be set manually at the top of the file.

6.4 Using the Backend REST API

Since the backend is based on the Django REST framework it comes with a browsable API by default. This means that the available API endpoints are also browsable with a web browser, and not just by issuing requests directly. Another student is currently working on the frontend for the system, which will allow users to access this backend through that instead. This section, however, outlines a couple of other ways of accessing it directly, since the frontend will just be a wrapper around this backend.

6.4.1 Using the browsable API

Every endpoint found in the backend can also be visited through the browser. By going to the URL in the browser, a page will pop up where the user can simulate sending requests. Figure 6.3 is an example of what this looks like. It is also possible to insert data and post requests through this site. For a list of all endpoints, look in Section 6.4.3.



```
GET /api/auth/login/
```

```
HTTP 405 Method Not Allowed
Allow: POST, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "detail": "Method \"GET\" not allowed."
}
```

Username:

Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.

Password:

Figure 6.3: An example of the browsable API, here looking at the `https://powiot.no/api/auth/login` endpoint.

6.4.2 Python example access

Below is a simple example of logging into the Django backend using the requests library in Python and then accessing one of the endpoints. Here to find details about the house of the logged-in user. To run the code, the requests library has to be installed.

```python
import requests

# Set the login URL and user credentials
login_url = "https://powiot.no/api/auth/login/"
username = "<USERNAME-HERE>"
password = "<PASSWORD-HERE>"

# Create a session object and make a POST request to the login URL
session = requests.session()
login_data = {"username": username, "password": password}
session.post(login_url, data=login_data)

# Make a GET request to the protected resource
protected_resource_url = "https://powiot.no/api/houses/"
response = session.get(protected_resource_url)

# Turn the received response into a Python dictionary.
data = response.json()

# Print the response content
print(data)
```

To install the requests library run the following code.

```
$ pip install requests
```

6.4.3 API documentation

Complete swagger documentation can be found on the website by going to `https://powiot.no/api/docs`. However, complete endpoint documentation is also available in Appendix B.

6.5 Connecting to the server

6.5.1 Accessing the server

To access a remote Linux server/workstation of any kind the most typical choice is Secure Shell (SSH). SSH creates a secure tunnel to the host machine, allowing a remote computer to open a terminal on the host machine. For this to be possible the remote user has to have credentials; a username and a password for the machine, as well as its public IP address. However, it also needs to have its port opened for SSH access. This port is normally port 22. For the supercomputer, this machine is available while connected to NTNU, or using a VPN to connect to NTNU. The IP for this particular machine can be retrieved by contacting the responsible people for the project.
To then connect to the machine issue the following command from a terminal.

```
$ ssh <USERNAME>@<PUBLIC-IP-HOST-MACHINE>
```

You will then be prompted to enter the password, and you will gain terminal access to the machine. If you're not a sudo user, contact the people responsible for the machine to get this access.

6.5.2 Set up user

For the user setup on the server, sudo access might not be needed depending on what you must do. The necessary packages should also already be installed and commands should just be able to be issued right away. To test this, try to run the following commmand.

```
$ microk8s kubectl get pods
```

You may get an error that you have insufficient permissions to access Microk8s. You will then either need sudo or you would need to be added to the Microk8s user group. An error with possible fixes will pop up. The most practical one is listed below.

```
$ sudo usermod -a -G microk8s <YOUR-USERNAME>
```

6.6 Kubernetes management

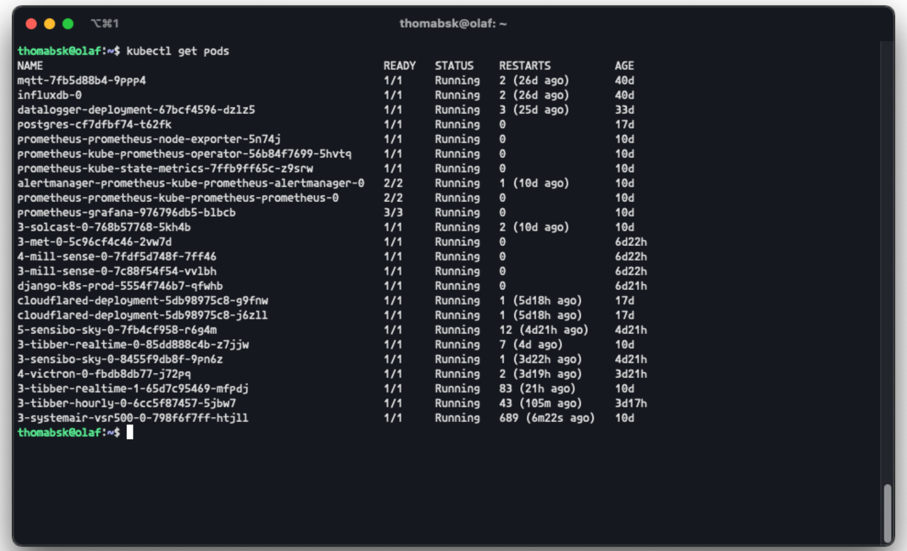
IMPORTANT: Before proceeding with any actions in this section, ensure that you have followed the steps in Section 6.5.1 to connect to the server and then Section 6.10.2 to enable direct usage of ***kubectl*** commands without the need to prefix them with ***microk8s*** every time. This section can be used for testing purposes, troubleshooting issues, or manually managing pods. Normally, pod startups and shutdowns are controlled by the backend API, but these commands allow for manual overrides.

6.6.1 Monitoring running pods

To check on already running pods, there are a few useful commands. To list all pods currently running in the system under the default namespace run the following command.

```
$ kubectl get pods
```

An example of the above command can be seen in Figure 6.4. The format of the pod names is the following: <HOUSE-ID>-<DEVICE-NAME>-<COUNT>-<RANDOM-HASH>. The count is the number of similar devices running under the same house, and the hash is automatically created by Kubernetes when a new pod boots up.



```
thomabsk@olaf:~$ kubectl get pods
NAME                                                    READY   STATUS    RESTARTS         AGE
mqtt-7fb5d88b4-9ppp4                                    1/1     Running   2 (26d ago)      40d
influxdb-0                                              1/1     Running   2 (26d ago)      40d
datalogger-deployment-67bcf4596-dzlz5                   1/1     Running   3 (25d ago)      33d
postgres-cf7dfbf74-t62fk                                1/1     Running   0                17d
prometheus-prometheus-node-exporter-5n74j               1/1     Running   0                10d
prometheus-kube-prometheus-operator-56b84f7699-5hvtq    1/1     Running   0                10d
prometheus-kube-state-metrics-7ffb9ff65c-z9srw          1/1     Running   0                10d
alertmanager-prometheus-kube-prometheus-alertmanager-0  2/2     Running   1 (10d ago)      10d
prometheus-prometheus-kube-prometheus-prometheus-0      2/2     Running   0                10d
prometheus-grafana-976796db5-blbcb                      3/3     Running   0                10d
3-solcast-0-768b57768-5kh4b                             1/1     Running   2 (10d ago)      10d
3-met-0-5c96cf4c46-2vw7d                                1/1     Running   0                6d22h
4-mill-sense-0-7fdf5d748f-7ff46                         1/1     Running   0                6d22h
3-mill-sense-0-7c88f54f54-vvlbh                         1/1     Running   0                6d22h
django-k8s-prod-5554f746b7-qfwhb                        1/1     Running   0                6d21h
cloudflared-deployment-5db98975c8-g9fnw                 1/1     Running   1 (5d18h ago)    17d
cloudflared-deployment-5db98975c8-j6zll                 1/1     Running   1 (5d18h ago)    17d
5-sensibo-sky-0-7fb4cf958-r6g4m                         1/1     Running   12 (4d21h ago)   4d21h
3-tibber-realtime-0-85dd888c4b-z7jjw                    1/1     Running   7 (4d ago)       10d
3-sensibo-sky-0-8455f9db8f-9pn6z                        1/1     Running   1 (3d22h ago)    4d21h
4-victron-0-fbdb8db77-j72pq                             1/1     Running   2 (3d19h ago)    3d21h
3-tibber-realtime-1-65d7c95469-mfpdj                    1/1     Running   83 (21h ago)     10d
3-tibber-hourly-0-6cc5f87457-5jbw7                      1/1     Running   43 (105m ago)    3d17h
3-systemair-vsr500-0-798f6f7ff-htjll                    1/1     Running   689 (6m22s ago)  10d
thomabsk@olaf:~$
```

Figure 6.4: Example of the output from the "kubectl get pods" command.

To get everything under all namespaces run the following command.

```
$ kubectl get pods --all-namespaces
```

All device pods in the system are also labelled based on their house ID, device name and count. This can also be used to only list pods from a certain house for example. A few examples of ways to use this has been shown below. First showing all pods with $house\_id = 3$ then with $house\_id = 3$ and $device\_name = met$ and lastly only devices with the $device\_name = met$.

```
$ kubectl get pods -l house_id=3
$ kubectl get pods -l house_id=3,device_name=met
$ kubectl get pods -l device_name=met
```

All device nodes are booted up as their own Kubernetes deployments since pods are orchestrated by their deployment. These deployments can be started, stopped and restarted manually, or by using the backend. This API allows for the starting of specific containers, and also the stopping of specific containers. However, it also allows for stopping of all containers linked to one house, or all containers of one specific *device_name*. This is done through the use of labelling of the pods and deployments. All the deployments are labelled both with *device_name* and *house_id*. To get deployments by label name, see the example below. -l is the label parameter.

```
$ kubectl get deployments -l <LABEL_NAME>=<LABEL>
# Examples
$ kubectl get deployments -l device_name=tibber-realtime
$ kubectl get deployments -l house_id=3
```

6.6.2 Procedure for Deleting pods

Since pods are orchestrated by their deployments, to delete them, the deployment responsible for them has to be stopped. This can also be run with the same label tags as shown previously. To delete a deployment run

```
$ kubectl delete deployment <DEPLOYMENT-NAME>
```

Where the deployment name is the name that is gotten from the list when issuing get deployments. Every device deployment also has a secret that accompanies it, this also has to be deleted manually if using this method. To get all secrets

```
$ kubectl get secrets
```

Then in the same manner run the following command to delete the accompanying secret.

```
$ kubectl delete secret <SECRET-NAME>
```

6.6.3 Reviewing Pod Logs

If you then want to check the logs for a running device, let's say you want to check the SystemAir device logs for house 3 from the results in Figure 6.4, you would use the following command. Where the pod name will be different for your case.

```
$ kubectl logs 3-systemair-vsr500-0-798f6f7ff-htjll
```

Here all information that is printed/logged from the device integration will show up. This can be useful for debugging pods that are not logging correctly or are crashing. Another useful command, especially if things are crashing unexpectedly is the following:

```
$ kubectl logs 3-systemair-vsr500-0-798f6f7ff-htjll --previous
```

Adding the previous tag will show the logs of the pod that was before the current one, therefore showing the error output before it crashed.

6.7 How to use the development repository

The development repository for the system encompasses everything required to launch the complete system in a production setting, while also including several beneficial features for local development. Consequently, the manner and extent of its utilization will depend on your specific use case. For conducting device integrations, only Docker needs to be installed. However, if backend work and integration testing against the Kubernetes cluster is necessary, additional tools will be required for a comprehensive setup. A list of all the tools required for a full installation can be found in Section 6.7.1.

6.7.1 Setting up a development environment

For reference, development has only so far been done on an M1 Mac. Instructions will be similar for a Linux distribution, but Windows development will vary, and a development environment on Windows has not been tested but should be quite similar.

In this section, there will be details about how to get one type of development setup up and running, where the main infrastructure as well as more device nodes can be booted up, and then taken down again after development is finished. This is useful both for testing existing components as well as for integrating new devices into the system before actually deploying them to the server. If things work here, they will work on the server as well. It is not necessarily needed to integrate new devices into the system but will make debugging a lot easier if things do not end up working at first.

Docker Desktop is the main part of the development infrastructure. It can host Kubernetes clusters and it is also used to build new Docker containers for use in the system. It can be installed from the Docker Desktop installation site [66].

To simulate a Kubernetes environment there are several different utilities. For this installation, Kind was chosen [67], a lightweight Kubernetes cluster which allows for quick development and setup. Alternatives such as Minikube can also be considered and there are guides on how to set this up on the Kubernetes wiki [68]. There is a bit of freedom here, and as long as some type of Kubernetes cluster is running, there will not be that much difference.
Install on macOS via homebrew

```
$ brew install kind
```

Install on Windows using chocolatey package manager

```
$ choco install kind
```

Then, to start up the kind cluster, run the following command

```
$ kind create cluster
```

If you want to remove it later, then run the following

```
$ kind delete cluster
```

After this is installed, the cluster should be running on your computer, and you should then only have to install Kubectl to access the cluster. To install Kubectl, find instructions on their website [69]. To install Skaffold, find instructions for your OS on their website [70].

Now all needed tools should be installed, for more info about developing for the system, head to Section 6.8.2 for creating integrations or Section 6.9 for developing the Django backend.

6.8 Device Integrations

6.8.1 Starting new devices

New devices are started through the backend API. It takes care of creating secrets for the devices, which hold the credentials needed to boot up the devices when they log into their respective accounts through their APIs. It also starts the devices themselves. They can be manually booted up using Kubernetes deployment files like in Section 6.10.4, however since credentials have to be input into the devices, this is automated through the backend POWIOT library which can be found in:

```
/
└─ backend/
   └─ powiot/
```

To start new devices for a house you either have to be logged in as the admin user or the user who owns that specific house. Currently, only one user can own a house. The admin account can boot up a device for any house. To login visit the login endpoint at `https://powiot.no/api/auth/login/`. After you are logged in, the username will be visible in the top right corner of the browsable API. Now a new device can be added by going to a URL that looks like this `https://powiot.no/api/houses/<HOUSE_ID>/devices/<DEVICE_ID>/new`, with house_id and device_id being the respective values of the house and device to be added to that house. An example post request is shown below, here for starting an instance of a Tibber Pulse real-time logger:

```
{
    "api-key": "erwkk32j231joprekewr",
    "pulse_index": "0"
}
```

The API-key is the one retrieved from Tibber(here randomly generated), and the pulse_index count starts at 0 and increments depending on how many pulse devices are in the house. However, the fields should represent the required fields found on that device. An error with what field is missing will be returned if not all the fields were given. With a successful ok message from the post request, a new device will have been booted up and it should be visible under `https://powiot.no/api/houses/<HOUSE_ID>/devices/`. Here the pod_state will represent the state of the pod. A refresh of the site will also refresh the state. If you are on the server you can check on the pod using commands found in Section 6.6.1.

More actions are available for the device integrations, so head to Appendix B to find out the endpoints used for restarting, updating or deleting running devices. These operate in the same manner as this one.

6.8.2 Creating new device integrations

This section will walk you through the process of creating and adding a new device to the system.

First, ensure you have cloned the development repository and have all necessary components installed as detailed in Section 6.7.1. In addition to this, a Docker Hub account is also needed for pushing the built images to it. For local development, Docker is the key tool needed.

To facilitate testing, `docker-compose-device-logging.yaml`, a Docker Compose file has been prepared. This file will help set up a MQTT broker, an InfluxDB database, and a Datalogger node, and even includes an example of how to start your custom integration. Essentially, it creates a small instance of the entire system locally. Any logging that happens here will also work in the production environment.

Along with the compose file, you will need a `.env` file which contains environment variables. These variables include credentials used to boot up the InfluxDB database as well as the individual device logging credentials. These environment variables are put into the Docker containers through the Docker Compose yaml script. Use the provided `.env.example` file as a template, copy it and rename it as `.env` in the same directory as the Docker Compose file.

Here's a step-by-step guide on how to create a new integration:

1. Write code that communicates with the device API.
2. Integrate this code into the given template.
3. Test if the device logs correctly in your local environment.
4. Upload the container to Docker Hub.
5. Use the admin panel to add the device to the backend.
6. Initiate the device for a house to test it in the production environment.

A sample device integration is available in the path:

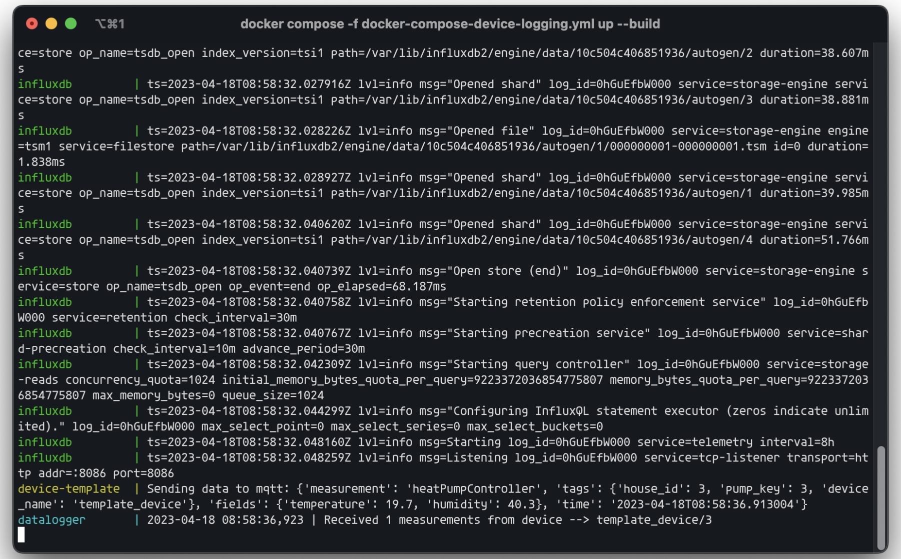
```
/
└── src/
    └── device-template/
```

This sample is in Python, but you can use any language that has a MQTT interface library. The testing process remains the same.

To test the existing integration, use the following command:

```
$ docker compose -f docker-compose-device-logging.yml up --build
```

This command builds necessary containers and initiates a testing environment. After a while, you should see an output like the one shown in Figure 6.5. At the end of this process, you should see the device template sending data and the datalogger logging it. **Important:** Ensure that the `.env.example` file is renamed to `.env` for Docker Compose to pick up the environment variables.


⌥⌘1
docker compose -f docker-compose-device-logging.yml up --build

```
ce=store op_name=tsdb_open index_version=tsi1 path=/var/lib/influxdb2/engine/data/10c504c406851936/autogen/2 duration=38.607m
s
influxdb        | ts=2023-04-18T08:58:32.027916Z lvl=info msg="Opened shard" log_id=0hGuEfbW000 service=storage-engine servi
ce=store op_name=tsdb_open index_version=tsi1 path=/var/lib/influxdb2/engine/data/10c504c406851936/autogen/3 duration=38.881m
s
influxdb        | ts=2023-04-18T08:58:32.028226Z lvl=info msg="Opened file" log_id=0hGuEfbW000 service=storage-engine engine
=tsm1 service=filestore path=/var/lib/influxdb2/engine/data/10c504c406851936/autogen/1/000000001-000000001.tsm id=0 duration=
1.838ms
influxdb        | ts=2023-04-18T08:58:32.028927Z lvl=info msg="Opened shard" log_id=0hGuEfbW000 service=storage-engine servi
ce=store op_name=tsdb_open index_version=tsi1 path=/var/lib/influxdb2/engine/data/10c504c406851936/autogen/1 duration=39.985m
s
influxdb        | ts=2023-04-18T08:58:32.040620Z lvl=info msg="Opened shard" log_id=0hGuEfbW000 service=storage-engine servi
ce=store op_name=tsdb_open index_version=tsi1 path=/var/lib/influxdb2/engine/data/10c504c406851936/autogen/4 duration=51.766m
s
influxdb        | ts=2023-04-18T08:58:32.040739Z lvl=info msg="Open store (end)" log_id=0hGuEfbW000 service=storage-engine s
ervice=store op_name=tsdb_open op_event=end op_elapsed=68.187ms
influxdb        | ts=2023-04-18T08:58:32.040758Z lvl=info msg="Starting retention policy enforcement service" log_id=0hGuEfb
W000 service=retention check_interval=30m
influxdb        | ts=2023-04-18T08:58:32.040767Z lvl=info msg="Starting precreation service" log_id=0hGuEfbW000 service=shar
d-precreation check_interval=10m advance_period=30m
influxdb        | ts=2023-04-18T08:58:32.042309Z lvl=info msg="Starting query controller" log_id=0hGuEfbW000 service=storage
-reads concurrency_quota=1024 initial_memory_bytes_quota_per_query=9223372036854775807 memory_bytes_quota_per_query=922337203
6854775807 max_memory_bytes=0 queue_size=1024
influxdb        | ts=2023-04-18T08:58:32.044299Z lvl=info msg="Configuring InfluxQL statement executor (zeros indicate unlim
ited)." log_id=0hGuEfbW000 max_select_point=0 max_select_series=0 max_select_buckets=0
influxdb        | ts=2023-04-18T08:58:32.048160Z lvl=info msg=Starting log_id=0hGuEfbW000 service=telemetry interval=8h
influxdb        | ts=2023-04-18T08:58:32.048259Z lvl=info msg=Listening log_id=0hGuEfbW000 service=tcp-listener transport=ht
tp addr=:8086 port=8086
device-template | Sending data to mqtt: {'measurement': 'heatPumpController', 'tags': {'house_id': 3, 'pump_key': 3, 'device
_name': 'template_device'}, 'fields': {'temperature': 19.7, 'humidity': 40.3}, 'time': '2023-04-18T08:58:36.913004'}
datalogger      | 2023-04-18 08:58:36,923 | Received 1 measurements from device --> template_device/3
```

Figure 6.5: Terminal output of docker-compose-device-logging.yaml.

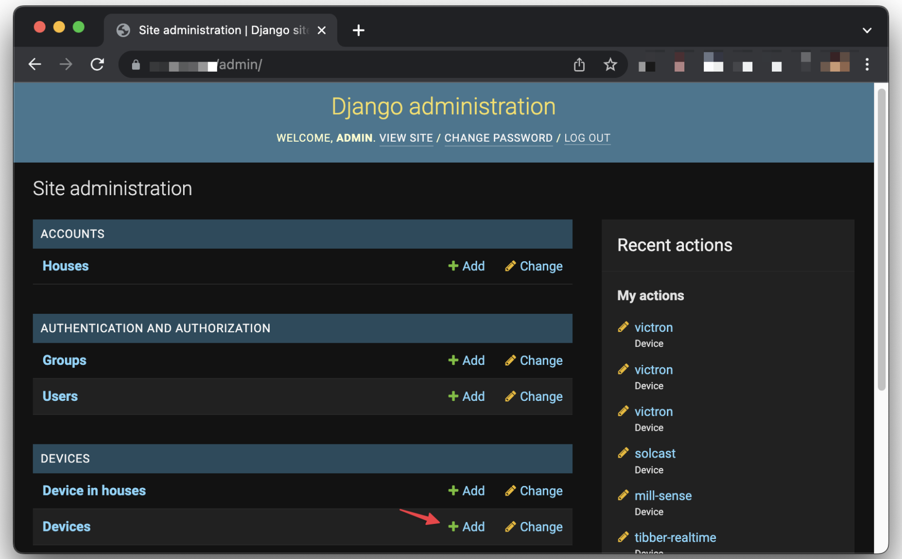
Now, you should be able to access the InfluxDB dashboard at `localhost:8086`. The default login details are admin (username) and password (password) unless you have changed them in the `.env` file. After logging into the dashboard, navigate to the data section to see if the device is logging *heatPumpController* measurements correctly.

Now, you have a functioning device that needs to be modified according to the new device requirements. The environment variables that are automatically inserted are the house ID and the sampling time. The house ID represents the ID of the house in the database. This ID has to be put as a tag on the data measurements, so that they are labelled correctly. The sampling time is a time in seconds, which tells the device how often it should query for more measurements. Check with the vendor API whether they have limits to how often you are allowed to access it.

This is where your new logic comes in. It is recommended to start creating a script for gathering data barebones first, and when this works, integrate it into this template.

The Datalogger will capture any message sent over the "sensors/#" topic. The name after "sensors/" is merely for logging purposes to help identify the origin of the message.

After implementing the new device retrieval logic and confirming that the device logs correctly, you can upload it to Docker Hub so that the server can pull the built image. A Docker Hub account is needed for this step. For more information about building and pushing Docker containers head to Section 6.11.


Site administration | Django site

[redacted]admin/

Django administration

WELCOME, **ADMIN**. VIEW SITE / CHANGE PASSWORD / LOG OUT

Site administration

ACCOUNTS

Houses + Add Change

AUTHENTICATION AND AUTHORIZATION

Groups + Add Change

Users + Add Change

DEVICES

Device in houses + Add Change

Devices + Add Change

Recent actions

My actions

victron
Device

victron
Device

victron
Device

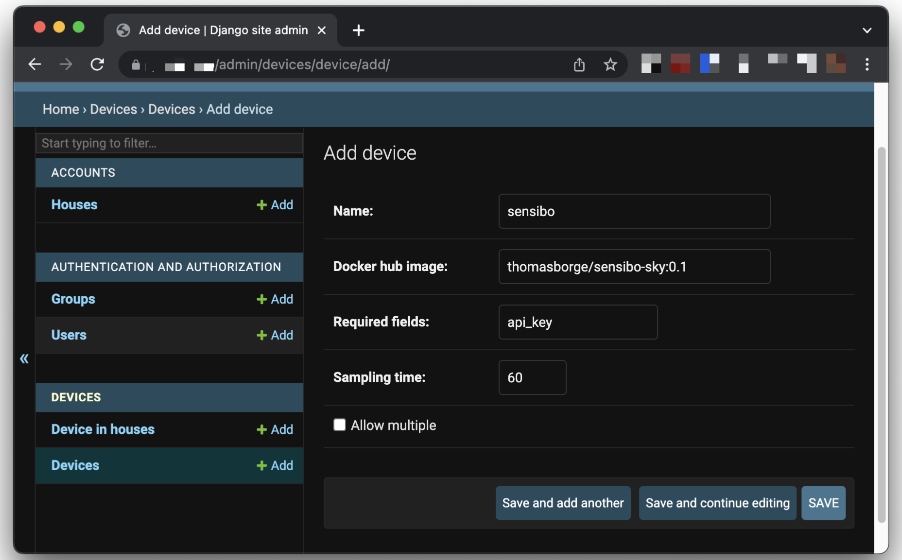
solcast
Device

mill-sense
Device

tibber-realtime

Figure 6.6: An overview of the admin panel, and where to add new devices.

Once the Docker image for a new device is prepared and uploaded, you can add it to the backend. Navigate to the admin panel at `https://powiot.no/admin`, and click on 'New Device' as shown in Figure 6.6. An example of data input can be found in Figure 6.7. If more than one field is required, separate them with a comma. Make sure that the Docker Hub image ID matches the one you uploaded. Now this device added in the backend will have enough information about the integration to boot it up. When started, the required fields will be input as environment variables into the container, and the Docker image will be pulled to start it.


Add device | Django site admin
/admin/devices/device/add/
Home › Devices › Devices › Add device
Start typing to filter...
ACCOUNTS
Houses + Add
AUTHENTICATION AND AUTHORIZATION
Groups + Add
Users + Add
DEVICES
Device in houses + Add
Devices + Add
Save and add another
Save and continue editing
SAVE

Add device

Name:	sensibo
Docker hub image:	thomasborge/sensibo-sky:0.1
Required fields:	api_key
Sampling time:	60

☐ Allow multiple

Figure 6.7: An example input of how to fill out the new device form, required fields are comma-separated if more than one. The name should be all lowercase and represent the device/sensor. The sampling time is represented in seconds. Allow multiple allows multiple of this type of device to be booted per house, this is often not necessary as one integration will retrieve all the data from various sensors in one house.

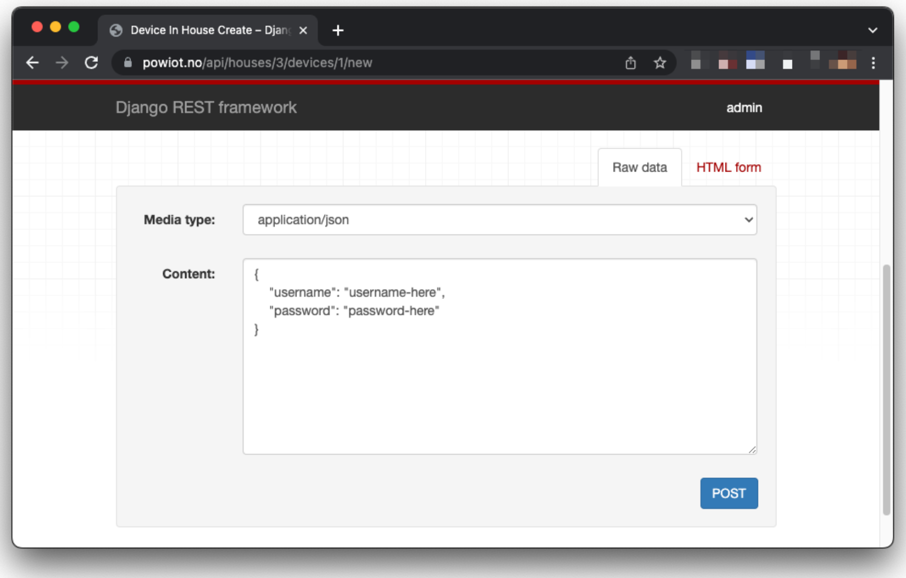
To view the test user house, visit `https://powiot.no/api/houses/2/`. Then find the ID of your newly created device. A list of all devices can be found at `https://powiot.no/api/devices/`. If your new device has the ID 8, add it to the house by navigating to `https://powiot.no/api/houses/2/devices/8/new`.

In the raw form section, input your data matching the required fields in the database. These values will be used within the respective device Docker container that boots up. JSON is quite strict about its formatting, so ensure that indentations are 4 spaces. Here is an example of this:

```
{
    "username": "username-here",
    "password": "password-here"
}
```

NOTICE: The username and password in this example is the username and password credentials used in the device integration to for example log into the Mill API if it is the Mill integration. An example of what this looks like on the website can be seen in Figure 6.8. After issuing a post command the device should boot up.

After this is done the device should be logging and it might be useful to check the logs for it to ensure it is working, for info about this visit Section 6.6.3.


Device In House Create – Djan
powiot.no/api/houses/3/devices/1/new
Django REST framework
admin
Raw data
HTML form
POST

Media type: application/json

Content:

```
{
    "username": "username-here",
    "password": "password-here"
}
```

Figure 6.8: Form data input example for starting new devices.

6.8.3 Change existing integrations

Changing existing integrations is done in a quite similar way to what is found in Section 6.8.2. Locate the existing code for the integration under:

```
/
└─src/
   └─<DEVICE-NAME>/
```

After that create/change the .env file and docker compose file to boot up this device instead. Credentials are needed for testing to see if it works. Some reading up on Docker Compose files might be necessary. However, examples are already available in the `docker-compose-device-logging.yaml` file, and just change the generic device to the one you are currently working on, remember to change the build folder to the correct one in this yaml file. Now testing can be done locally and the rest is the same as in creating a new device from scratch in Section 6.8.2. The Datalogger node can also be changed here in this same manner if changes are needed to it.

6.9 Working on the backend

The backend is a bit more intricate to develop for, as it relies on a lot of moving parts. Depending on the feature you are working on, you can boot it up in a few different ways. The backend directory is found under:

```
/
└── backend/
```

The backend communicates with the Kubernetes cluster for info about the various devices in it, to boot up and delete existing pods based on the devices registered in the database. It also talks to the InfluxDB database to retrieve users' requested data. To fully test the backend then, the entire cluster needs to be simulated, and this is done through the Skaffold framework using the kind cluster in this example. This is done to more easily boot up temporary clusters that are only important for development purposes. However, to make it easier and faster to develop, locally the backend is usually just run outside the cluster, and there is automatic switching inside the backend that switches how it operates based on an env variable, called *DJANGO_IN_KUBERNETES*. When this is set to 1, it will retrieve the appropriate Kube config, and vice versa when it is set to 0. This Kube config contains information about how to talk to the Kubernetes cluster, and this is different depending on whether it runs within the Kubernetes cluster or outside of it. The example `.env` file is found at:

```
/
└── backend/
    └── .env.prod.example/
```

For local development, debug should be switched to 1. The InfluxDB access token has to be added. Currently, to ease development, the backend always talks to the production InfluxDB database remotely. In this way, an InfluxDB access token has to be added to the `.env` file and exchange the <ACCESS-TOKEN-HERE> with a new token from InfluxDB or one that has already been made. After this, you are ready for local development. All of this is done to make it quicker to develop, as there is no need to load the Django backend into a Kubernetes cluster every time a change is made. It also decouples the backend and the time series database more. An overview of how this works can be seen in Figure 6.9. The backend will be run directly through Python, this will allow it to auto-update when code is changed, and there is no need to build a new Docker container after every change. The database for the backend, PostgreSQL, is hosted within a Docker Compose file and its port is exposed at 5432, allowing the backend to access it as if they were running in the same network. Skaffold boots up the rest of the infrastructure, the local InfluxDB instance, the Datalogger, the MQTT network and various local devices can also be booted up here through the backend, as it connects using the Kubernetes API. For endpoints that request data from the InfluxDB database, it is easier to talk to the production server directly instead of the local one, as it really doesn't matter for development purposes, as it is only retrieving data from there.

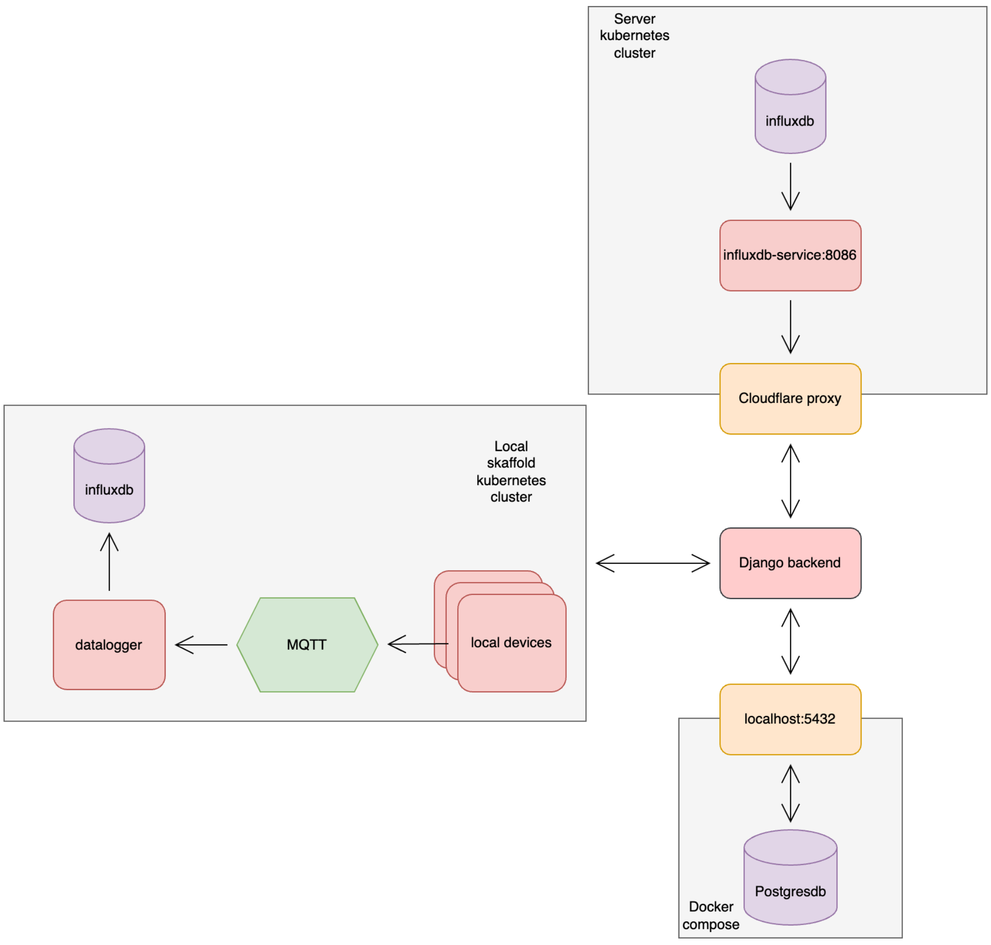


Figure 6.9: Backend development communication overview.

The backend file structure is based on Django modules. Currently, there are the accounts and devices modules, and they have their respective folders within the backend folder. Here the API routes are declared under the `urls.py`, and the functions connected to these routes are defined in the `views.py`. The data structures are defined in `models.py` and serializers for these models are in `serializers.py`.

Other than this the library connecting to the rest of the Kubernetes system is defined under the POWIOT folder. It contains various functions used in talking to the Kubernetes cluster, like booting up new deployments, restarting, getting info and so on.

6.9.1 Starting development

Step-by-step guide for launching everything below.

1. Run the Postgres database

```
$ docker compose -f docker-compose-django-database.yaml
```

2. Create the virtual environment for the backend if not done already

```
$ python3 -m venv venv
$ source venv/bin/activate
$ pip install -r requirements.txt
```

3. Running Django for the first time, or after making database-specific changes run the following to connect and migrate the Postgres database.

```
$ python manage.py makemigrations
$ python manage.py migrate
```

4. Then run the backend server

```
$ python manage.py runserver
```

5. Access to the backend is now available at `localhost:8000`
6. Boot up Skaffold Kubernetes infrastructure if that is needed for development. For instance, if testing booting devices and communicating with the Kubernetes cluster.

```
$ skaffold dev
```

6.9.2 Pushing changes to production

This is a step-by-step guide to pushing these changes into the production server.

1. Create spectacular documentation used by Swagger.

```
$ python manage.py spectacular --file schema.yml
```

2. Build and push the container to docker hub, for information about this see Section 6.11. The current name for the backend container is django-k8s-backend
3. Now access the server as seen in Section 6.5.1.
4. If there were changes made to environment files, the secret attached to the backend needs to be updated, otherwise ignore this step. This step also assumes that there has been made a `.env.prod` file if this is not done look at the example `.env.prod.example` in the same directory.

```
$ kubectl delete secret django-secret
$ kubectl create secret generic django-secret \
--from-env-file=backend/.env.prod
```

5. Update the launch file for the Django deployment. Open the current file in a text editor of your choice on the server, here vim.

```
$ vim k8s/django/deployment.yaml
```

Then change the image of the deployment on the line seen in Figure 6.10. Here it is version 2.0. If the image is now hosted on a different Docker Hub account, *thomasborge* also has to be changed to the appropriate Docker Hub account name.

6. Roll out the changes

```
$ kubectl apply -f k8s/django/deployment.yaml
```

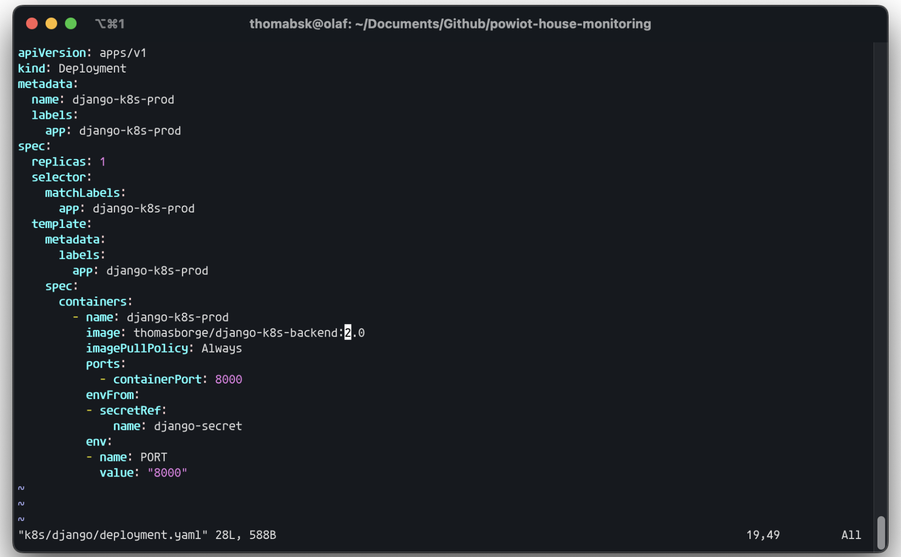
7. Wait for changes to finish.

```
$ kubectl rollout status k8s/django/deployment.yaml
```

8. Migrate the database in case of changes to data structures. The following executes the migrate shell script within the newly created Django container.

```
$ export SINGLE_POD_NAME=$(kubectl get pod -l  \
app=django-k8s-prod -o jsonpath="{.items[0].metadata.name}")
$ kubectl exec -it $SINGLE_POD_NAME -- bash /app/migrate.sh
```

9. The new backend should now be running on `https://powiot.no`.



```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: django-k8s-prod
  labels:
    app: django-k8s-prod
spec:
  replicas: 1
  selector:
    matchLabels:
      app: django-k8s-prod
  template:
    metadata:
      labels:
        app: django-k8s-prod
    spec:
      containers:
        - name: django-k8s-prod
          image: thomasborge/django-k8s-backend:2.0
          imagePullPolicy: Always
          ports:
            - containerPort: 8000
          envFrom:
          - secretRef:
              name: django-secret
          env:
          - name: PORT
            value: "8000"
~
~
~
"k8s/django/deployment.yaml" 28L, 588B                                19,49         All
```

Figure 6.10: Updating deployment file when updating Django backend

6.10 Initial set up of a Production Server

This section outlines the process of establishing a production server from scratch, a procedure designed for a fresh start or a complete overhaul when no operational systems are currently in place.

This guide is tailored for servers running on Ubuntu 22.04.1 LTS [71], but it's likely adaptable to other Ubuntu distributions as well. The setup process is extensive due to multiple interconnected components, but once set up, the system is designed for longevity and isn't intended for frequent replication.

During the setup of our team's supercomputer, the aim was minimal disturbance to the existing system. The permanent operation of the server on this machine isn't ideal due to shared user access, but for the purposes of this thesis, the available hardware necessitated this approach. Consequently, we opted for Microk8s [72], a choice driven by its contained nature and less demanding setup requirements compared to a standard Kubernetes installation. This choice may be re-evaluated if the server setup is migrated to a different machine in the future.

It's important to note that all commands, with the exception of the Microk8s installation section, are presented without the *microk8s* prefix to ensure their universal applicability. If you are using Microk8s on the host machine and this guide issues *kubectl* commands, they should be preceded by *microk8s*. To avoid having to do this, refer to Section 6.10.2 for creating an alias.

6.10.1 Microk8s installation

Run the following command to install the snap package manager if not already installed:

```
$ sudo apt update
$ sudo apt install snapd
```

Then install Microk8s:

```
$ sudo snap install microk8s --classic
```

Configure the firewall to allow pod-to-pod Kubernetes communication:

```
$ sudo ufw allow in on cni0 && sudo ufw allow out on cni0
$ sudo ufw default allow routed
```

To be able to use Microk8s without sudo you need to add yourself to the user group which can be done with the following commands:

```
$ sudo usermod -a -G microk8s $USER
$ sudo chown -f -R $USER ~/.kube
```

Lastly enable the following base addons:

```
$ microk8s enable dns dashboard storage
```

Also add your kube config to your home config directory using the following commands:

```
$ cd $HOME
$ mkdir .kube
$ cd .kube
$ microk8s config > config

$ sudo microk8s kubectl config view --raw > $HOME/.kube/config
```

Microk8s is now installed and running a contained Kubernetes cluster, this cluster can then be started/stopped using the following commands:

```
$ microk8s start
```

```
$ microk8s stop
```

For more info about the Microk8s commands, visit the Microk8s command reference [73].
Now Kubectl commands are accessible as the subcommands of Microk8s. All commands for the rest of this guide will be without the *microk8s* as it is just a framework for running normal Kubectl commands. But if the commands don't work right out of the box, just add *microk8s* to the front of them. The format of using Kubectl with the Microk8s prefix is the following:

```
$ microk8s kubectl <COMMAND>
```

The rest of this guide assumes that the Microk8s alias in Section 6.10.2 has been set up. If that is not done all the same commands can still be run, they just have to be preceded by the $microk8s$ command as seen in the code above.

6.10.2 Setting up a Microk8s alias

If you don't want the hassle of writing *microk8s* in front of every Kubernetes command, and Kubectl is not already installed, you can alias it to do the same thing. To do this open your aliases file in your favourite text editor, here using Vim:

```
$ vim ~/.bash_aliases
```

And append the following to the aliases file.

```
alias kubectl='microk8s kubectl'
```

Then relaunch the terminal and commands should now work by just issuing "kubectl"

6.10.3 Fixing Microk8s restarting pods every day on NTNU server

Since the NTNU server is hosted within its own network, there are some peculiarities that have to be ironed out. One of these is that new connections will make the pod network restart every day as new connections are discovered. To prevent this from happening, as the cluster is never accessing these new VPN connections the following change has to be made.

SSH into the host computer. Open the following file in your favourite text editor, here using Vim.

```
$ vim /var/snap/microk8s/current/args/kube-apiserver
```

Add the following line to this file you just opened.

```
--advertise-address 0.0.0.0
```

Then restart the cluster by issuing the following commands

```
$ microk8s stop
$ microk8s start
```

6.10.4 Starting the main system components

The system consists of several main components.

- Influxdb (Timeseries database)
- MQTT (Message passing system)
- Datalogger (For parsing MQTT data and putting it into the database)
- Cloudflare (For accessing the system without port forwarding)

MQTT

MQTT yaml files can be found in

```
/
└─ k8s/
   └─ mqtt/
      ├─ mqtt-deployment.yaml
      └─ mqtt-service.yaml
```

To start all yaml files in the directory run the following command from root

```
$ kubectl apply -f k8s/mqtt/
```

This will start the MQTT server, hosted locally.

InfluxDB database and Datalogger

The InfluxDB database can be started using the yaml files found under

```
/
└─ k8s/
   └─ logging/
```

To start the InfluxDB node, if it is the first time it is started on a new installation, it needs to be given credentials to startup properly. These are given to the node as secrets. The secret data can be located in:

```
/
└─ k8s/
   └─ logging/
      └─ influxdb-secret-dev.yaml/
```

Exchange the tokens and password for something appropriate.

To start all yaml files in the directory run the following command from root:

```
$ kubectl apply -f k8s/logging/
```

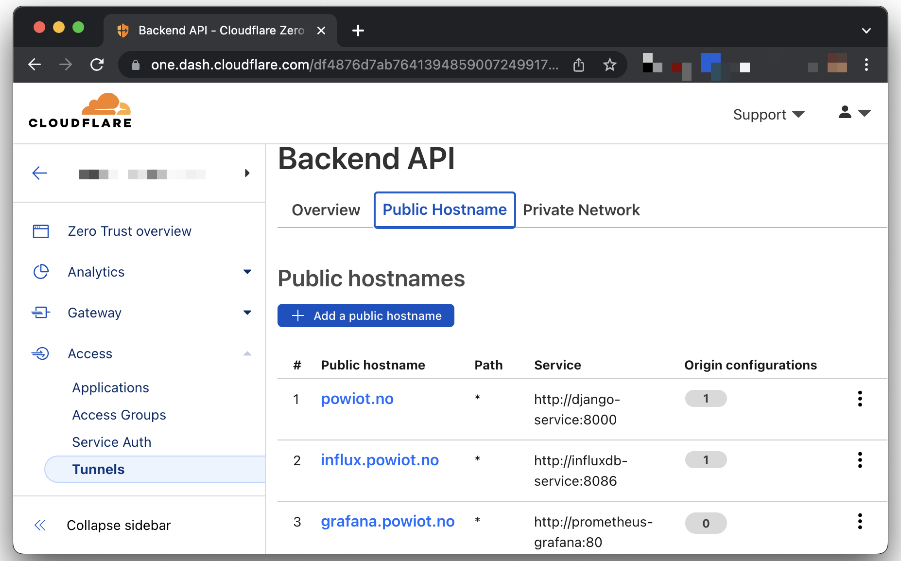
This will start up the InfluxDB database as well as the Datalogger gathering data from MQTT.

Cloudflare

Since the system is running within a network without the ability to port forward, Cloudflare has been used as a reverse proxy. A node is running within the cluster that exposes the relevant servers to the web. This is a free service for smaller projects. First, create or login to a Cloudflare account at `https://cloudflare.com/`. Navigate to domain, and add a new domain. Follow the step-by-step, and when you get to add nameservers, head to your domain provider and add the nameservers to your external nameserver.

A full guide and the guide used to set up this in the first place can be found in a YouTube video [74].

To get the token for the Kubernetes pod it is the same as the Docker token that can be found by selecting Docker. This Docker token can then be inserted into the `cloudflare-tunnel.yaml` file found under the `k8s` folder, and it can then be started. The rest of the services can then be set up within the web interface as seen in [74]. The local Kubernetes endpoints are then used, these correspond to the Kubernetes services that are linked to the various systems. For instance, the InfluxDB is exposed through the influxdb-service at `http://influxdb-service:8086`. An example of this can be seen in Figure 6.11.



Backend API

Public hostnames

#	Public hostname	Path	Service	Origin configurations
1	powiot.no	*	http://django-service:8000	1
2	influx.powiot.no	*	http://influxdb-service:8086	1
3	grafana.powiot.no	*	http://prometheus-grafana:80	0

Figure 6.11: Cloudflare tunnel setup overview.

6.11 Procedure for Building and Deploying Docker Containers

This section outlines the building and pushing of Docker containers. To push Docker containers to the Docker Hub; an account is needed.

6.11.1 Generating Single Architecture Docker Images

If you only need to create single architecture images for development purposes this is the way to do it. This will create images for the operating system you're currently using. Otherwise look in Section 6.11.2 for how to push to all major operating systems, including the Ubuntu server. To build an image go into the respective Docker image source folder, and run the following commands.

```
$ docker login
$ docker build <IMAGE_NAME> .
$ docker tag <IMAGE_NAME> <HUB_NAME>/<IMAGE_NAME>:<VERSION>
$ docker push <HUB_NAME>/<IMAGE_NAME>:<VERSION>
```

6.11.2 Building Multi-Architecture Docker Images for Non-x86 Architectures

If development is done on an arm-based laptop, such as the new M1 and up-based Macs, images have to be cross-compiled for both arm and x86/amd64 architectures. This can easily be done through the Docker interface. To be able to upload any image you need to use a Docker Hub account.

To be able to build for different architectures, have Docker Desktop installed and then simply run the following command in the Docker image source folder. It will build images for arm64 and amd64. It first needs to login to the account and then build and push it by exchanging the tag parameters.

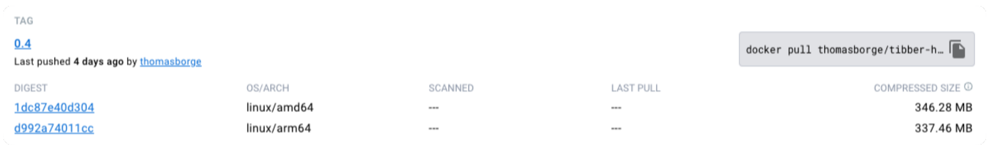
```
$ docker login
$ docker buildx build --push --platform linux/arm64/v8,linux/amd64 \
--tag <DOCKER-HUB-USERNAME>/<IMAGE-NAME>:<VERSION> .
```

If you get an error that the multiple platforms feature is not available, then run the following command and rerun the above command.

```
$ docker buildx create --use
```

Your image will then be uploaded as a multi-architecture build onto the Docker Hub, and the machine using it will automatically download their respective architecture build.

An example of what the correct image will look like on the Docker Hub can be seen in figure 6.12, where two different OSs are listed for the same tag of an image.


TAG

0.4

Last pushed **4 days ago** by thomasborge

```
docker pull thomasborge/tibber-h…
```

DIGEST	OS/ARCH	SCANNED	LAST PULL	COMPRESSED SIZE
1dc87e40d304	linux/amd64	---	---	346.28 MB
d992a74011cc	linux/arm64	---	---	337.46 MB

Figure 6.12: Example of a docker image with 2 different architectures in the same build.

6.12 Troubleshooting Common Errors

6.12.1 System restarting every day with no error

This could be the case if the system is running within the NTNU network. The solution to this is outlined in Section 6.10.3.

6.12.2 Can't SSH into server

If no response is received at all, this could point to the actual server being down, contact the people responsible.

6.12.3 Kubectl command not found

If you are running commands on the server, currently everything is running within Microk8s, and commands have been aliased so that $microk8s\ kubectl$ commands work as $kubectl$ commands. Look in Section 6.10.2 for info about how to fix this.

6.12.4 Dashboard and backend not available at domain

There are two possible reasons for this issue. Firstly, it could be due to a system-wide failure, such as a power outage or a complete system shutdown. Alternatively, the problem might arise from malfunctioning Cloudflare pods. To troubleshoot, ensure that the Cloudflare pods are operational. If they are not running, please refer to Section 6.10.4 for instructions on how to set them up.

Chapter 7

Results

This chapter examines results derived from system tests and software validation. It begins with an analysis of the system's response times during database queries in Section 7.1. This is followed by an evaluation of the system's uptime and resilience in Section 7.2. The topic of fault tolerance is then explored in Section 7.3, succeeded by an account of the outcomes from the control running on the system in Section 7.4. Subsequently, a system evaluation is conducted in Section 7.5, comparing the system against its initial requirements in Section 7.6.

7.1 Response times

An essential aspect of using this system is determining the response times for data querying, which is crucial for subsequent data manipulation. The tests in this section were conducted on a 16 GB RAM, 10 CPU core M1 MacBook Pro. The data was queried over the internet instead of locally to simulate a worst-case scenario for accessing the data. Furthermore, processing nodes will access the database locally through HTTP requests, reducing response times. Cloudflare acting as the reverse proxy may also increase response times further.

Range	Average response time	Standard deviation	Datapoints
30 days	7.534s	0.217s	2 397 905
10 days	2.699s	0.107s	850 289
24 hours	0.349s	0.040s	85 309
10 hours	0.248s	0.059s	35 491
1 hour	0.109s	0.060s	3563
30 minutes	0.053s	0.018s	1782
5 minutes	0.048s	0.016s	298
1 minute	0.050s	0.005s	59

Table 7.1: Response times according to time range for the measurement "power-Consumption" with field "power" for house_id=3. All queries were repeated ten times to gather the data.

Response times were gathered by sending measurement requests to the server remotely from a Python script. The results of this response test can be seen in Table 7.1. The data represents the response times for retrieving the *power* field under the *powerConsumption* measurement for the house with ID 3 with varying time ranges. This is the test house under the DEC. Figure 7.1 presents the response times plotted together. The graph demonstrates a decrease in response times as the time range decreases. The response times appear to level off at around a 30-minute time range, where the amount of data is no longer a limiting factor. This is especially useful in some control schemes where only the most recent measurements are needed, for instance, in Section 7.4.1. The query times will then be negligible compared to the remaining system. In the MPC used in Section 7.4.2, the time ranges of queries varied from 5 minutes to 13 hours. According to Table 7.1, all of these queries will fall under 0.35 seconds.

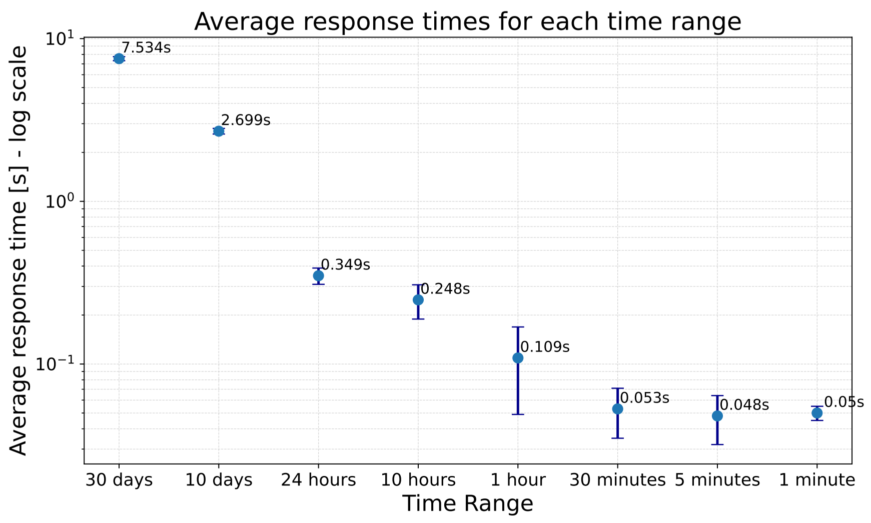


Figure 7.1: Response times with standard deviation based on data in Table 7.1

Figure 7.2 illustrates how the throughput changes with different time ranges, highlighting the presence of overhead in the data retrieval. In most cases, the most recent data will be queried, resulting in fast response times. This differs from the previously developed system, where response times increased throughout the day as it had to fetch the entire day's worth of data each time. This system allows specifying the time frame of the query, resulting in consistent response times regardless of the time of the day the query is made.

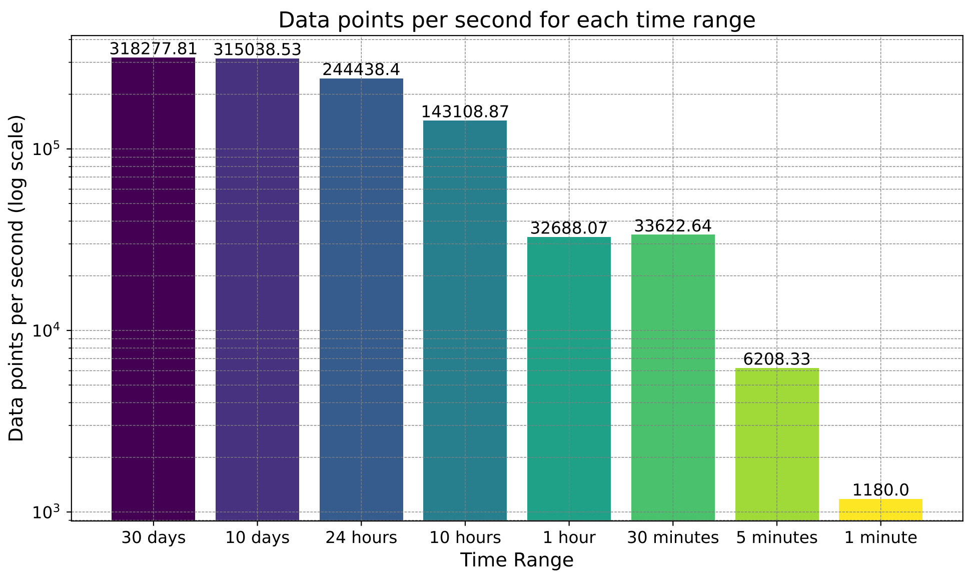


Figure 7.2: Data points per second based on the data in Table 7.1.

7.2 Uptime and resilience

The system has been running continuously for several weeks without requiring any user intervention. Throughout this period, data logging has been consistent as long as the APIs have remained operational. An example showcasing the steady data logging can be observed in Figure 7.3.

Final uptime metrics can be found in Table 7.2, which were calculated by considering the sampling time of the device and any data gaps within the total time range. It's important to note that some devices were undergoing fixes during this period, resulting in lower uptime than expected during the product's development phase. There was also a power outage on the server during this time, however, this is kept in to reflect the actual uptime of the system.

Notably, the Verisure Smoke Detector only provided data for the last 20 days, so its resolution differs from the other devices. However, the standout outlier is the Systemair VSR500 integration, which experienced a significantly high amount of downtime. Upon contacting them, we discovered that the local in-house device frequently experienced outages, and they are currently developing a new unit to address this issue.

As a result of the excessive downtime with the Systemair VSR500 integration, running control on it may not be feasible due to the frequency of interruptions. On the other hand, the MET and Solcast do not rely on physical devices, and their perfect uptime reflects this fact. Since predictions are overwritten as they come closer, they are also much more lenient in their described uptime. They have been down, but predictions were still in the system, just not the newest ones.

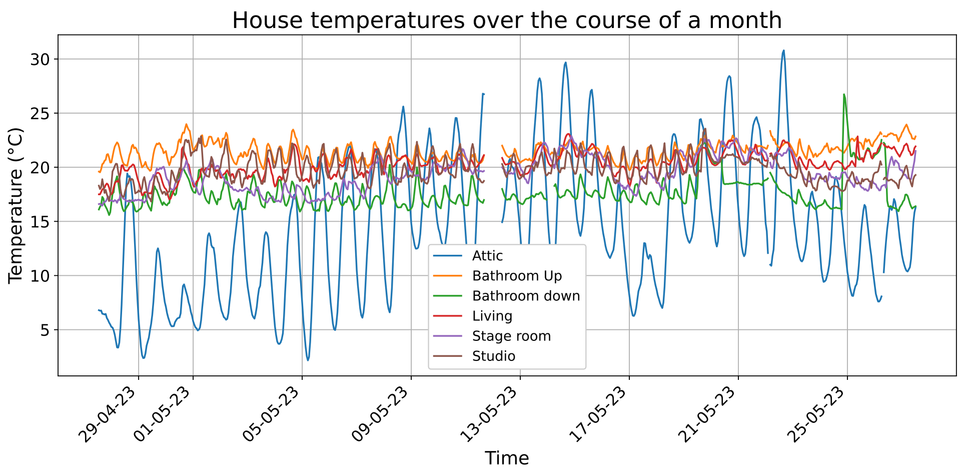


Figure 7.3: A plot showing the logged temperature data from the Mill Sense *airSensor* over the course of a month, also showing a gap in the data at about 12-05-23, which here occurred as the Mill API went down.

Table 7.2: Uptimes of the various device integrations over the last 30 days, except for the Verisure integration which had only run for 20 days. Uptime is calculated by finding gaps in the measurements based on its supposed sampling time and by seeing how much this makes up of the entire period. Some of these results will come from the vendor APIs being down, but some of it also comes from the system being down.

Device	Uptime
Sensibo Sky	93.95%
Systemair VSR500	64.35%
Tibber Pulse	97.80%
Mill-sense	95.28%
MET	100%
Solcast	100%
Victron	95.76%
Verisure Smoke Detector	96.86%

7.3 Fault tolerance

7.3.1 Dealing with Internet Outages

In terms of this system, an internet outage bears a similar effect to an API going offline. Consequently, the pods will crash, repeating the cycle until they can successfully reboot with a restored internet connection. As a result, data logging will naturally resume, given the system's inherent 'fail fast' architecture. This was demonstrated during instances of API downtime, where the pods persistently crashed until they could safely reboot and resume data logging.

The same principle applies to the control loops operating within the system. In the absence of new measurements, the control loop will eventually go offline due to the unreliability of the outdated feedback loop data. It will then remain inactive, awaiting fresh measurements before rebooting. Naturally, this will lead to the control of the respective house going offline, preventing the in-house devices from receiving any inputs until system functionality is restored. Currently, this will lead to the last input being the one that is stored in the heat pump. To avoid this, schedules can instead be sent, so that in the event of an outage, it can follow the previous plan.

7.3.2 Response to Server Reboot

The current system, being hosted within Microk8s, is designed to automatically restart upon a server reboot. The same is applicable when running Kubernetes on bare metal, as it is configured to boot upon server startup. This functionality has been tested several times, with the system consistently restarting as anticipated. Upon reboot, the framework and all logging activity resume operation. The pod lifetimes of which some exceed 80 days at this point, further confirm this automatic restart feature, eliminating the need for manual intervention.

7.3.3 Handling API Downtime

The Systemair integration exemplifies the resilience of the system during API downtime. Over a period of 35 days, the device integration had to restart 3634 times due to recurring API stability issues, primarily owing to some limitations of the installed device. Despite these challenges, the system consistently resumes logging each time the device becomes available and successfully reboots each time it crashes. The system's response to an API going offline largely depends on the respective implementation of the device, a process that is fully modular. As long as the device integration is programmed to crash upon losing connection, the overarching system will manage the recovery process and restore operation. However, during an API downtime, data outages are inevitable, and the system has to wait for the API to become accessible again.

7.3.4 Data Collection Efficacy

The system has been operational for a significant period, progressively gathering an increasing amount of data as more nodes have come online. All data captured has been readily accessible for querying. The influxDB database has accumulated approximately 600 MB of data, which is automatically compressed by the database, facilitating instant access via queries. During the period of this thesis, three houses have contributed to the data collection, each logging data independently. All previously logged data has been transferred to the new system to facilitate a smooth transition. An overview of the devices involved in logging data across the houses is provided in Table 7.3. House 3, the supervisor's residence, has served as the main testing platform for integrating new devices.

Table 7.3: Distribution of devices across houses.

Device/House ID	**3**	**4**	**5**
Mill Sense	✓	✓	
MET	✓	✓	
Tibber RT	✓		
Tibber Hourly	✓		
Solcast	✓	✓	
Sensibo	✓		✓
Systemair VSR500	✓		
Verisure	✓		
Victron		✓	

7.3.5 Autonomous Operation

The system currently exhibits a high degree of autonomy, requiring minimal user intervention. It has successfully logged data continuously for over a month, demonstrating resilience through several power outages and server reboots. Instances of certain integrations failing have been observed, however, these failures were largely attributable to bugs within the integrations themselves. As such, these bugs only affected the corresponding logging pods without disrupting the entire system. Once these bugs were resolved, the affected logging pods resumed normal operation and have been functioning continuously since then.

It should be noted that potential updates to vendor APIs may trigger similar issues. This has been observed in the past year and often necessitates minor code rewrites for re-establishing effective integration. This aspect underlines the importance of some monitoring within the system and to have someone responsible for updating integrations along the way.

7.4 Running control on the system

7.4.1 Use case - Direct response on DEC test house

In the use case outlined in Chapter 5, the server retrieves data from the system and manages control by running a loop that calculates new actuation outputs every five minutes. Actuations are sent only when changes are necessary. Target temperatures are then transmitted to the Sensibo Sky units, adjusting the heat pump settings accordingly. The control parameters used in the test can be found in Table 7.4.

Table 7.4: Control parameter values used in the control use case in Chapter 5.

Parameter	Value
γ	0.7
K_{price}	0.1
K_i	$\frac{1}{24}$
K_p	1
α	0.1

The supervisor, and owner of the house, could manually change the desired temperature settings for his comfort, which were obtained from the backend through an API endpoint. A random day's sample of this operation is illustrated in Figure 7.4. The system clearly adapts to fluctuating spot prices by heating in advance while maintaining the supervisor's preferred temperatures. This is especially evident before 8 in the morning when the heating starts earlier to not be affected by the morning spot prices. The control has functioned autonomously for over two weeks, effectively adjusting to any new settings established by the supervisor.

This system could be easily adapted to new houses by utilizing a Sensibo Sky device, or a similar one, for heat pump actuation and temperature data collection. Additionally, the spot market zone is needed to retrieve relevant pricing information. With these parameters in place, the system can efficiently adjust based on spot market fluctuations.

7.4.2 Use Case - Model Predictive Control (MPC)

A parallel study deployed an advanced Model Predictive Control (MPC) strategy on the same house, demonstrating the system's capacity to support complex control schemes [75]. The data retrieval speed of the system was found to be efficient, with latencies insignificant compared to the computation time of the MPC, with data retrieval taking less than a second. The data needed in this system queried data for up to 13 hours in the future (weather forecast data) and in the past. This demonstrates the system's ability to perform its function without causing unnecessary delays. Moreover, if run locally within the Kubernetes cluster, the system can directly interact with the InfluxDB service.

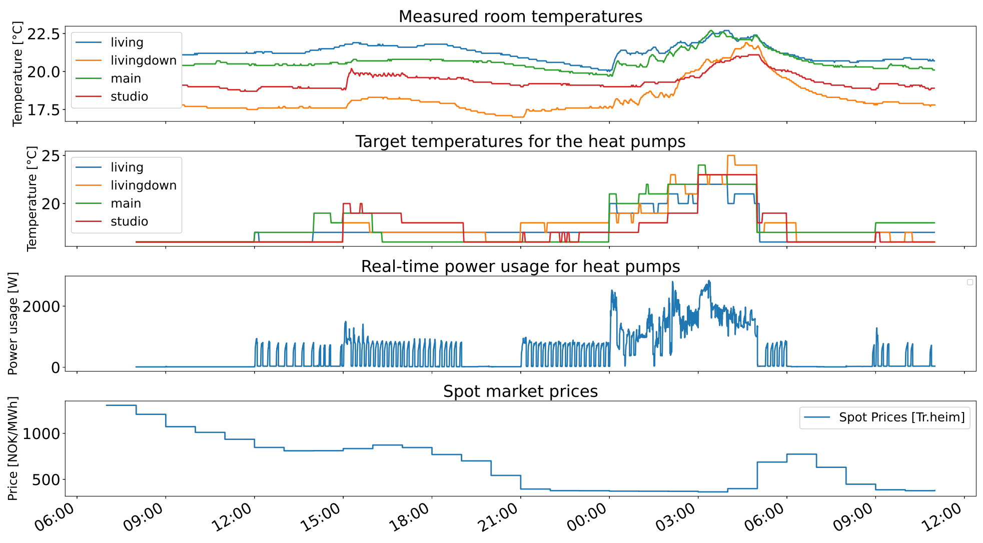


Figure 7.4: This figure demonstrates the operation of the direct response control as it responds to spot market prices. The control manages four heat pumps located in the supervisor's residence, specifically in the living area, lower living area, main room, and studio, each with unique temperature settings. The depicted real-time power usage originates from a tibber-pulse plugged into a smart meter linked exclusively to the heat pumps. The timeline for this plot spans from 6:00 on May 8th, 2023, to 12:00 on May 9th, 2023.

Another aspect that came in handy during weather forecast retrieval for the MPC was interpolating data directly from the database if a certain data structure is needed. No local manipulation was needed. This and several other data manipulation functions can be found in the InfluxDB documentation [76]. An example of what this query looked like in Python code can be seen below:

```
measurement = "weatherForecast"
house_id = "3"
# Querieing the database based on the above information
query = f"""
    import \"interpolate\"
    from(bucket:\"{bucket}\")
    |> range(start: -1d)
    |> filter(fn:(r) => r._measurement == \"{measurement}\")
    |> filter(fn:(r) => r.house_id == \"{house_id}\")
    |> interpolate.linear(every: 5m)
"""
```

7.4.3 Use Case - MPC and Solar Battery Inverter

Another student utilized the system to develop a control algorithm for a solar battery inverter [77]. The control strategy used solar forecast data, weather forecasts, and Chainpro/Victron data stored by the system. This demonstrates the system's utility as a data repository and underscores its potential as a base for further control applications. A Docker container of the control system could be created, making it feasible to execute the entire feedback loop on the server. The lower time constants of this system, compared to a heating solution, demonstrate its potential to operate effectively for systems that require closer to real-time responses.

7.5 Performance Evaluation

We evaluated the performance impact of our system using Prometheus for data collection from the Kubernetes cluster during operation [78]. The server, equipped with an AMD Threadripper PRO 3995WX with 64 cores and 220 GB of RAM, hosts the code execution. The rest of the specifications are in Table 7.5.

Table 7.5: Server Specifications.

Component	**Specification**
Case	FRACTAL DESIGN Define 7 XL BK TGL
Power Supply Unit	BE QUIET! PSU Dark Power Pro 12 1500W
SSD	SAMSUNG SSD 980 PRO 2TB M.2 NVMe PCIe 4.0
Motherboard	ASUS PRO WS WRX80E-SAGE
Processor	AMD Threadripper PRO 3995WX
Cooler	BE QUIET! Be Quiet_ Dark Rock Pro AMD TR4
RAM	KINGSTON 32GB 3200MHz DDR4 ECC CL22 (7 units)
Graphics Card	ASUSGF TUF-RTX3080TI-O12G GAMING 12GB GDDR5

The fundamental infrastructure's CPU and memory utilization, depicted in Figure 7.5 and Figure 7.7 respectively, remain relatively constant due to the consistent overhead of running only a single instance of this software. The device-specific pods' CPU and memory consumption are illustrated in Figure 7.6 and Figure 7.8. The CPU metrics are quantified in Kubernetes CPU units, where 1 CPU unit corresponds to 1 vCPU/Core or 1 hyper-thread on Intel processors [79]. This correspondence suggests that this server could theoretically operate up to 64 of these CPU units, and 128 if we are counting hyper-threading. An interesting observation is the lack of correlation between the pod sampling time and CPU usage, with pods exhibiting performance spikes and then returning back to zero usage. This could be due to some requests taking more CPU usage than normal, for instance, when requests take time to go through.

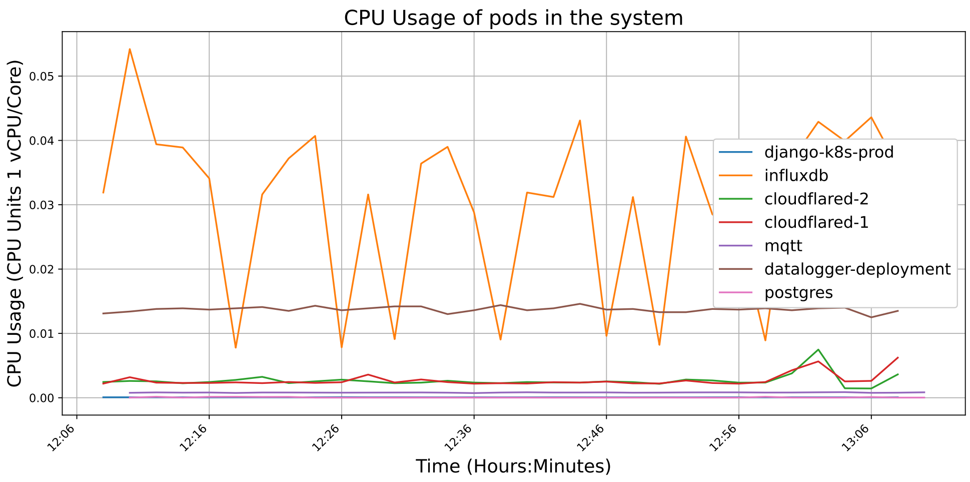


Figure 7.5: CPU usage of the base infrastructure running on the server with an AMD Threadripper PRO 3995WX.

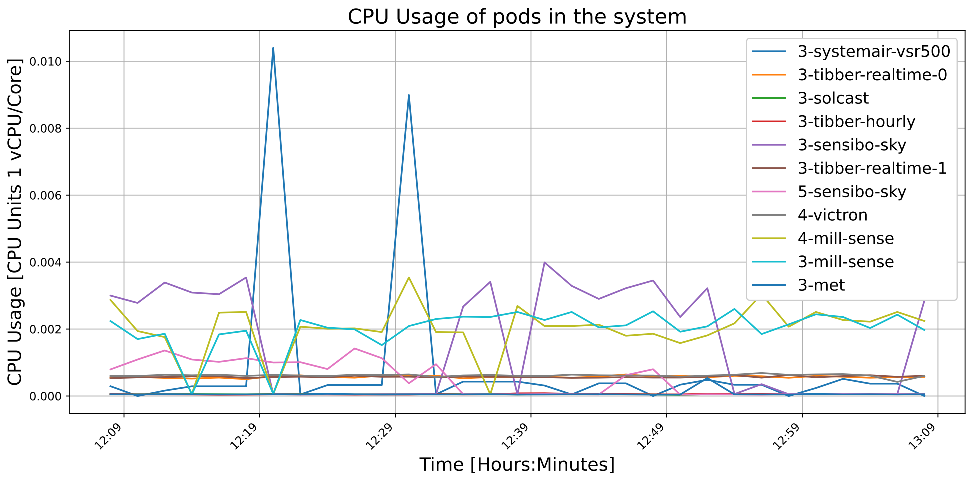


Figure 7.6: CPU usage of the device pods running on the server with an AMD Threadripper PRO 3995WX.

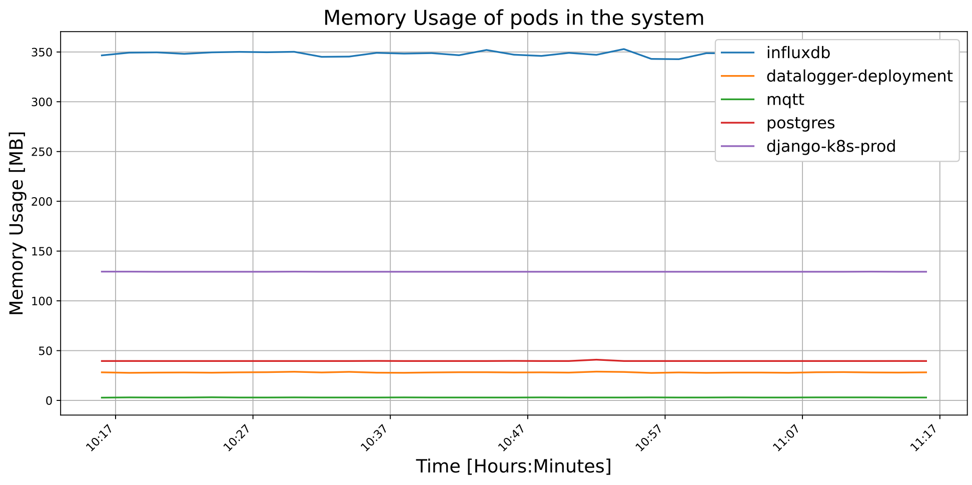


Figure 7.7: The memory usage for the different base infrastructure pods running in the system, running on the server with an AMD Threadripper PRO 3995WX.

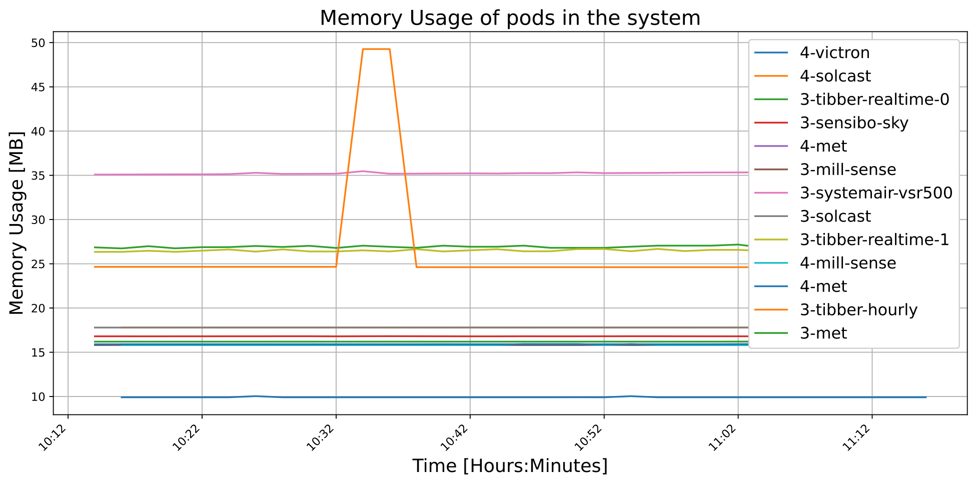


Figure 7.8: The memory usage for the different device pods currently running in the system, running on the server with an AMD Threadripper PRO 3995WX.

7.6 Evaluating against requirements

This section presents an evaluation based on the criteria initially specified in Chapter 3. It aims to demonstrate the current capabilities of the system while also identifying existing gaps in functionality.

7.6.1 Data logging

An overview of the data logging evaluation can be found in Table 7.6.

Table 7.6: Data logging requirements evaluation.

Requirement	Description	Result
DLR-1	High available resolution of measurements.	Fully achieved
DLR-2	High throughput.	Fully achieved
DLR-3.1	Automatic restart on crashes.	Fully achieved
DLR-3.2	Credentials saved through crash/restart..	Fully achieved
DLR-3.3	Reconnection after internet failure.	Fully achieved
DLR-4	Persistent storage of data for long-term access and analysis.	Fully achieved
DLR-5	Automatic backup and recovery in case of data loss.	Partly
DLR-6	Scalable architecture for handling large amounts of data and users.	Fully achieved
DLR-7.1	Compliance with relevant data privacy regulations (e.g. GDPR).	Partly
DLR-7.2	Robust data encryption and protection against unauthorized access.	Partly

DLR-1: High available resolution of measurements

- *Evaluation*: The employed InfluxDB database supports nanosecond precision, providing sufficient granularity for any smart house scenario.

DLR-2: High throughput

- *Evaluation*: InfluxDB is designed for high throughput, offering solid performance given sufficient hardware resources.

DLR-3: Resilient against system crashes and restarts

- **DLR-3.1: Automatic restart on crashes**
  - *Evaluation*: System crashes triggered a full restart, including the reinitialization of all device nodes.

- **DLR-3.2: Credentials saved through crash/restart**
  - *Evaluation*: After a server reboot or crash, all devices were able to successfully login again.
- **DLR-3.3: Reconnection after internet failure**
  - *Evaluation*: While simulating an internet failure is challenging on the NTNU Eduroam internet, the system was tested with devices going offline and successfully came back online once connectivity was restored.

DLR-4: Persistent storage of data for long-term access and analysis

- *Evaluation*: All data is stored and compressed on the server. This dataset now spans over 3 years for the supervisor's house, as data was migrated from the previous system.

DLR-5: Automatic backup and recovery in case of data loss

- *Evaluation*: At present, there is no automatic backup system in place. However, a simple script can be run to backup data to user laptops. The team is awaiting NTNU's provision of a secondary server for data backup, preferably located in a different geographical area.

DLR-6: Scalable architecture for handling large amounts of data and users

- *Evaluation*: InfluxDB supports the handling of large data volumes, and the backend, powered by the PostgreSQL database for user data, should manage any feasible user load.

DLR-7: Security and Compliance

- **DLR-7.1: Compliance with relevant data privacy regulations (e.g., GDPR)**
  - *Evaluation*: Users can retrieve their data through the backend API. However, the system does not yet support complete data deletion.
- **DLR-7.2: Robust data encryption and protection against unauthorized access**
  - *Evaluation*: Sensitive data is securely stored within Kubernetes secrets, while user passwords are protected through Django's hashing and salting mechanisms. Nonetheless, it is important to acknowledge that a proficient attacker with server access might still be capable of extracting this information. Recognizing the inherent nature of security as an ongoing concern, addressing this issue will necessitate consistent maintenance efforts.

7.6.2 Backend

An overview of the backend evaluation can be found in Table 7.7.

Table 7.7: Backend requirements evaluation.

Requirement	Description	Result
BR-1.1	API access for developers to build custom applications and services.	Fully achieved
BR-1.2	Standard REST API to support multiple data formats.	Fully achieved
BR-2.1	Independence of the backend from new integrations.	Fully achieved
BR-2.2	Template for creating new integrations.	Fully achieved
BR-2.3	Provide support for custom credential inputs.	Fully achieved
BR-3.1	Enable users to start devices based on their home setup.	Fully achieved
BR-3.2	Enable users to check the status of their devices.	Fully achieved
BR-3.3	Enable users to stop devices on demand.	Fully achieved
BR-3.4	Enable users to restart their devices on demand.	Fully achieved
BR-3.5	Enable users to retrieve their own data.	Fully achieved
BR-3.6	Notify users if their device is not working properly.	Partly
BR-3.7	Provide different tiers of use with configurable parameters for each tier.	Partly
BR-4.1	Encrypt secret data.	Partly
BR-4.2	Enable user authentication.	Fully achieved

BR-1: Integration with other systems

- **BR-1.1: API access for developers to build custom applications and services.**
  - *Evaluation*: Comprehensive API documentation is available to facilitate developer use.
- **BR-1.2: Standard REST API to support multiple data formats.**
  - *Evaluation*: Django, the system's framework, has built-in support for multiple data formats, promoting easy and seamless data exchange.

BR-2: New device integrations

- **BR-2.1: Independence of the backend from new integrations.**

  - *Evaluation*: The backend, which primarily concerns itself with Docker Hub images and credential passing, is fully compatible with any type of integration.

- **BR-2.2: Template for creating new integrations.**
  - *Evaluation*: A device creation template is available, simplifying the integration process.
- **BR-2.3: Support for custom credential inputs.**
  - *Evaluation*: The system can accommodate any number of text or number inputs, allowing for the integration of various device credentials.

BR-3: User features

- **BR-3.1: Enable users to start devices based on their home setup.**
  - *Evaluation*: A list of currently supported devices is provided. If users possess one of these devices, they can easily initiate the operation.
- **BR-3.2: Enable users to check the status of their devices.**
  - *Evaluation*: The backend API provides users with access to their data and the current state of their logging pod.
- **BR-3.3: Enable users to stop devices on demand.**
  - *Evaluation*: An endpoint is available for users to halt their logging pod operation.
- **BR-3.4: Enable users to restart their devices on demand.**
  - *Evaluation*: An endpoint is available for users to restart their logging pod.
- **BR-3.5: Enable users to retrieve their own data.**
  - *Evaluation*: Data can be accessed by users through the data endpoint.
- **BR-3.6: Notify users if their device is malfunctioning.**
  - *Evaluation*: While email notifications are not yet implemented, users can access the status of their device through a dedicated endpoint.
- **BR-3.7: Provide different tiers of use with configurable parameters for each tier.**
  - *Evaluation*: Currently, users can log their devices and choose the ones to include. Control functionality testing will precede the user rollout of this feature.

BR-4: Security

- **BR-4.1: Encrypt secret data.**

  - *Evaluation*: The system presently encodes but does not encrypt data at rest. Future work should implement encryption. However, with a server and internal cluster breach, an attacker could access secret credentials regardless of encryption. This risk underscores the importance of system access security.

- **BR-4.2: Enable user authentication.**
  - *Evaluation*: Registered users can log in/out and access their own houses and devices.

7.6.3 Processing

An overview of the processing evaluation can be found in Table 7.8.

Table 7.8: Processing requirements evaluation.

Requirement	Description	Result
PR-1.1	Resilient against network connectivity issues.	Fully achieved
PR-1.2	Capability to send commands to various actuators.	Partly
PR-2.1	Optional customisation of control algorithms.	Partly
PR-2.2	Variety of supported devices for control.	Partly

PR-1: Reliability

- **PR-1.1: Resilience against network connectivity issues.**
  - *Evaluation*: The system remains inactive during network issues, allowing users to manually control their home devices. This attribute makes the system resilient to network connectivity problems.
- **PR-1.2: Capability to send commands to various actuators.**
  - *Evaluation*: Actuation functions have been implemented for Sensibo and the VSR500 ventilation system. Further development is needed to expand actuation capabilities to other devices.

PR-2: Configuration

- **PR-2.1: Optional customization of control algorithms.**
  - *Evaluation*: As a proof of concept, different schedules can be added in the backend, which the control will retrieve. Users can adjust desired temperatures for each hour of the day, with different settings for weekdays and weekends. Additionally, the PI controller parameters and gain for the spot market offset can also be adjusted.

- **PR-2.2: Variety of supported devices for control.**
  - *Evaluation*: Currently, Sensibo and the VSR500 ventilation system are the only supported devices for control. Expansion to include other devices is necessary, and investment in these devices for integration purposes is recommended.

Chapter 8

Discussion

This chapter discusses several key topics. It commences with an evaluation of the system, as detailed in Section 8.1, followed by a discussion on the system's capacity in Section 8.2. Subsequently, user retention will be explored in Section 8.3 and provide an overview of our software selection in Section 8.4. The implications of the GDPR on the system are then analyzed in Section 8.5, before examining how system downtime is managed in Section 8.6. The project's contributions towards the United Nations Sustainability goals will be discussed in Section 8.7. The chapter concludes with a discourse on ontological considerations in Section 8.8.

8.1 System Evaluation

The system successfully fulfilled all the components detailed in Section 1.3. It evolved into a comprehensive infrastructure adept at both gathering and processing data. During the course of this thesis, the system accumulated data from three distinct residences. Numerous individuals utilized the system's data for their unique analytical purposes. In addition, a backend was developed with the necessary endpoints for users to activate and access devices in their homes. A final achievement was demonstrating a proof-of-concept scenario where the server controlled the heating pumps in a house, effectively closing the feedback loop. Below are the highlights of the system's strong points and potential areas for improvement.

Strengths of the system:

- Stable data logging system.
- Custom backend API enables users to add new devices for their homes.
- Backend API also lets users view device status and retrieve data.
- System framework supports expansion, allowing processing nodes to be booted within the Kubernetes cluster for local data retrieval or credential access.
- Successful use by research group members for data retrieval and control loop completion.

- Scalability to accommodate at least 1100 additional homes, as will be described in Section 8.2.
- Docker containers facilitate isolated development of new device integrations.
- Local development environments mimic production servers for ease of development.
- Comprehensive documentation.

Areas for Improvement:

- Certain software choices, like the time series database, may need reevaluation as querying proved to be a bit complex.
- Implementation of automatic backups and a designated location for them is necessary.
- A more streamlined method for sending control to devices on a per-house/device basis needs to be developed.
- Aspects of the system may be overly complex due to the project's broad scope, indicating areas for potential simplification by future developers.

8.2 System Capacity: How many houses/devices can it handle?

Estimating the number of houses our system can support on the current server is somewhat theoretical. Still, an educated guess can be made based on the available CPU and memory usage of the currently running pods. Assuming we reserve half of the server's CPU and memory for control and other tasks, we have about 110 GB of RAM and 32 CPU cores available (∼32 Kubernetes CPU units [79]).

From Figure 7.6, we observe that the average CPU usage is below 0.002 units, suggesting a potential capacity of 16,000 devices. However, considering the average memory usage of about 20 MB per pod from Figure 7.8, the memory capacity is limited to 5,500 devices. Thus, the limiting factor is the RAM.

Assuming an average of 5 devices per house, the system could feasibly support about 1,100 additional houses, leaving ample capacity for running control algorithms alongside these logging devices, as only half of the current machine's resources are being utilized.

Storage requirements are primarily determined by the volume of data each house generates. With an estimated requirement of 500 MB of storage per house over a two-year period, 1,100 houses would require about 550 GB of storage over two years, a negligible amount compared to the memory and CPU usage required by the system.

The system, built on the Kubernetes framework, inherently supports horizontal scalability, theoretically extending its capacity almost indefinitely. While all system components can be scaled up, such a setup would require a certain level of expertise and commitment to maintenance, particularly if scaled across multiple servers.

In practical terms, hosting everything on a single server might be most efficient, focusing on module optimization to support as many houses as possible. Python uses a significant amount of memory as it stores most of its variables on the heap, so writing these integrations in a faster language could significantly increase the number of devices the system can handle. The immediate challenge, however, might not be technical optimization but rather encouraging user adoption of the system.

8.3 User retainment

One of the compelling inquiries in the design of this system is whether users would find it valuable enough not just to use it initially but continue to use it in the long run. A significant selling point is the practical benefit of saving money at no cost other than data, which inherently enhances user retention. The creation of a high-quality product naturally encourages increased usage. This is further complemented by the system's "set-it-and-forget-it" style, which requires minimal user intervention. However, it might be worth exploring additional strategies to sustain user engagement.

Integrating elements of gamification could serve as one such strategy. Once a substantial user base is established, comparisons between users' electricity consumption patterns could be facilitated. Questions like when they use electricity, how efficiently they use it when costs are low, and how they fare against other users could be posed. Such a competitive aspect may incentivize electricity use at certain times of the day. Furthermore, this data could be regionally aggregated, offering insights about which areas are most adept at this. Consequently, friendly competition with a low entry barrier could be instituted.

Another compelling feature could be the provision of interesting insights derived from their personal usage data. For example, users could view how their usage correlates with outdoor temperatures, thereby gauging their house's heat retention capacity. This could help users ascertain whether a recent renovation has resulted in lower energy bills. Additionally, this data could be utilized to estimate a house's energy rating based on its heating response. This information could be intriguing and useful for users, contributing to user retention.

8.4 Software Selection

The selection of software libraries plays a significant role in the design of this project, influencing its usability, scalability, and maintainability. Choices such as InfluxDB for time series data management and Django for the backend framework have their respective merits but also have potential drawbacks.

For instance, some of these tools may be more advanced than necessary for this type of project, increasing the technical threshold for developers who work on the system. With new developers joining the project annually, the system's complexity could hinder swift and efficient development. In this context, the system's learnability and accessibility could be just as important as the advanced features of the software used.

Django was selected as the backend solution, and it proved to be relatively easy to set up despite its considerable overhead. Given that Django is utilized in production environments by various large companies, investing time in learning the framework is more advantageous than developing a custom setup from scratch. While a custom approach might have required fewer lines of code, standardizing the system is likely more beneficial for future developers working with the code.

One key strength of the current system is its modularity. If Django proves unsuitable for the system in the long run, it can be replaced with a different backend since the POWIOT Kubernetes module is not Django-dependent. It can be ported to a new project, necessitating only the backend components' rewriting. This approach ensures that the system isn't locked into its initial software choices, although significant changes would require time for re-implementation.

The choice of InfluxDB for data storage also presents an interesting trade-off. On the one hand, its performance and efficient storage of data points are clear advantages. On the other hand, its usability proved somewhat complex. Although manageable, this complexity might impact future usage. Creating effective wrappers could alleviate this issue, but alternatively, a simpler MySQL database might suffice for time series data if the system doesn't scale significantly. SQL is not well-suited for large-scale time series data, but this system may not reach that kind of scale in the first place. As such, it might be worthwhile to reassess the choice of database. Nevertheless, the data is currently centralized and easily retrievable, simplifying potential migration.

The decision to implement devices via Docker containers has imparted considerable flexibility to the system. Each device is isolated, ensuring that code for a new device can't interfere with any existing devices, as well as keeping credentials for different devices completely separate. Additionally, developers can choose to write code in the programming language they're most comfortable with, enhancing productivity and code quality. While this may lead to potential dips in performance compared to all-in-one systems, the emphasis here is on the ease of creating new integrations.

8.5 GDPR Compliance and Personal Data Storage

The crux of this project involves handling a considerable amount of sensitive user data, including data from devices accessed by the system and user credentials for these devices. Consequently, adhering to the General Data Protection Regulation (GDPR) is paramount to assure users of the safety and responsible use of their data for research and operational purposes only.

8.5.1 Rationale for Data Storage

A primary stipulation of the GDPR pertains to the rationale for data storage [80]. This stipulation requires a justifiable reason for data storage. For our system, the justification is straightforward: data is stored for research and to provide users with enhanced control algorithms. Only data pertinent to these goals are stored. Credentials for accessing devices are retained only as long as the devices remain active. If users remove devices from the system, they will need to re-input credentials, ensuring only necessary data is stored. The subsequent data processing aligns with the original purpose for data storage [81]. Therefore, data collected to enhance user algorithms cannot be repurposed, such as selling for different uses, without user consent.

8.5.2 User Data Rights

A crucial aspect of GDPR is ensuring user control over personal data [82]. It is not sufficient to use data responsibly; users must also have insight into their data and the capacity to delete all their data upon request. Thus, the system should have the infrastructure to delete all a user's data points and compile all data into a downloadable file for user review upon request. While these services must be free, this does not present significant challenges as the system is research-oriented and thus free for end users. Interestingly, the GDPR does not mandate data deletion for scientific research purposes even upon user request [83]. However, ethical considerations should guide this process, and we believe user data should be deleted if the user no longer wishes to retain it in the system.

8.5.3 Nature of Data Stored

One advantage of our system is that we only store data already retained by device vendors. Since the data we accumulate is also stored by the vendors' APIs, we merely aggregate it. However, this aggregation creates more personally identifiable data, highlighting the importance of GDPR compliance. Therefore, we should avoid storing superfluous data that could impose unnecessary risks to users. For example, precise location data may not be needed if regional data suffices for control algorithms, thereby reducing the amount of traceable user data.

8.5.4 Key Takeaways for GDPR Compliance and This System

Based on the aforementioned points, here are some critical guidelines for this project to ensure GDPR compliance:

- Secure user consent before any data collection.
- Establish a valid purpose for data storage. If the data does not contribute to research or control algorithms, consider if its storage is necessary.
- Ensure that the use of data remains consistent with the initial project scope. The project's objective is to provide users with better control algorithms

for their smart appliances, which should be the primary reason for data collection.
- Enable the system to provide all personal data to users and offer the option to delete all personal data.
- Minimize the collection of personally identifiable data, such as location or personal details.

8.6 Managing Downtime in a Continuously Online Smart-home System

A critical aspect to consider in this system is its reliance on continuous online connectivity. If either the server or individual vendor APIs experience downtime, the system's functionality could be significantly disrupted. For example, if a vendor's API becomes temporarily unavailable, access to the corresponding device will be interrupted, and no data logging or control commands will be possible until the API service resumes. Similarly, unexpected API updates by a vendor could temporarily hinder device access. It is essential to acknowledge these potential scenarios as inevitabilities in the system design.

The system is engineered to prioritize user-friendliness and ease of installation, which necessitates certain compromises compared to a wholly local system. For instance, a persistent internet connection is mandatory for the system to operate effectively. The same requirement applies to individual smart devices that users integrate into the system, such as the Sensibo heat pump controller, which relies on an active internet connection. If a network outage occurs, users can still manually control devices using their respective remote controls.

A key strength of this system lies in its compatibility with multiple existing solutions. This interoperability provides users with alternate control options in the event of system downtime—a notable advantage over more monolithic systems, where technician intervention may be needed to rectify malfunctions. Additionally, diversifying device vendors within the system can reduce the probability of complete system downtime. However, should a complete internet outage occur, all smart devices would become non-functional—an unavoidable limitation of a system of this nature.

During an internet outage, the system reverts to a standard house setup, with all devices available for manual control. The impact on user experience under these circumstances is most likely relatively minimal. The crucial factor here is timely user notification of the downtime to maintain awareness of the system status. However, some control could still be maintained, if schedules are sent to the actuators in question. If a heat pump for instance receives the schedule for the MPC and the system goes down, the heat pump can continue on this schedule until another input is sent over. Based on the uptime found in Table 7.2, the devices are not often down for long, and would probably result in downtime that users might not even notice, as the schedules continue running.

Power outage

An illustrative example of this occurred during the semester when a power outage at NTNU temporarily disabled the server. Although the server successfully rebooted with all credentials intact once power was restored, such incidents underscore the inevitability of downtime events. To mitigate this, introducing redundancy into the system is crucial.

One potential solution involves physically adding another server in a geographically distinct location. To be more specific, this location has to be in a different power and communications network, to fully ensure redundancy. This secondary server could serve as an additional Kubernetes cluster linked to the primary one, ready to share the workload in the event of an outage. However, this approach introduces complexities around user credential storage. Should credentials be replicated across all servers to enhance redundancy, or would this approach pose an unacceptable security risk? The decision is not straightforward and would benefit from rigorous testing. In systems like this, it might sometimes be prudent to sacrifice some reliability in favour of enhanced security.

8.7 A step towards achieving the UN Sustainability goals

In this section, we discuss the contribution of our scalable system for smart-home IoT devices to the United Nations Sustainable Development Goals (SDGs). Specifically, we identify four SDGs that our system primarily addresses:

8.7.1 SDG 7: Affordable and clean energy

Our system actively supports SDG 7 by optimizing energy consumption through the integration of smart IoT devices in residential settings. By intelligently managing energy usage, our system effectively increases overall energy efficiency and contributes to the broader goal of providing access to affordable, reliable, sustainable, and modern energy for all. As a result, the system reduces the demand for energy, minimizing reliance on fossil fuels and promoting the adoption of cleaner, more sustainable energy sources.

8.7.2 SDG 11: Sustainable Cities and Communities

Our system plays a vital role in fostering sustainable cities and communities (SDG 11) by enhancing resource efficiency, reducing energy consumption, and improving the overall quality of life in urban areas. With the potential for our system to enable houses to collaborate in the future, we expect to see increased resilience within communities and the development of a more sustainable built environment.

8.7.3 SDG 12: Responsible Consumption and Production

By utilizing open APIs and sensor data to intelligently control smart pumps and ventilation systems, our system actively encourages responsible consumption and production patterns (SDG 12). It does so by minimizing waste and promoting more efficient resource utilization, which aligns with the broader objective of achieving sustainable consumption and production patterns. Using existing devices, rather than requiring new installations, promotes responsible consumption and production.

8.7.4 SDG 13: Climate action

Our system's contribution to reducing energy consumption and promoting the use of clean energy sources directly impacts SDG 13, which focuses on climate action. By enhancing energy efficiency and diminishing the reliance on fossil fuels, our system plays a role in combating climate change and mitigating its impacts. Furthermore, as our system evolves to enable houses to work together, its potential contribution to large-scale climate mitigation efforts could become increasingly significant.

8.8 Ontological Considerations in Device Correlation and Data Naming

A significant challenge that surfaces within this system is the issue of ontology—the classification and organization of information. In this case, it pertains to how different devices, such as a varying number of sensors and actuators within a house, are connected and correlated.

The problem at hand can be articulated as follows: if a house is equipped with multiple climate sensors and heat pumps distributed across several rooms, how should these devices be linked? Furthermore, how should their data be correlated to form meaningful insights?

One potential solution to address this problem involves utilizing a graphical interface. Users can employ this interface to group devices based on their physical location, thereby establishing connections between sensors (devices that collect data about the environment) and actuators (devices that perform actions based on the sensor data). In this context, the sensors can be linked to the actuators according to their respective locations.

The system can further enhance this user-defined grouping by employing data analysis to verify the connections. If data patterns suggest that a particular sensor and actuator are not interacting as expected (for example, if a heat pump is commanded to alter the temperature, but the associated sensor registers no change), the system could alert the user to a potential mismatch.

The system also faces challenges in the nomenclature of database groupings and individual measurement fields. This variation in naming conventions for general measurements might lead to complications in the future as more devices are added, with various different measurements available. Hence, it is crucial to strive for measurement names that are as general and reusable as possible, aiding subsequent development. If a more appropriate or universal naming scheme is identified in the future, changing the current names is an option, albeit one that would necessitate some manual management.

Chapter 9

Conclusion

This thesis has explored methods of creating a more scalable data retrieval system on a larger scale, utilizing off-the-shelf IoT devices with an internet connection. The collected data is stored in a central location that is readily accessible, facilitating more informed decision-making processes and providing opportunities to complete control loops. Furthermore, the system serves as a hub for data science, enabling users to glean more information from the data they receive.

In practice, the system can accumulate data from numerous houses, with the capacity to seamlessly add more. The data becomes immediately available and can be retrieved either remotely for research and development purposes or via a processing node operating on the server for a production setup. The system has demonstrated significant fault tolerance, reducing the need for user intervention. This is particularly beneficial as it is unrealistic to expect someone to monitor its functioning consistently. Thus, the system becomes a valuable tool for various research tasks within the group, delivering data in an easily digestible format. It was also shown in practice that a control loop could be implemented by retrieving the logged data from the system, doing calculations on it and sending inputs back into these IoT devices.

However, the system does have certain limitations, primarily its complexity. Given its broad functionality, a degree of knowledge is necessary to operate and develop it further. There is a risk of the system being discontinued as new students seek to engage with it. To mitigate this issue, substantial effort has been devoted to creating a comprehensive user guide, as documented in Chapter 6. The hope is that new tools can be integrated into the existing system, extending its lifespan and usability. The system's modularity is a noteworthy advantage in this regard, as it allows for replacing components that do not meet requirements while preserving the rest of the system.

9.1 Future Work

The system presented in this thesis, which took inspiration from a previous master's project [1], could be further refined by subsequent contributors. The system is designed with modularity and scalability in mind, paving the way for future developers to adapt and expand upon it. The following aspects present possibilities for future enhancements and developments of this system.

Device Integrations and Control Mechanisms:

- Increase the number of device integrations and verify their logging capabilities to broaden system adaptability.
- Design a universal method for actuator control to facilitate the use of control algorithms across a wide set of actuators. This could, for instance, be a universal method of controlling heating systems.

User Experience Improvements:

- Create a frontend to make the system usable for normal users.
- Let users map room sensors in their homes and associate them with corresponding heat pumps to simplify system setup and increase intuitiveness.

System Performance and Security:

- Assess and optimize existing device integrations for performance, potentially rewriting them in a less memory-intensive language.
- Emphasize system security by implementing encryption at rest to protect sensitive credentials.
- Strengthen Kubernetes access controls for enhanced system security.

Compliance and User Management:

- Achieve full GDPR compliance by integrating a "Retrieve all my data" feature.
- As it stands, new users must be added by an admin to prevent bot infiltration and unauthorized access during this development phase. Therefore, creating an interface for user registration is a crucial next step. Including sending emails to confirm users and sending crucial information about the state of their devices.

Data Analysis:

- Investigate data analysis capabilities to provide system users with valuable insights and identify meaningful trends in user data. This could be to estimate the heating dynamics of individual homes to provide additional value to the system and its users.

Bibliography

[1] E. Törn, 'Iot software for smart houses: Managing data collection and connectivity between iot devices for an advanced heating control algorithm,' M.S. thesis, 2021.

[2] T. B. Skøien, 'Iot software for integration of smart home devices,' 2022.

[3] 'Json.' (2023), [Online]. Available: https://snl.no/JSON (visited on 31/05/2023).

[4] 'Mqtt introduction.' (2023), [Online]. Available: https://www.hivemq.com/blog/mqtt-essentials-part-1-introducing-mqtt/ (visited on 29/05/2023).

[5] 'Kubernetes namespace.' (2023), [Online]. Available: https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/ (visited on 31/05/2023).

[6] 'What is an orm.' (2023), [Online]. Available: https://www.freecodecamp.org/news/what-is-an-orm-the-meaning-of-object-relational-mapping-database-tools/ (visited on 31/05/2023).

[7] 'Proxy.' (2023), [Online]. Available: https://snl.no/proxy (visited on 31/05/2023).

[8] 'Xml.' (2023), [Online]. Available: https://snl.no/XML_-_IT (visited on 31/05/2023).

[9] 'Smart meters.' (2023), [Online]. Available: https://www.sintef.no/en/expertise/sintef-energy-research/advanced-metering-and-control-systems/ (visited on 31/05/2023).

[10] 'Fornybar energi.' (2020), [Online]. Available: https://www.fn.no/Statistikk/fornybar-energi (visited on 31/05/2023).

[11] 'Energy use by sector.' (2020), [Online]. Available: https://energifaktanorge.no/en/norsk-energibruk/energibruken-i-ulike-sektorer/ (visited on 12/03/2023).

[12] C. Heinrich, C. Ziras, A. L. Syrri and H. W. Bindner, 'EcoGrid 2.0: A large-scale field trial of a local flexibility market,' *Applied Energy*, vol. 261, p. 114 399, Mar. 2020. DOI: 10.1016/j.apenergy.2019.114399. [Online]. Available: https://doi.org/10.1016/j.apenergy.2019.114399.

[13] 'Sensibo sky.' (2023), [Online]. Available: `https://sensibo.com/products/sensibo-sky` (visited on 31/05/2023).

[14] 'Tibber pulse.' (2023), [Online]. Available: `https://tibber.com/no/store/produkt/pulse` (visited on 29/05/2023).

[15] 'Mill sense.' (2023), [Online]. Available: `https://millnorway.no/produkt/mill-sense-luftsensor/` (visited on 31/05/2023).

[16] 'Dette er han-porten.' (2023), [Online]. Available: `https://www.elvia.no/smart-forbruk/alt-om-din-strommaler/dette-er-han-porten/` (visited on 29/05/2023).

[17] 'Mill invisible wifi panelovn 400w.' (2023), [Online]. Available: `https://millnorway.no/produkt/mill-invisible-wifi-panelovn-400w/` (visited on 31/05/2023).

[18] 'Aquarea smart cloud.' (2023), [Online]. Available: `https://www.aircon.panasonic.eu/NO_no/happening/aquarea-smart-cloud-intro/` (visited on 12/03/2023).

[19] 'Easee hjemmelader.' (2023), [Online]. Available: `https://easee.com/no/hjemmelading/` (visited on 31/05/2023).

[20] 'Nest learning thermostat.' (2023), [Online]. Available: `https://store.google.com/gb/product/nest_learning_thermostat_3rd_gen?hl=en-GB` (visited on 31/05/2023).

[21] 'Tado smart thermostat.' (2023), [Online]. Available: `https://www.tado.com/all-en/smart-thermostat` (visited on 31/05/2023).

[22] 'Homey.' (2023), [Online]. Available: `https://homey.app/no-no/` (visited on 08/03/2023).

[23] 'Futurehome.' (2023), [Online]. Available: `https://www.futurehome.io/no/` (visited on 08/03/2023).

[24] 'Home assistant.' (2023), [Online]. Available: `https://www.home-assistant.io/` (visited on 08/03/2023).

[25] 'Google home.' (2023), [Online]. Available: `https://home.google.com/welcome/` (visited on 31/05/2023).

[26] 'Apple home.' (2023), [Online]. Available: `https://www.apple.com/home-app/` (visited on 31/05/2023).

[27] 'Tibber.' (2023), [Online]. Available: `https://tibber.com/no` (visited on 29/05/2023).

[28] 'Enode.' (2023), [Online]. Available: `https://enode.com/` (visited on 04/05/2023).

[29] 'Samsung smartthings.' (2023), [Online]. Available: `https://www.samsung.com/us/smartthings/` (visited on 10/05/2023).

[30] 'Matter standard.' (2023), [Online]. Available: `https://csa-iot.org/all-solutions/matter/` (visited on 04/05/2023).

[31] Y. Yao and D. K. Shekhar, 'State of the art review on model predictive control (MPC) in heating ventilation and air-conditioning (HVAC) field,' *Building and Environment*, vol. 200, p. 107 952, Aug. 2021. DOI: `10.1016/j.buildenv.2021.107952`. [Online]. Available: `https://doi.org/10.1016/j.buildenv.2021.107952`.

[32] J. Nubert, J. Köhler, V. Berenz, F. Allgöwer and S. Trimpe, 'Safe and fast tracking on a robot manipulator: Robust mpc and neural network control,' *IEEE Robotics and Automation Letters*, vol. 5, no. 2, pp. 3050–3057, 2020. DOI: `10.1109/LRA.2020.2975727`.

[33] M. G. Forbes, R. S. Patwardhan, H. Hamadah and R. B. Gopaluni, 'Model predictive control in industry: Challenges and opportunities,' *IFAC-PapersOnLine*, vol. 48, no. 8, pp. 531–538, 2015. DOI: `10.1016/j.ifacol.2015.09.022`. [Online]. Available: `https://doi.org/10.1016/j.ifacol.2015.09.022`.

[34] B. Liang, W. Liu, L. Sun, Z. He and B. Hou, 'Economic mpc-based smart home scheduling with comprehensive load types, real-time tariffs, and intermittent ders,' *IEEE Access*, vol. 8, pp. 194 373–194 383, 2020. DOI: `10.1109/ACCESS.2020.3033275`.

[35] M. Khosravi, N. Schmid, A. Eichler, P. Heer and R. S. Smith, 'Machine learning-based modeling and controller tuning of a heat pump,' *Journal of Physics: Conference Series*, vol. 1343, no. 1, p. 012 065, Nov. 2019. DOI: `10.1088/1742-6596/1343/1/012065`. [Online]. Available: `https://doi.org/10.1088/1742-6596/1343/1/012065`.

[36] S. Gros and M. Zanon, 'Data-driven economic nmpc using reinforcement learning,' *IEEE Transactions on Automatic Control*, vol. 65, no. 2, pp. 636–648, 2020. DOI: `10.1109/TAC.2019.2913768`.

[37] 'Docker.' (2023), [Online]. Available: `https://www.docker.com/` (visited on 04/05/2023).

[38] 'Docker container overview.' (2023), [Online]. Available: `https://www.docker.com/resources/what-container/` (visited on 29/05/2023).

[39] 'Docker hub.' (2023), [Online]. Available: `https://hub.docker.com/` (visited on 29/05/2023).

[40] 'Docker network.' (2023), [Online]. Available: `https://docs.docker.com/network/` (visited on 29/05/2023).

[41] 'Docker compose.' (2023), [Online]. Available: `https://docs.docker.com/compose/features-uses/` (visited on 29/05/2023).

[42] 'Kubernetes overview.' (2023), [Online]. Available: `https://kubernetes.io/docs/concepts/overview/` (visited on 29/05/2023).

[43] 'Kubernetes secrets.' (2023), [Online]. Available: `https://kubernetes.io/docs/concepts/configuration/secret/` (visited on 29/05/2023).

[44] 'Kubernetes components.' (2023), [Online]. Available: https://kubernetes.io/docs/concepts/overview/components/ (visited on 29/05/2023).

[45] 'Kubernetes api.' (2023), [Online]. Available: https://kubernetes.io/docs/concepts/overview/kubernetes-api/ (visited on 29/05/2023).

[46] 'Mqtt encryption.' (2023), [Online]. Available: https://www.hivemq.com/blog/mqtt-security-fundamentals-payload-encryption/ (visited on 29/05/2023).

[47] 'Django overview.' (2023), [Online]. Available: https://docs.djangoproject.com/en/4.2/intro/overview/ (visited on 29/05/2023).

[48] 'Django packages.' (2023), [Online]. Available: https://djangopackages.org/ (visited on 29/05/2023).

[49] 'Django rest framework.' (2023), [Online]. Available: https://www.django-rest-framework.org/ (visited on 29/05/2023).

[50] 'Influxdb concepts.' (2023), [Online]. Available: https://docs.influxdata.com/influxdb/v1.8/concepts/ (visited on 29/05/2023).

[51] 'Influxdb telegraf.' (2023), [Online]. Available: https://www.influxdata.com/time-series-platform/telegraf/ (visited on 29/05/2023).

[52] 'Influxdb flux introduction.' (2023), [Online]. Available: https://docs.influxdata.com/flux/v0.x/get-started/ (visited on 29/05/2023).

[53] 'Postgresql overview.' (2023), [Online]. Available: https://www.postgresql.org/about/ (visited on 29/05/2023).

[54] 'Paho mqtt pip package.' (2023), [Online]. Available: https://pypi.org/project/paho-mqtt/ (visited on 29/05/2023).

[55] 'Django.' (2023), [Online]. Available: https://www.djangoproject.com/ (visited on 23/05/2023).

[56] 'Fastapi.' (2023), [Online]. Available: https://fastapi.tiangolo.com/ (visited on 23/05/2023).

[57] 'Flask.' (2023), [Online]. Available: https://flask.palletsprojects.com/en/2.3.x/ (visited on 23/05/2023).

[58] 'Django databases.' (2023), [Online]. Available: https://docs.djangoproject.com/en/4.2/ref/databases/ (visited on 29/05/2023).

[59] 'Docker swarm.' (2023), [Online]. Available: https://docs.docker.com/engine/swarm/ (visited on 29/05/2023).

[60] 'Systemair vsr500.' (2023), [Online]. Available: https://www.systemair.com/no-no/produkter/boligventilasjon/ventilasjonsaggregater/save?sku=488863 (visited on 29/05/2023).

[61] 'Solcast api.' (2023), [Online]. Available: https://docs.solcast.com.au/#weather-sites (visited on 13/03/2023).

[62] 'Nordpool python api.' (2023), [Online]. Available: https://github.com/kipe/nordpool (visited on 16/05/2023).

[63] 'Kubernetes.' (2023), [Online]. Available: https://kubernetes.io/ (visited on 25/05/2023).

[64] 'Influx cli install instructions.' (2023), [Online]. Available: https://docs.influxdata.com/influxdb/v2.6/tools/influx-cli/ (visited on 20/03/2023).

[65] 'Influxdb flux queries.' (2023), [Online]. Available: https://docs.influxdata.com/influxdb/cloud/query-data/get-started/query-influxdb/ (visited on 19/05/2023).

[66] 'Docker desktop installation.' (2023), [Online]. Available: https://www.docker.com/products/docker-desktop/ (visited on 08/02/2023).

[67] 'Kind.' (2023), [Online]. Available: https://kind.sigs.k8s.io/docs/user/quick-start#installation (visited on 08/02/2023).

[68] 'Kubernetes creating a cluster.' (2023), [Online]. Available: https://kubernetes.io/docs/tutorials/kubernetes-basics/create-cluster/ (visited on 02/08/2023).

[69] 'Kubectl installation instructions.' (2023), [Online]. Available: https://kubernetes.io/docs/tasks/tools/ (visited on 25/04/2023).

[70] 'Skaffold installation.' (2023), [Online]. Available: https://skaffold.dev/docs/install/#standalone-binary (visited on 25/04/2023).

[71] 'Ubuntu 22.04.2 lts.' (2023), [Online]. Available: https://releases.ubuntu.com/jammy/ (visited on 26/05/2023).

[72] 'Microk8s.' (2023), [Online]. Available: https://microk8s.io/ (visited on 08/02/2023).

[73] 'Microk8s command reference.' (2023), [Online]. Available: https://microk8s.io/docs/command-reference (visited on 08/02/2023).

[74] NetworkChuck. 'Cloudflare reverse proxy tutorial,' Youtube. (14th Dec. 2022), [Online]. Available: https://www.youtube.com/watch?v=ey4u7OUAF3c.

[75] O. K. Husby, 'Subspace predictive control for smart homes,' M.S. thesis, Norwegian University of Science and Technology, 2023.

[76] 'Influxdb query functions.' (2023), [Online]. Available: https://docs.influxdata.com/influxdb/cloud/query-data/flux/ (visited on 26/05/2023).

[77] S. B. Solbakken, 'Optimal control of local energy storage in a residential home,' M.S. thesis, Norwegian University of Science and Technology, 2023.

[78] 'Prometheus.' (2023), [Online]. Available: https://prometheus.io/ (visited on 03/06/2023).

[79] 'Kubernetes metrics - meaning of cpu.' (2023), [Online]. Available: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/#meaning-of-cpu (visited on 27/05/2023).

[80] 'Datatilsynet purpose for collection.' (2023), [Online]. Available: https://www.datatilsynet.no/rettigheter-og-plikter/virksomhetenes-plikter/fastsette-formal/%20Don't%20store%20credentials%20longer%20than%20what%20is%20needed%20for%20the%20purpose,%20here%20it%20makes%20sense%20to%20delete%20passwords%20when%20users%20delete%20devices (visited on 27/03/2023).

[81] 'Gdpr, can we use data for another purpose?' (2023), [Online]. Available: https://commission.europa.eu/law/law-topic/data-protection/reform/rules-business-and-organisations/principles-gdpr/purpose-data-processing/can-we-use-data-another-purpose_en (visited on 27/03/2023).

[82] 'Datatilsynet: Rights of users.' (2023), [Online]. Available: https://www.datatilsynet.no/rettigheter-og-plikter/virksomhetenes-plikter/legge-til-rette-for-rettigheter/ (visited on 27/03/2023).

[83] 'Gdpr, do we always have to delete personal data if a person asks?' (2023), [Online]. Available: https://commission.europa.eu/law/law-topic/data-protection/reform/rules-business-and-organisations/dealing-citizens/do-we-always-have-delete-personal-data-if-person-asks_en (visited on 27/03/2023).

Appendix A

InfluxDB Python example code

```python
import influxdb_client
from influxdb_client.client.write_api import SYNCHRONOUS
from influx_database_tools import (
    measurement_structure_to_dict,
    get_measurements,
    database_to_dict,
)
import time

bucket = "HouseData"
org = "powiot"
token = "<ADD-TOKEN-HERE>"

# This instance may change, as long as we don't have a static address for the server
url = "https://influx.powiot.no"


# Intro to flux queries at
# https://docs.influxdata.com/influxdb/cloud/query-data/get-started/query-influxdb/


def get_values_from_flux_table(tables):
    values = []
    for table in tables:
        for record in table.records:
            values.append(record.values["_value"])
    return values


def get_measurements(bucket: str, org: str, client: influxdb_client):
    # If you want to see avaialble measurements in a bucket, use the following query
    query_available_measurements = f"""
    import \"influxdata/influxdb/schema\"
    schema.measurements(bucket: \"{bucket}\")
    """

    # print(f"Query: \n {query_available_measurements}")
    query_api = client.query_api()
    result = query_api.query(org=org, query=query_available_measurements)
    return get_values_from_flux_table(result)
```

```
def get_fields_for_measurement(
    measurement_name: str, bucket: str, org: str, client: influxdb_client
):
    query_available_fieldkeys = f"""
    import \"influxdata/influxdb/schema\"
    schema.measurementFieldKeys(
        bucket: \"{bucket}\",
        measurement: \"{measurement_name}\"
    )
    """
    query_api = client.query_api()
    result = query_api.query(org=org, query=query_available_fieldkeys)
    return get_values_from_flux_table(result)


def get_tag_keys_for_measurement(
    measurement_name: str, bucket: str, org: str, client: influxdb_client
):
    # List tag keys for that same measurement

    standard_tags = ["_start",
      "_stop",
      "_field",
      "_measurement",
      "_time",
      "house_id"
      ]

    query_available_fieldkeys = f"""
    import \"influxdata/influxdb/schema\"
    schema.measurementTagKeys(
        bucket: \"{bucket}\",
        measurement: \"{measurement_name}\"
    )
    """
    query_api = client.query_api()
    result = query_api.query(org=org, query=query_available_fieldkeys)
    available_tags_in_measurement = []
    for table in result:
        for record in table.records:
            value = record.values["_value"]
            if value not in standard_tags:
                available_tags_in_measurement.append(value)
    return available_tags_in_measurement


def get_tag_key_values_for_measurement(
    measurement_name: str, bucket: str, org: str, tag: str, client: influxdb_client
):
    # List tag keys for that same measurement
    query_available_fieldkeys = f"""
    import \"influxdata/influxdb/schema\"
    schema.measurementTagValues(
        bucket: \"{bucket}\",
        measurement: \"{measurement_name}\",
        tag : \"{tag}\"
    )
    """
    query_api = client.query_api()
    result = query_api.query(org=org, query=query_available_fieldkeys)
```

```python
    return get_values_from_flux_table(result)


def measurement_structure_to_dict(
    bucket: str, org: str, measurement: str, client: influxdb_client
):
    """
    Gets a dict structure of a single measurement.
    """
    measurement_to_dict = {}
    measurement_to_dict[measurement] = {}
    measurement_to_dict[measurement]["fields"] = get_fields_for_measurement(
        measurement, bucket, org, client
    )
    available_tags = get_tag_keys_for_measurement(measurement, bucket, org, client)
    available_house_ids = get_tag_key_values_for_measurement(
        measurement, bucket, org, "house_id", client
    )
    measurement_to_dict[measurement]["tags"] = {}
    measurement_to_dict[measurement]["tags"]["house_id"] = {}
    for house_id in available_house_ids:
        measurement_to_dict[measurement]["tags"]["house_id"][house_id] = {}
        for tag in available_tags:
            measurement_to_dict[measurement]["tags"]["house_id"][house_id][
                tag
            ] = get_tag_values_for_house_by_measurement(
                measurement, bucket, org, house_id, tag, client
            )
    return measurement_to_dict


def database_to_dict(bucket: str, org: str, client: influxdb_client):
    """
    Gets a total dict structure of the entire database, may take some time to run.
    """
    available_measurements = get_measurements(bucket, org, client)
    measurement_to_dict = {}
    for measurement in available_measurements:
        measurement_to_dict.update(
            measurement_structure_to_dict(bucket, org, measurement, client)
        )
        print("Done with measurement: ", measurement)
    return measurement_to_dict


def get_tag_values_for_house_by_measurement(
    measurement_name: str,
    bucket: str,
    org: str,
    house_id: str,
    tag: str,
    client: influxdb_client,
):
    # List tag keys for that same measurement
    query_available_fieldkeys = f"""
    import \"influxdata/influxdb/schema\"
    schema.tagValues(
    bucket: \"{bucket}\",
    tag: \"{tag}\",
    predicate: (r) => r["house_id"] == \"{house_id}\" and
    r["_measurement"] == \"{measurement_name}\"
    )
    """
    query_api = client.query_api()
    result = query_api.query(org=org, query=query_available_fieldkeys)
    return get_values_from_flux_table(result)


client = influxdb_client.InfluxDBClient(
    url=url,
    token=token,
    org=org,
)


# To get the available measurements use this function
available_measurements = get_measurements(bucket, org, client)
print("------------------")
print("Available measurements: ", available_measurements)
print("------------------")
measurement = available_measurements[0]  # Picking the first measurement

""" To get the entire structure
database = database_to_dict(bucket, org, client)
print(database)
"""


start_time = time.time()  # For timing the function

database = database_to_dict(bucket, org, client)
print("------------------")
print("Getting entire structure took %s seconds" % (time.time() - start_time))

print("------------------")
print("Database structure:", database)
print("------------------")

start_time = time.time()  # For timing the function
database_single_measurement = measurement_structure_to_dict(
    bucket, org, measurement, client
)
print("Getting single measurement took %s seconds" % (time.time() - start_time))
print("------------------")

print("Database structure for single measurement:", database_single_measurement)
print("------------------")

field = database[measurement]["fields"][0]  # Picking the first field
print("Measurement: ", measurement)
print("Field: ", field)
house_id = list(database[measurement]["tags"]["house_id"].keys())[
    0
]  # Picking the first house_id
print("House id: ", house_id)

print("------------------")
# Querieing the database based on the above information
query = f"""from(bucket:\"{bucket}\")
```

```
|> range(start: -20m)
|> filter(fn:(r) => r._measurement == \"{measurement}\")
|> filter(fn:(r) => r._field == \"{field}\")
|> filter(fn:(r) => r.house_id == \"{house_id}\")
"""
# Here I just take the last element of the time series,
# but you can do other things as well, or just remove it.

print(f"Query: \n {query}")

query_api = client.query_api()
result = query_api.query(org=org, query=query)

# To convert it into i.e a dictionary, do the following
results = {}
results[measurement] = {}
for table in result:
    for record in table.records:
        location = record.values["location"]
        # This is the field that groups the data
        if location not in results[measurement]:
            results[measurement][location] = {}
            results[measurement][location]["field"] = record.get_field()
            results[measurement][location]["time"] = []
            results[measurement][location]["value"] = []
        results[measurement][location]["time"].append(record.get_time())
        results[measurement][location]["value"].append(record.get_value())
print(results)
```

Appendix B

Backend API overview

Table B.1: Endpoint for **POST** /api/auth/login/

Field	**Value**
Method	**GET** /api/auth/login
Description	Logs into the backend.
Headers	Content-Type: application/JSON
Parameters	None
Responses	200: Success, logged in. 400: Bad request, invalid login.

Request body:

```
{
    "username" : "string",
    "password" : "string"
}
```

Table B.2: Endpoint for **POST** /api/auth/logout/

Field	**Value**
Method	**GET** /api/auth/logout
Description	Logs out of the backend.
Headers	Content-Type: application/JSON
Parameters	None
Responses	200: Success, logged out.

Table B.3: Endpoint for **GET** /api/devices/

Field	**Value**
Method	**GET** /api/devices/
Description	Gets a list of all current device integrations.
Headers	Content-Type: application/JSON
Parameters	None
Responses	200: Success, list of device integrations. 403: Unauthorized access, user not logged in.

Table B.4: Endpoint for **POST** /api/devices/

Field	Value
Method	**POST** /api/devices/
Description	Creates a new device
Headers	Content-Type: application/JSON
Parameters	None
Responses	200: Success, added device integration. 400: Bad request, bad data.

Request body:

```
{
  "name": "string",
  "docker_hub_image": "string",
  "required_fields": [
    "string"
  ],
  "database_group_name": "string"
}
```

Table B.5: Endpoint for **GET** /api/devices/{device_id}

Field	Value
Method	**GET** /api/devices/{device_id}
Description	Gets information about a specific device based on its id.
Headers	Content-Type: application/JSON
Parameters	{device_id} → id of device
Responses	200: Success, data about that device 404: Not found

Table B.6: Endpoint for **DELETE** /api/devices/{device_id}

Field	**Value**
Method	**DELETE** /api/devices/{device_id}
Description	Deletes a specific device integration based on its id.
Headers	Content-Type: application/JSON
Parameters	{device_id} → id of device
Responses	200: Success, deleted device 404: Not found

Table B.7: Endpoint for **PATCH** /api/devices/{device_id}

Field	**Value**
Method	**PATCH** /api/devices/{device_id}
Description	Updates an existing device integration.
Headers	Content-Type: application/JSON
Parameters	{device_id} → id of device
Responses	200: Success, data updated 404: Not found 400: Bad request

Request body has to include at least one of the following:

```
{
  "name": "string",
  "docker_hub_image": "string",
  "required_fields": [
    "string"
  ],
  "database_group_name": "string"
}
```

Table B.8: Endpoint for **POST** /api/devices/{device_id}/update/

Field	**Value**
Method	**POST** /api/devices/{device_id}/update
Description	Updates the image of all running pods with the selected device id.
Headers	Content-Type: application/JSON
Parameters	{device_id} → id of device
Responses	200: Success, images updated 404: Not found 204: No content, no house has that device.

Table B.9: Endpoint for **GET** /api/houses/

Field	**Value**
Method	**GET** /api/houses/
Description	Gets a list of all houses under the current user, admin gets all houses.
Headers	Content-Type: application/JSON
Parameters	None
Responses	200: Success, list of houses. 403: Unauthorized access, user not logged in.

Table B.10: Endpoint for **GET** /api/houses/{house_id}

Field	**Value**
Method	**GET** /api/houses/{house_id}
Description	Gets information about the selected house using its id.
Headers	Content-Type: application/JSON
Parameters	{house_id} → id of the house
Responses	200: Success, information about the house. 403: Unauthorized access, user not logged in, or doesn't have access to that house.

Table B.11: Endpoint for **DELETE** /api/houses/{house_id}

Field	**Value**
Method	**DELETE** /api/houses/{house_id}
Description	Deletes the selected house using its id.
Headers	Content-Type: application/JSON
Parameters	{house_id} → id of the house
Responses	200: Success, information about the house. 403: Unauthorized access, user not logged in, or doesn't have access to that house.

Table B.12: Endpoint for **GET** /api/houses/{house_id}/devices

Field	**Value**
Method	**GET** /api/houses/{house_id}/devices
Description	Lists all devices for a selected house, by id.
Headers	Content-Type: application/JSON
Parameters	{house_id} → id of the house
Responses	200: Success, list of devices for house with id. 403: Unauthorized access, user not logged in, or doesn't have access to that house.

Table B.13: Endpoint for **GET** /api/houses/{house_id}/devices/{devices_id}

Field	**Value**
Method	**GET** /api/houses/{house_id}/devices/{devices_id}
Description	Shows data about a device based on house id and device id.
Headers	Content-Type: application/JSON
Parameters	{house_id} → id of the house {device_id} → id of the device
Responses	200: Success, data about device. 403: Unauthorized access, user not logged in, or doesn't have access to that house.

Table B.14: Endpoint for **GET** /api/houses/{house_id}/devices/{devices_id}/count/{count}

Field	**Value**
Method	**GET** /api/houses/{house_id}/devices/{devices_id}/{count}
Description	Shows data about a device based on house id, device id and count.
Headers	Content-Type: application/JSON
Parameters	{house_id} → id of the house {device_id} → id of the device {count} → numbering of the device, usually 0 unless there are multiples of same device in a house.
Responses	200: Success, data about device. 403: Unauthorized access, user not logged in, or doesn't have access to that house.

Table B.15: Endpoint for **DELETE** /api/-houses/{house_id}/devices/{devices_id}/count/{count}

Field	**Value**
Method	**DELETE** /api/houses/{house_id}/devices/{devices_id}/
Description	Deletes a device based on house id, device id and count. This will delete it from the database as well as shut down the pod in kubernetes
Headers	Content-Type: application/JSON
Parameters	{house_id} → id of the house {device_id} → id of the device {count} → numbering of the device, usually 0 unless there are multiples of same device in a house.
Responses	204: No content 403: Unauthorized access, user not logged in, or doesn't have access to that house. 404: Not found.

Table B.16: Endpoint for **POST** /api/houses/{house_id}/devices/{devices_id}/count/{count}/restart

Field	**Value**
Method	**POST** /api/houses/{house_id}/devices/{devices_id}/count/{count}/restart
Description	Restarts a device based on house id, device id and count. This will restart the kubernetes pod linked to this instance.
Headers	Content-Type: application/JSON
Parameters	{house_id} → id of the house {device_id} → id of the device {count} → numbering of the device, usually 0 unless there are multiples of same device in a house.
Responses	200: Device restarted. 403: Unauthorized access, user not logged in, or doesn't have access to that house. 404: Not found.

Table B.17: Endpoint for **POST** /api/houses/{house_id}/devices/{devices_id}/data

Field	**Value**
Method	**POST** /api/houses/{house_id}/devices/{devices_id}/data
Description	Retrieve data about the specified device for that house. Automatically scales the resolution of data based on timeframe to avoid overloading the system.
Headers	Content-Type: application/JSON
Parameters	{house_id} → id of the house {device_id} → id of the device
Responses	200: Success, with accompanying data. 403: Unauthorized access, user not logged in, or doesn't have access to that house. 404: Not found.

Request body has to include at least field and start, and in the time format below:

```
{
  "field": "string - name of measurement field",
  "start": "2023-04-20T10:00 - string in ISO format",
  "end": "2023-04-20T11:00 - string in ISO format",
}
```

Table B.18: Endpoint for **POST** /api/houses/{house_id}/devices/{devices_id}/new

Field	**Value**
Method	**POST** /api/houses/{house_id}/devices/{devices_id}/new
Description	Boots up new device under the selected house.
Headers	Content-Type: application/JSON
Parameters	{house_id} → id of the house {device_id} → id of the device
Responses	200: Device started. 401: Unauthorized access, user not logged in, or doesn't have access to that house. 404: Not found.

Request body has to include all the fields required from the devices list of required fields

```
{
  "required_field_1": "string",
  "required_field_2": "string",
  "...": "...",
}
```

Table B.19: Endpoint for **POST** /api/houses/new

Field	**Value**
Method	**POST** /api/houses/new
Description	Creates a new house under the current user.
Headers	Content-Type: application/JSON
Parameters	None
Responses	200: House created. 401: Unauthorized access, user not logged in.

Table B.20: Endpoint for **GET** /api/users/

Field	**Value**
Method	**GET** /api/users/
Description	Lists the current user, or if admin lists all users.
Headers	Content-Type: application/JSON
Parameters	None
Responses	200: Success and list of users. 400: Bad request.

Table B.21: Endpoint for **POST** /api/users/new

Field	**Value**
Method	**POST** /api/users/new
Description	Creates a new user, currently only admin users can create new users to limit access to the system.
Headers	Content-Type: application/JSON
Parameters	None
Responses	201: User created. 400: Bad request.

Request body:

```
{
  "username": "string",
  "password": "string",
  "email": "string",
}
```