UNIVERSITY OF DUISBURG-ESSEN
FACULTY OF ENGINEERING
DEPARTMENT OF COMPUTER SCIENCE AND APPLIED COGNITIVE SCIENCE

Master's thesis

FooSH: A Framework for outcome-oriented Smart Homes

Malte Josten
Matriculation number: 3066184
Applied Computer Science (Master)

UNIVERSITÄT DUISBURG ESSEN

Distributed Systems Group, Department of Computer Science
Faculty of Engineering
University of Duisburg-Essen

December 1, 2023

First Reviewer: Prof. Dr-Ing. Torben Weis
Second Reviewer: Prof. Dr. Gregor Schiele
Period of time: June 5 2023 - December 4 2023

Abstract

The ubiquity of smart homes, driven by the increasing performance and availability of IoT devices, brings challenges due to the diverse range of appliances, vendors, protocols, and interfaces. The main issue lies in users knowing what they want to achieve but struggling with how to execute it in smart homes that often lack elaborate goal-oriented support. Thus, we address the existing gap in universally applicable goal-oriented smart home systems while prioritizing the user-centric goals of comfort and convenience by developing a Java framework prototype (*FooSH*) with the emphasis on software qualities. Based on the design choices and its architecture, the prototype allows for its integration into state-of-the-art smart home systems without interfering with existing parts while accommodating arbitrary prediction methods. These prediction models help smart home users to achieve a desired smart home state, only by defining the final goal and not an explicit way to reach it. A proof of concept demonstrating the deployment into an existing system and the ease of incorporating new prediction models was constructed. Despite some limitations, the framework suggests a successful contribution to (goal-oriented) smart home research, serving as a promising starting point for further enhancements in smart home user experience and satisfaction.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

With the performance increase of IoT devices and the improvement of their availability to the broad public, smart homes are more ubiquitous than ever. However, the variety of smart home appliances, their various vendors, protocols, and different interfaces pose a big hurdle, especially for newcomers, by increasing the system's complexity. Unfortunately, this comes with a straitened user experience as interacting with a potentially complex smart home system also becomes more challenging, either due to the user's technical inexperience or the smart home's extensive setup, configuration, and operation. Here, one of the main problems is that "users usually know what they want to do, but they do not know how to do it" [36, p. 126]. Since modern smart homes only provide limited support for goal-oriented operations, situations often arise where users more or less successfully define home automations, scenes, and similar on their own that only partly satisfy their initial needs. Consequently, the user is the "smart" thing in the smart home system that does all the cognitive work [27] - essentially making the home smart using their own intelligence. Thus, goal-oriented approaches in smart homes are highly attractive as they fulfill the original intentions of smart homes: comfortability and convenience without straining the user's cognitive abilities. Though numerous elaborate approaches introduce goal-oriented systems for smart homes, none of them provides a universally deployable framework. With the wide selection of smart home ecosystems and the various possibilities for achieving user goals, a framework's generalizability is one of the key factors determining its success and usefulness in a goal-oriented environment.

For this reason, we established three research questions that deal with the challenges of how to develop a system that can be integrated into (nearly) every existing smart home system while still being able to use arbitrary goal resolution mechanisms. They also highlight beneficial steps and assisting measures for developing a sustainable and useful software solution.

The main contribution of this thesis is the development of an outcome-oriented (goal-oriented) framework prototype that aims to answer the research questions by providing the necessary interfaces and features to be universally applicable while also considering its sustainability, making it a long-lasting, easy-to-maintain, and extendable product.

The thesis is structured into eight chapters, with each chapter addressing a specific aspect of research, implementation, or evaluation. Chapter 2 highlights the current state of the art of smart homes and emerging problems, and it defines the goal and relevant research questions for this work. The following two chapters (Chapter 3 and Chapter 4) present the related work and important background information for the upcoming chapters. We introduce our approach in Chapter 5 and provide an abstract framework definition. Chapter 6 explains the prototype's design process and the architecture, followed by implementation details and a short showcase of an exemplary user interaction. Then, we evaluate the prototype by first discussing a proof of concept in Section 7.1 and subsequently picking up on the research questions, pointing out the prototype's limitations and comparing it to similar approaches. Lastly, Chapter 8 concludes the thesis and presents some ideas for future work.

Chapter 2

System Model

For this work, we consider IoT systems, namely smart homes, which, according to Satpathy, can be described as "a home which is smart enough to assist the inhabitants to live independently and comfortably with the help of technology" and where "all the mechanical and digital devices are interconnected to form a network, which can communicate with each other and with the user to create an interactive space" [43, pp. 43-44]. One can therefore say that smart homes aim to improve their users' quality of life, with a special emphasis on comfort and convenience [6, 17, 30]. Additionally, a smart home can be formally described as consisting of:

- A **gateway** or **hub** H that acts as the mediator between the user and smart things by posing as a central point of communication. Popular examples for hubs are Amazon Alexa[1] or Google Nest Hub[2].
- A set of **smart things** $D := \{d_1, \ldots, d_u \mid u \in \mathbb{N}\}$ that comprises smart objects which have some kind of controllable actuator and/or a readable sensor. Thus, a smart thing $d := (\{a \mid a \in A \cup \emptyset\}, \{s \mid s \in B \cup \emptyset\})$ has a set of actuators and sensors respectively.
- A set of **actuators** $A := \{a_1, \ldots, a_v \mid v \in \mathbb{N}\}$ that includes the actuators of all smart things where the value of $a_v \in \mathbb{R}$ indicates the actuator's state, e.g., $a = 60$ could be the brightness in percent of the actuator (light unit) of a smart light bulb.
- A set of **sensors** $S := \{s_1, \ldots, s_w \mid w \in \mathbb{N}\}$ that contains all the sensors currently present in the smart home system. Similar to an actuator, a sensor $s_w \in \mathbb{R}$ provides information about its reading. A reading of a thermostat could yield the value $s = 21.3$, giving the temperature of, e.g., an underfloor heating installation in $[°C]$.
- Some kind of **user interface** in form of, e.g., a physical device, a graphical user interface (e.g., mobile app), or a web interface. It can also be part of the hub, as some models, like Amazon Alexa, can be directly interacted with.
- A **user** that interacts with the smart home's user interface and is affected by changes in and of the smart home state.

[1] `https://www.amazon.com/alexa-smart-home/`
[2] `https://store.google.com/us/category/nest_hubs_displays`

- A **developer** which can, but does not need to, be the same person as the user, configures the smart home system and is assumed to have technical experience.

Figure 2.1 shows an exemplary smart home structure with a hub and four smart devices $d_1 - d_4$. Each device has at least one actuator or sensor and communicates with the hub. Instead of interacting with the smart devices directly, the user utilizes the UI to pass down any instructions to the hub, which, in turn, forwards them to the corresponding devices.

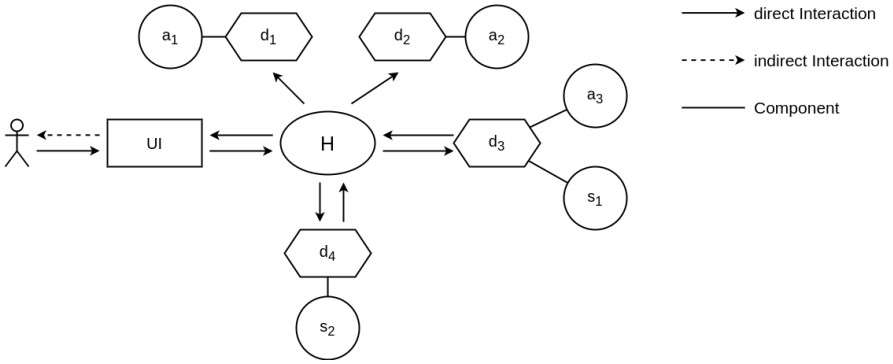


Figure 2.1: Smart Home Structure

Besides the commonly used grouping feature of smart homes, with which one can group multiple smart devices and control them all at once, there is also another kind of grouping or assignment concerning so-called *properties*. Something like noise, humidity, or temperature can be seen as a smart home *property*. They are characterized by the fact that they cannot (usually) be influenced directly by an interaction with the smart home, as their changes often arise indirectly as a result of side effects from various devices. Hence, based on their (side-)effects, smart devices can be additionally assigned or linked to one or multiple different properties. An air conditioning (AC) unit can be an example of having an effect on multiple properties: when active, it produces noise; it regulates the temperature and may also change the air's humidity.

2.1 Problem Definition

The presented system model not only comes with the positive sides of smart homes, but it also brings with it the following yet to solve problems:

Usually, users control devices and actuators manually. Figure 2.2 depicts an exemplary interaction in a scenario where the user wants to achieve a brightness level of 600lm. They set the brightness of the two smart things d_1 and d_2 (e.g., smart light bulbs) individually, and re-adjust the value of the first instruction, as, after checking the actual brightness with the sensor reading of s_2, a brightness level of 450lm does not achieve the initial goal.

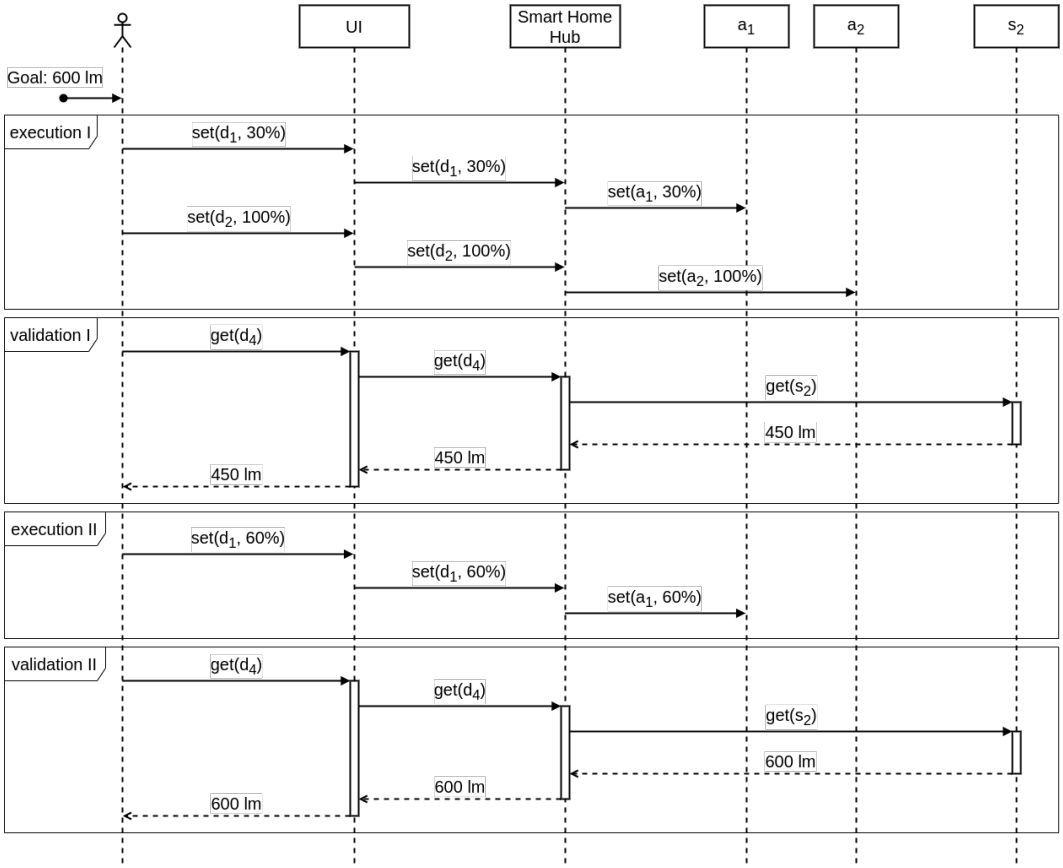


Figure 2.2: Smart Home Interaction

It might seem to be a good idea to create and use home automations instead. Given an average smart home, this might not be feasible as the effects of actuators on the real world are not captured, and other physical and environmental influences are not considered. If the smart home user manipulates the smart home's state, e.g., turns on

a light, the effect on the entire system varies depending on more factors than just the instruction from the user. The time of the day or weather might play a vital role in the outcome and system changes. Since the smart home does not know about its physical surroundings nor does it have a simulation for trying to emulate them, a consistent home automation result is unlikely and only achievable by trial-and-error or by introducing further functionalities.

Home automations and the smart home system are very limited in capturing smart devices' effects concerning the external[3] influences on the environment. They cannot really capture external factors, as this would establish the need for a sophisticated simulation or physical model. Deploying such a model, in general, is not feasible since most smart home systems run on embedded or IoT devices, which run on limited resources. And even if powerful hardware were available, creating a performant, reliable, and user-friendly model that takes into account all the different smart home properties and their impact on the smart home and its devices is considered unrealistic [37].

Based on the limitations described in the preceding paragraph, smart home users cannot rely on any capturing method for external factors and need to configure their smart home and instructions manually. Unfortunately, it can be challenging and tedious to create a sustainable automation for, let's say, reading a book on a sofa. Just setting the brightness of a smart lamp could suffice for some scenarios (at noon in spring), but for others (in the late afternoon in autumn) additional actions might be necessary to reach a consistent level of brightness. Usually, environmental factors, such as the weather, date, or time are not considered by automations. Thus, manually crafted and executed instructions, especially without considering other (external) factors, tend to be inconsistent and, subsequently, lead to unsatisfactory results and poor user experience.

The introduction of properties and the fact that one smart device can have an influence on multiple properties means that possible conflicts can arise in the course of achieving specific property values. Since one device can be assigned to multiple properties, e.g., temperature and noise, in the case of an AC unit, reaching the different properties' goal values includes changing the AC's state at various times. However, the different changes in the AC's state can yield conflicting behaviour. One might define a lower temperature goal but simultaneously want the room to be quiet. To reach both goals, the system would likely produce a conflict by trying to (i) use the AC to cool down the room but also by trying to (ii) not use the AC to keep the noise level low.

Looking at the aforementioned issues, it becomes clear that the central goals of smart homes, being convenient and ameliorating overall comfort while being in and using a smart home, suffer greatly. Furthermore, the problems hamper further advances towards more sophisticated user- and goal-centric smart homes.

[3] Here, *external* refers to anything outside the smart home system, like the weather, human behaviour, or other machines/software.

2.2 Goal

A (software) system is needed that, based on the input of a desired value for a variable, determines a corresponding smart home configuration and gives the smart home system appropriate instructions to achieve the wanted result. For this, it should include mechanisms for detecting and capturing environmental influences as well as their effects on the smart home and methods for determining the resulting necessity of including and changing the state of additional smart devices. The necessity of changing a device's state should be concluded with the help of the determined configuration and the underlying prediction method. The developer can freely define suitable or necessary factors as selection/decision criteria. Therefore, the results of determining an appropriate configuration can vary immensely from one model to another. Possible selection criteria would be, for example, predetermined user preferences or the reduction of electricity consumption.

It is also substantial to solve or prevent conflicts when trying to achieve multiple property values concurrently. While ignoring the apparent likelihood of clashes, the user's needs stay unsatisfied, and, if not given proper feedback or insights, a user might feel annoyed or even fooled by the system. Consequently, it can be said that the problem of property conflicts, if not properly addressed, will severely impact the user experience and render any benefits or improvements of possible solutions nugatory, however good they may be.

The software design should be constructed with the aim to minimize user interaction as much as possible and thereby increase user-friendliness, especially for non-technical users.

2.3 Research Questions

The overarching goal of this work is to develop a software solution to mitigate the aforementioned problems and challenges. Designing and implementing such software requires extensive planning and thoughtful considerations regarding various aspects. For this reason, the following research questions were developed, intended for aiding the conceptualization of solutions and assuring that working on it is done based on objective and scientifically supported processes.

(RQ 1) How to deal with and connect to a wide variety of smart home systems?
Given the wide range of different smart home systems, like Amazon's *Alexa Smart Home* or *Google Home*[4], determining an appropriate interface to connect to the existing smart home can be tedious and is not per se generalizable. The presence of multiple smart

[4] `https://home.google.com/welcome/`

home systems inside a smart home makes this point even more evident. A solution should include an abstraction that lets the developer and user decide which smart home to integrate it into and not restrict the pool of compatible smart home systems by only supporting selected systems. Hence, it is necessary to find a way to generalize the connection between solution software and smart home in a way that still leaves enough room for smart home selection, future adjustments, and a holistic view of all smart home systems.

(RQ 2) How to deal with and allow incorporation of different prediction models?
From now on, a mechanism that combines the properties of an environment model and its effects on the smart home system with goal-oriented user interactions is called a *Prediction Model*. It approximates the system environment on the one hand, and, on the other hand, it also allows users to define goals that, based on the known approximated system behaviour, are used to determine a corresponding smart home configuration[5] to achieve said goal. Not every method for the prediction/generation of smart home configurations is suitable for every use case. It is unlikely that a given prediction method for, let's say, brightness levels is also applicable for generating smart home configurations regarding humidity levels or temperature. Because there is a variety of different prediction methods, a solution should provide the developer with an interface that allows the incorporation of arbitrary prediction methods without interfering with other capabilities. With the potential conflicts between multiple prediction models comes another challenge that must be taken into account when devising a solution.

(RQ 3) Which software qualities are crucial for developing a sustainable and useful solution? What can be done to accomplish and enforce selected software qualities?
In this work, we base our understanding of software sustainability on the definition of Venters *et al.* [56]. They define software sustainability as the degree that indicates the software's ability to be maintained or extended by a developer over time and be used as long as possible without degrading the developer or user experience. The latter also introduces the term *longevity*, for it is tightly coupled with sustainability, contributing a big part to a software's overall quality. Hence, it is necessary to find means for the development of sustainable software and the methods for its evaluation. Besides choosing an appropriate software architecture and design principles, which are already widely known and applied, software qualities offer another great opportunity to help during development as well as provide a form of comparability and means for evaluation.

The system in which the solution software will be deployed, is a user-centric system, with mostly technically-inexperienced users. Here, the degree to which the solution's interfaces are intuitive to work with, the frequency of errors and faults, i.e., the user experience in general, and the solution's degree of sustainability determine the usefulness

[5] A smart home configuration describes the state of all smart devices in the smart home.

of the solution. Developing impractical and inconvenient software misses the main point of smart homes: their pursuit of improving comfort and convenience.

Therefore, solutions should follow and make use of best practices as well as consider software qualities to ensure their sustainability and usefulness. This usually improves the overall quality of the final software product and makes it more appealing for end-users. However, since there exist diverse definitions of software qualities and views on their respective relevance, it is necessary to (i) ascertain a fitting software quality model, (ii) determine relevant qualities, and (iii) find fitting evaluation methods. Using a software quality model in combination with established quality evaluation methods should allow for better comparability of the solution software with other approaches.

Chapter 3

Related Work

This chapter is divided into two sections. The first part deals with approaches similar to the solution presented in Chapter 5 that, at least to some extent, solve (parts of) the preceding problem description. The second part highlights the necessary information for the REST architecture and a way to evaluate it.

3.1 Goal-orientated Smart Homes Systems

Trying to predict user behavior and their needs is not a new problem researchers are trying to solve. In 1998, Mozer proposed his system *ACHE* (adaptive control for home environments), with the intent to program and adapt itself to the needs of the smart home's users [32]. During the adaption, it has two main objectives: (i) predict the inhabitants' needs, and (ii) conserve energy/costs as much as possible. To find the optimal compromise between (i) and (ii), reinforcement learning, based on a dollar cost function (putting a price tag on the inhabitants' discomfort and energy usage), and dynamic programming or models of the environment were used. Ultimately, ACHE should find a policy, i.e., a sequence of controls, to minimize average costs. For each control domain (we call it "environment variable"), like temperature or lighting, a different policy has to be defined, and based on the type of control domain, the policy would enforce either a reinforcement learning or environment modeling approach. ACHE was tested in multiple real-world scenarios where it also considered weather data, like outdoor temperature, to find a promising policy for the indoor heating of a house. He showed that, at this time, ACHE outperformed other common policies, encouraging dividing the smart home into smaller parts (control domains) and not aiming to simulate or predict the entire smart home system, including external factors.

King *et al.* published a paper in which they introduced *Sasha* (smarter smart home assistant), an LLM (Large Language Model)-based reasoning system, to support voice assistants in comprehending and extracting the goal of or intent behind a vocally issued user command [27]. Thereby, it addresses the weaknesses of current voice interfaces to reliably interpret and act upon commonly ambiguous and colloquial vocal interactions.

For this, it uses an iterative approach, shown in Figure 3.1, that first extracts the central goal of the voice command using an LLM, checks whether the detected goal is reachable (*Clarifying*), determines what smart devices and actions are necessary to reach the said goal (*Filtering* and *Planning*), and ultimately presents the user with the generated actions (*Feedback*) before executing them (*Execution*). During the *Feedback* phase, the user has the opportunity to modify the actions and influence the reasoning cycle. They emphasized the automatic selection of relevant smart devices during the *Filtering* step since LLMs tend to include irrelevant devices to achieve the goal [27]. Thus, a dataset of the most popular IFTTT[1] automations was used to train the LLM to recognize common automation commands and the corresponding relevant devices. *Sasha* was successfully tested in a real-world scenario within a smart home environment and showed improved human-perceived quality in comparison to other approaches.

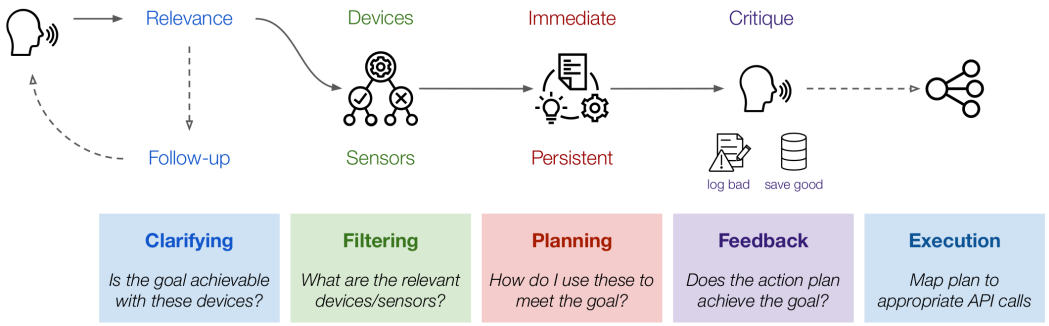


Figure 3.1: Sasha's iterative reasoning cycle to achieve vocally induced goal-oriented instructions [27]

Palanca *et al.* first started working on their goal- and service-oriented architecture for smart home environments in 2011 and introduced it as the *Distributed Goal-Oriented Computing (DGOC) architecture* in one of their latest publications [35, 36]. The architecture is built around so-called agents that provide some kind of service (e.g., smart or I/O devices) and plans (execution sequences of said services). Both the available services and plans are considered when determining the most promising plan to reach a goal. If no matching plan is found, the framework is able to construct a new one based on the already existing plans and services. The user inputs a goal with the help of an agent (e.g., a voice assistant agent), and the goal is assumed to be already extracted from the user's interaction by utilizing, for example, natural language processing mechanisms. Their long-term goal is to create an operating system based on the introduced DGOC architecture, which, at this point, was not yet done.

[1] `https://ifttt.com/`

Bisicchia *et al.* proposed a framework using the declarative programming language Prolog [39] for solving the problem of potential conflicts between multiple user-defined goals, introduced in Section 2.2 [7]. This framework is deployed as an LPaaS (Language Programming As A Service [10]), which allows for easy integration into other software products and ecosystems and deployment on various platforms. For the LPaaS to work, one first needs to define the smart home, its devices (actuators and sensors), and their relationships to various *propertyTypes* (analogous to an environment variable in FooSH) using Prolog. After properly set up, it can mediate between conflicting goal-oriented user interactions and/or administrator-specified goals using mediation policies (e.g., find the average between conflicting states). This mediation process can be divided into the following steps:

1. Collect the pending user requests that include a propertyType's goal value,
2. Mediate between the collected requests based on a previously specified mediation policy and
3. Determine the state of actuators to reach the smart home configuration determined in step 2.

Lastly, the proposed framework enables switching between and creation of mediation policies at runtime, which results in high flexibility and an improved user experience.

3.2 REST(ful) web API

Representational State Transfer (REST) is one of the most known and used web API architectures [42]. The *2021 State of the API Report* from Postman[2] showed that nearly 95% of survey participants used REST in 2021 (see Figure 3.2). This trend continues to this day with roughly 85% working with REST in 2023 [2], and it suggests a promising future for the ongoing usage of and interest in RESTful web APIs.

3.2.1 Definition

REST is an architecture for distributed hypermedia systems (e.g., the World Wide Web) developed by Roy T. Fielding as part of his dissertation [14]. Fielding saw multiple problems with creating and evaluating improvements to the Web and its protocols, as they were growing very fast and there was no method to expand the already deployed Web architecture [14, pp. 71-75]. For this reason, he introduced six constraints for designing a RESTful system, which will shortly be summarized in the following paragraph.

Client-Server. A client-server architecture helps to enforce the separation-of-concern design pattern, thus improving the *scalability* and *portability* of the system.

[2] `https://www.postman.com/`

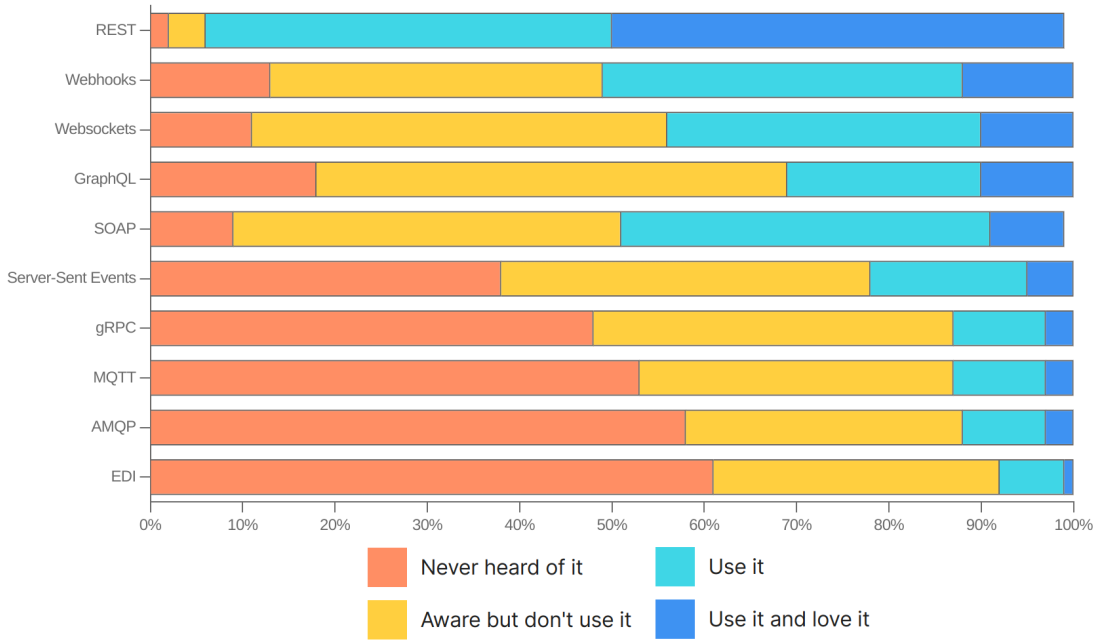


Figure 3.2: API architecture popularity, adapted from [1]

Stateless. Each request to the server must contain all the required information needed to process it. Therefore, each client stores the system's state locally and no (shared) context is present at the server. This affects the overall *reliability* and *scalability* of the system as the server does not need to manage any kind of state and can process each request individually and without context.

Caching. It should be possible to cache responses for a given time interval to reduce network traffic and improve the *performance* by preventing repeated requests.

Uniform Interface. A uniform interface supports the interfaces *usability* since the architecture is simplified and further evolvements of both the client and the server can be done independently.

Layered System. Within a layered system, each component can only see and interact with its own and adjacent layers, guaranteeing separation of concern and providing *security* as well as *scalability* benefits.

Code-On-Demand. This is the only optional constraint of the REST architecture, and it allows clients to download additional code to reduce the amount of initially needed features as they can be fetched and implemented at runtime.

Seeing that there sometimes seems to exist a bit of confusion concerning the difference between the terms *REST* and *RESTful* both in academics and industry, a brief explanation is given: REST is the name of the architecture, and every system or web API that sticks to the previously depicted constraints is called RESTful [26].

3.2.2 Maturity Models

Maturity models help determine a software's maturity level, i.e., making it easier to categorize and subsequently evaluate them. Therefore, they create a good starting point and aid the developer and evaluator in finding the proper focus for implementing, validating, and assessing a software system that uses a web API [42]. There are a multitude of different use cases: cybersecurity [3], software processes [20], and web APIs; to name a few. This work only considers the following two maturity models for RESTful web APIs: the *Richardson Maturity Model* and the *WS*^{3} *Maturity Model*.

Richardson Maturity Model

The *Richardson Maturity Model* was developed and presented by Leonard Richardson in 2008 and is the most used maturity model when talking about RESTful web APIs [41]. It establishes four levels for which an API, upon reaching level 3 is called "truly RESTful" [14, 31]. Each level adds additional functionality to the system and they can be described as follows:

Level 0: Starting at level 0, the API only consists of one URI[3], one HTTP method (`POST`), and only uses POX (plain old XML), similar to RPC (remote procedure calls) or SOAP (simple object access protocol).

Level 1: The next level introduces different resources. Hence, each resource has a corresponding URI and can be addressed with the HTTP `POST` method.

Level 2: By not only allowing the HTTP `POST` method but also the other HTTP methods (e.g., `GET` and `DELETE`)[4], the API can now use them as close to as they are used by HTTP. This leads to intuitive interactions based on the CRUD operations (Create, Read, Update, and Delete).

Level 3: The final level adds hypermedia controls that are also called HATEOAS (Hypermedia As The Engine Of Application State). They enable resources to describe their capabilities and relations to other resources so that the client does not need to know anything about the API structure but can navigate it only using the provided controls.

[3]Uniform Resource Identifier as defined by [55]
[4]Includes all methods defined by [23]

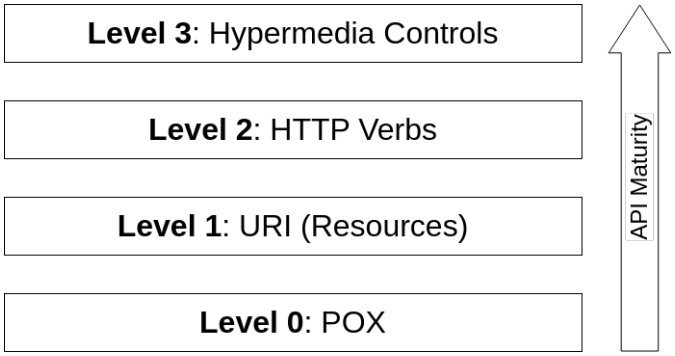


Figure 3.3: The four levels to reach REST [15][41]

WS^3 Maturity Model

The *WS^3 Maturity Model* was explicitly designed for semantic RESTful APIs by Salvador & Siqueira in 2015 and it comprises three dimensions [42]. A semantic (RESTful) API is an API where resources and available operations are semantically described, similar to the principles of the *Semantic Web*[5]. That means that machines can understand the API and its resources without needing prior manual configuration. Technologies like HATEOAS can help to achieve just that. The three dimensions are the Design, Profile, and Semantic dimension.

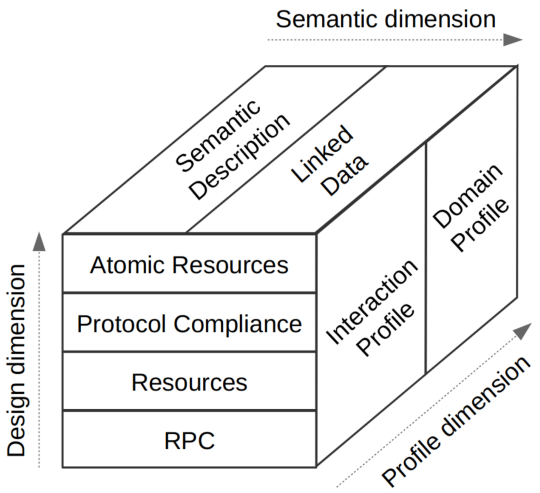


Figure 3.4: WS^3 Maturity Model [42]

[5]For more details, see [60]

The design dimension correlates to the different levels of the Richardson Maturity Model.

The semantic dimension deals with the degree to which resources and their relations are semantically described. APIs that only describe their properties and respective operations fall into the level of *Semantic Description*. They reach the level of *Linked Data* if they also provide a semantic description of their relationships.

The profile dimension reflects in which detail the API deals with documentation and if it only provides semantics for the communication protocol (*Interaction Profile*). If it also includes application domain-specific details, it is categorized into the *Domain Profile*.

With these three dimensions, one can classify web APIs with the triplet

$$\mathrm{D}X\text{-}\mathrm{S}Y\text{-}\mathrm{P}Z$$

where DX corresponds to the Design dimension level X, SY corresponds to the Semantic dimension level Y, and PZ corresponds to the Profile dimension level Z. A dimension with level 0 indicates, that this dimension is not supported. For example, D2-S2-P0 is the classification for a Semantic web API that uses resources, and describes its resources and their relationships semantically but does not support profiles, i.e., is undocumented.

3.2.3 Differences to other web APIs

In this section, REST is compared to other popular web technologies, namely RPC and SOAP. We include RPC for this comparison, as it is one of the most referred to architectural styles for building web APIs; right besides resource-oriented architectures like REST [34, 42]. Logically, SOAP is also included for its prominence in the web API community (see Figure 3.2). Consequently, RPC and SOAP can be seen as the most vigorous competitors to REST, making it worth comparing them.

RPC A Remote Procedure Call architecture focuses on functions or actions, whereas REST focuses on resources or objects [4]. It generally behaves as if the client would call a local procedure or service, but instead, it is a remote endpoint. The newer version, gRPC, basically acts the same as RPC with the exception that it uses HTTP 2.0 instead and allows communication streams instead of relying on the request-response pattern [34]. For this reason, gRPC is more popular today than the original concept [4]. On the one hand, it tends to be easy to implement and efficient during runtime, but it also has a steep learning curve in the sense that a developer needs to know all function names and parameters before being able to interact with and use a RPC API. In contrast, with the uniform interface constraint in mind, a developer only needs to know the REST API's resources (URIs) [4, 33].

SOAP The Simple Object Access Protocol was developed by the W3C and designed for exchanging information in a distributed system [61]. Because of its nature as a protocol, it can be quite complex to implement, and it comes with some overhead caused by its rules and restrictions [40]. It has similar (software) quality goals as REST since it also has predefined compliance constraints and enables the extension with Web Service Extensions like *WS-Security* [24] to enforce and support software qualities. That is why SOAP is often used in enterprise scenarios [40] and was a valid alternative to RPC systems before the introduction of REST.

Technology	**Type**	**Message format**	**Interface format**
REST	architecture	any	resource-based
(g)RPC	architecture	JSON, XML	function-based
SOAP	protocol	XML	function-based

Table 3.1: Summary of REST, (g)RPC, and SOAP differences and similarities

Chapter 4

Background

4.1 Software Qualities

Software qualities are defined as the "degree to which the system satisfies the stated and implied needs of its various stakeholders," and can be further described as the "capability of software product[s] to satisfy stated and implied needs when used under specified conditions" [54, Sec. 3.3259][52, Sec. 4.33]. Therefore, they significantly contribute to the overall quality and maturity of a software system.

To help specify and provide a standardized way of evaluating software qualities, one can choose from a variety of Software Quality Models (SQMs). An SQM sets the different qualities into context with each other and might combine multiple qualities into a more general characteristic. Galli *et al.* [16] evaluated a broad selection of SQMs in terms of their relevance to industry and research. They concluded that the ISO standards ISO/IEC 25010:2011 *System and Software Quality Requirements and Evaluation (SQuaRE)* [53] and ISO/IEC 9126 *Software Engineering - Product Quality* [48] have the highest relevance rating and should be selected when looking for an SQM. Singh & Kannojia [47] showed that the most recognized SQMs, such as the models of McCall [11] or Boehm [8], use similar qualities to ISO/IEC 9126. Oppositely, the renowned software engineer Barry Boehm called the standard "particularly weak" in the context of software quality practices [9]. Instead, he referred to the *System Qualities Ontology, Tradespace, and Affordability (SQOTA)* model for mitigating ISO/IEC 25010's flaws. Unfortunately, it is not that popular (yet) and is funded and used by the US military; consequently providing the broad public only restricted access to resources and further material.

4.1.1 ISO/IEC 25010

ISO/IEC 25010:2011 introduces two quality models: The *quality in use model* and the *quality product model*. The former consists of five, and the latter comprises eight properties, of which some are composed of multiple sub-characteristics which represent a set of related qualities [53]:

Quality in use characteristics

Effectiveness
Efficiency
Satisfaction
- Usefulness
- Trust
- Pleasure
- Comfort

Freedom from risk
- Economic risk mitigation
- Health and safety risk mitigation
- Environmental risk mitigation

Context coverage
- Context completeness
- Flexibility

Quality Product Characteristics

Performance efficiency
- Time behavior
- Resource utilization
- Capacity

Compatibility
- Co-existence
- Interoperability

Usability
- Appropriateness recognizability
- Learnability
- Operability
- User error protection
- User interface aesthetics
- Accessibility

Reliability
- Maturity
- Availability
- Fault tolerance
- Recoverability

Functional suitability
- Functional completeness
- Functional correctness
- Functional appropriateness

Security
- Confidentiality
- Integrity
- Non-repudiation
- Accountability
- Authenticity

Maintainability
- Modularity
- Reusability
- Analysability
- Modifiability
- Testability

Portability
- Adaptability
- Installability
- Replaceability

These models enable different stakeholders to define and evaluate software (system) qualities efficiently. The *quality in use model's* quality attributes relate to the user interactions with the finished system, as they help to describe and achieve in-use goals. Hence, the end user is the targeted stakeholder of this model. In contrast, the *quality product model* is set to aid developers, maintainers, and project management with specifying requirements and by helping to assess and evaluate the current project status based on the achieved qualities.

Ghezzi *et al.* [18] state that software qualities also need appropriate measurements to determine the extent to which they were achieved. Measuring software qualities, unfor-

tunately, highly depends on the use case and is - for most scenarios - not generalizable and no generally accepted metrics exist. Nevertheless, there are widely accepted and used measurement methods for some software qualities. For example, *reliability* can be determined by extensive testing and given a clear description of the performance goals, e.g., memory efficiency vs. time efficiency, and *performance efficiency* can also be measured by observing resource usage and execution times.

Further, we define *scalability* as an additional quality, for it is frequently used in the context of distributed systems and their architecture [14, 28]. We characterize it as the combination of *modifiability* and *performance*.

As quality attributes must be considered throughout the entire development process [5], we address them in different places during this work, especially during the evaluation in Chapter 7.

4.2 RFC 6902 - JavaScript Object Notation (JSON) Patch

JSON Patch defines a format to express sequential (partial) changes to a JSON document, e.g., a REST resource while being compatible to be used with the HTTP `PATCH` method [25].

Every JSON Patch document is a valid JSON document containing an array of objects. Each object must hold an operation field `op` and, depending on the operation, a `from`, a `path`, or a `value` field. Possible operations are `add`, `copy`, `move`, `remove`, `replace`, and `test`. Depending on the operation and path, one can modify a specific field (`add`, `move`, `remove`, `replace`) or use its content for copying (`copy`) or validation (`test`) purposes.

Listing 4.1: Example JSON Patch document

```
[
    { "op": "test",    "path": "/a/b/c", "value": "foo" },
    { "op": "remove",  "path": "/a/b/c" },
    { "op": "add",     "path": "/a/b/c", "value": ["foo", "bar"] },
    { "op": "replace", "path": "/a/b/c", "value": 42 },
    { "op": "move",    "from": "/a/b/c", "path": "/a/b/d" },
    { "op": "copy",    "from": "/a/b/d", "path": "/a/b/e" }
]
```

4.3 RFC 7807 - Problem Details

Problem Details, as defined by RFC 7807, aim to make HTTP error response more comprehensive for both human and non-human readers [38]. They are used as either a JSON or an XML document inside the HTTP response's body and provide high-level as well as low-level information about the error occurrence using different fields.

HTTP messages holding problem details either use the `application/problem+json` or `application/problem+xml` media type. A problem detail is composed of the following fields and can be extended with an arbitrary number of custom fields:

- `type`: A URI reference that identifies the problem type, pointing to a human-readable HTML document containing problem documentation.
- `title`: A human-readable summary of the problem type.
- `status`: The HTTP status that fits this problem type.
- `detail`: A human-readable description of the occurrence of this problem.
- `instance`: A URI reference points to the origin path of the problem.

Listing 4.2 shows the body of an HTTP response containing a problem detail if, for an imaginary web service, a document with the identifier 'abc' was requested but not found.

Listing 4.2: Example Problem Detail JSON document

```
{
  "type": "https://example.com/error/doc_not_found",
  "title": "The document could not be found.",
  "status": 404,
  "detail": "Could not find a document with id 'abc'!",
  "instance": "/docs/abc"
}
```

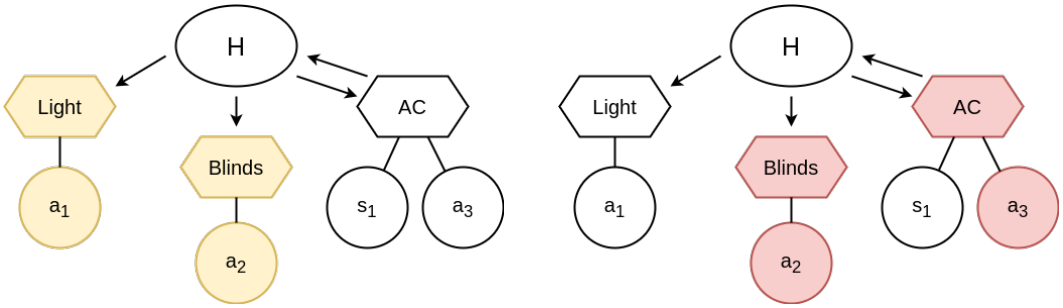
Section 6.3.2 shows how Problem Details are used in this work and what additional fields where added.

Chapter 5

Solution

Given the system model defined in Chapter 2, we propose FooSH (*Framework for outcome-oriented smart homes*): a goal-oriented framework based on a web API to create, configure, and manipulate *Environment Variables* with the help of *Prediction Models* to tackle the described problems and challenges of today's smart home systems. It can be incorporated into an extant smart home system as a non-invasive extension without disrupting it or the need for any modifications. This way, FooSH can be seen as some kind of "parasite" (in a good way), docking onto the existing software without it even noticing any changes.

FooSH allows users to define environment variables to describe phenomena that usually cannot be directly controlled by a smart home and need multiple devices to act in unison to achieve a desired variable state. For this, controllable smart devices, i.e., devices that contain at least one actuator, can be assigned to one or multiple environment variables. It is worth mentioning that a device can be assigned to no, to one, or even to multiple variables as its action might influence none, one, or numerous factors. Looking at Figure 5.1, one can see the device assignments to two variables: *brightness* and *temperature*. Here, the smart home comprises a smart light, smart blinds, and an AC unit.



(a) Assignment of smart devices to *brightness*

(b) Assignment of smart devices to *temperature*

Figure 5.1: Assignment of smart devices to different environment variables

It, therefore, makes sense to assign the light and the blinds to the variable *brightness*, as both their states possibly influence the room's brightness. Moreover, the AC unit and the blinds should be assigned to *temperature*. Since the smart blinds restrict the light coming in through the windows as well as (partly) blocking out the sun, reducing its induced heat on the environment, it can and should be assigned to both variables.

The formal description of a smart home from Chapter 2 can be extended to include environment variables and allow the assignment of smart devices to a set of **environment variables** $V := \{v_1, \ldots, v_i \mid i \in \mathbb{N}\}$, with each variable $v \in V$ containing a subset of relevant smart devices:

$$v_n \subseteq D \tag{5.1}$$

Relevant smart devices are devices that influence the variable in question, when changing their state. As an example, a smart light bulb can be seen as a relevant smart device in the context of a variable representing brightness, for when turning on and off the light, the room's brightness also changes.

Once the user creates a prediction model, it can be integrated and linked to previously defined environment variables. It is important to note that the developer/user has the sole responsibility for the correctness and integrity of the prediction model. The framework only enables its integration into the smart home system and provides corresponding functionalities.

Hence, the framework further extents the existing smart home description with the set of **prediction models**

$$P := \{(T, LV, f, g) \mid LV \subseteq V\} \tag{5.2}$$

with each prediction model $p \in P$ consisting of a target space T, its linked variables LV, a prediction function f, and a translation function g.

- The target space's purpose is to allow developers to define bounds for the variable to allow for more efficient error checking and handling. An example could be the target space $T = [0, 50]$ for an indoor temperature variable.
- The collection of linked variables LV contains the variables on which the prediction model can be applied. Linking a variable to a prediction model is necessary as the risk of errors or unexpected behaviour and results during usage is lowered.
- The prediction function f depends on the used model and, in general, based on the approximated environment and its effects on the smart home and a given target value $t \in T$, it calculates a fitting smart home configuration.
- Using the translation function g, the smart home configuration is translated to a sequence of smart home instructions that can then be executed to eventually reach the previously specified target. The step of translating the configuration into smart home instructions is necessary since we assume the prediction model to have no knowledge of the actual devices in the smart home. It only knows the smart home's

configuration consisting of a number of device states. The model cannot close the gap between the configuration states c_1 - c_j and the corresponding physical smart devices d_1 - d_j. Consequently, it also cannot tell the smart home on how to reach the previously calculated configuration. With the help of g, however, one can map the configuration and its states to the smart devices and generate the according smart home instructions i_1 - i_j. This way, FooSH is able to use the output of any prediction model p and forward its generated instructions to the smart home API.

$$t \xrightarrow{f} \begin{pmatrix} c_1 \\ \vdots \\ c_j \end{pmatrix} \xrightarrow{g} \begin{pmatrix} i_1 \\ \vdots \\ i_j \end{pmatrix} \tag{5.3}$$

The preceding definitions only apply to a smart home wherein the developer/user provided and configured a prediction model and at least one variable, and only if the smart home system has an accessible API. Otherwise, FooSH cannot use its full potential and some functionalities might not be available.

Figure 5.2 shows a prediction interaction with a smart home, with one variable representing brightness. The devices d_1 and d_2 were assigned to said variable and are thus taken into account during the smart home configuration generation process. To start a prediction, the user simply needs to specify a target (here: 600 lumens). Then, FooSH uses the designated prediction model p_1 to determine an appropriate smart home configuration, including the states of the smart device. Finally, it communicates with the original smart home API to change the smart devices' states and ultimately reach the desired goal. Compared to the default interaction depicted in Figure 2.2, the user does not need to set the states of the devices individually but only needs to define the end result instead. The reduction of user interactions and overall cognitive load will become even more evident as the number of relevant smart devices increases.

For the aforementioned system to work correctly, it is assumed that the developer has access to previously collected system usage data, as some prediction methods might require training data or data for data analysis. The data collection could be done by directly accessing the smart home API or listening into the smart home communication channels, to name only two examples. The former approach is used by the proof of concept discussed in Section 7.1. In the end, it is the developer's task to determine a way to collect data - of course, with the selected prediction method in mind. It is further presumed that technically inexperienced users can access a user-friendly UI provided by the developer using the framework's web API.

The proposed framework is mainly designed to be used in the context of smart homes. However, in theory, it can be applied to a broad selection of cyber-physical systems, with variables generalized as environmental factors and devices being actuators affecting these factors.

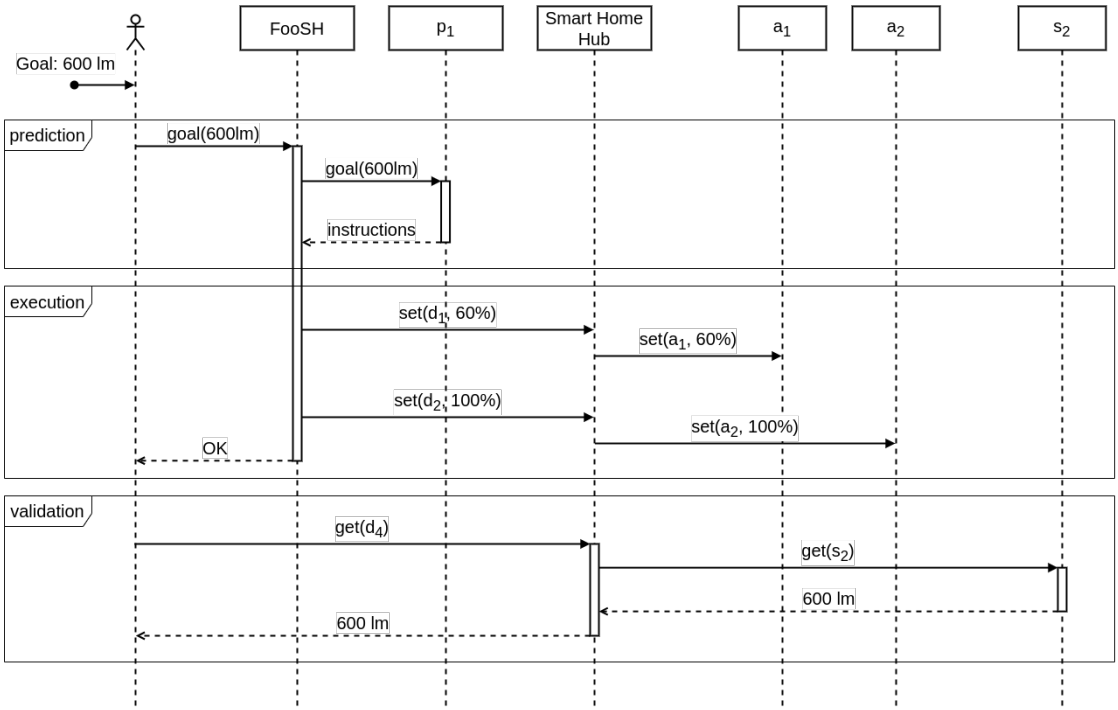


Figure 5.2: Exemplary interaction with FooSH

Chapter 6

Prototype

This chapter covers the design choices and decisions on what technologies, presented in Chapter 3 and Chapter 4 to include in this approach. It also addresses the prototype's architecture and implementation and succinctly gives an outline of how to interact with and utilize the prototype's application programming interface.

6.1 Design

Looking at the advantages of REST towards more sustainable and robust development of software qualities, its architectural constraints, and its popularity presented in Section 3.2, the decision to choose REST as the architecture for the framework, i.e., the framework's web API, was made. The constraints introduced by REST not only impose restrictions on design choices, which might be interpreted as unnecessary or burdensome but also guarantee and improve software qualities, which, again, speaks in favour of implementing a RESTful API.

After considering the advantages and disadvantages of (g)RPC and SOAP over REST, and providing a summary of their properties in Table 3.1, one can say that the decision to choose REST can be considered a reasonable choice. Even though SOAP also makes use of software quality-enhancing practices and extensions, it nonetheless does not suit this use case as it is too restrictive in terms of message formats. Additionally, since SOAP is "just" a protocol, one would still need to find and implement a fitting architecture to go hand in hand with it. Being able to implement a RESTful API and not need to worry about potentially missing extensions or other protocol complexities is a big benefit over SOAP. Like SOAP, RPC also restricts the message format to XML (and JSON) which impedes its *scalability* and overall use cases. Moreover, RPC postulates a client to know about all its functions and parameters a priori which hurts the general *usability* of a system.

After all, REST seems to be the best fit for a modern and sustainable solution for building a web API with the goal of maximizing software quality characteristics.

6.2 Architecture

To get a better understanding of the architecture and not need to rely on classic UML diagrams, the *C4 Software Architecture Model* [46], which introduces four hierarchical diagrams, each with a differently detailed view of the architecture, is used. With this approach, we want to provide adequate and precise depictions of both the entire framework, but also individual sub-parts without sacrificing clarity for the level of detail and vice versa.

The *C4 Software Architecture Model* comprises four diagrams, each diagram being a "zoomed-in" version of a part of the previous one. The first level portrays the context of the application and sets the software in relationship to the user as well as other relevant software systems (see Figure 6.1).

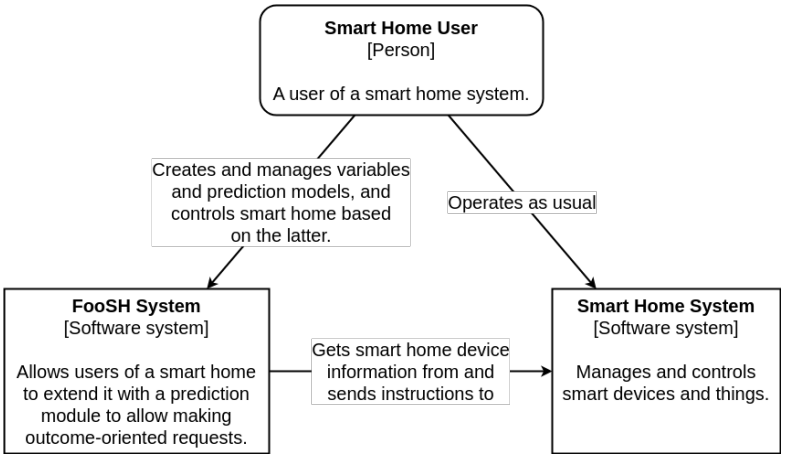


Figure 6.1: C4 Level 1 - Architecture Context Diagram

In this work's context, two additional entities interact with the system under development (FooSH): the *Smart Home User* and the *Smart Home System*. The *Smart Home User* operates the *Smart Home System* as usual, e.g., turning lights on and off, using speech assistants, or creating home automations. Utilizing FooSH, the user can define and manage additional variables and prediction models to extend the existing Smart Home System with outcome-oriented prediction mechanisms. Thus, the user does not need to control individual smart devices or groups. Instead, the user sets a goal value for a previously defined variable and lets a prediction model take over. Subsequently, the prediction model constructs a series of smart home instructions to achieve the user-defined value. Before, the *FooSH System* needs to retrieve smart home-specific information, such as a list of the deployed smart devices, from the Smart Home System using its API to function properly and allow users to link variables and prediction models to specific smart devices.

6.2.1 Spring

As mentioned in Section 6.1, the *REST architecture* from Roy T. Fielding [14] was chosen to design and implement the web API. Based on our experience with and the popularity of Java in the smart home context (e.g., openHAB[1] is entirely Java-based), the decision to select it as the language for this work was made early during the development process. Furthermore, Java supports the prototype's interoperability as the Java Virtual Machine (JVM) detaches Java byte code, i.e., programs written in and compiled with Java, from the underlying hardware - allowing it to be executed on every machine meeting the system requirements [29]. The Spring ecosystem, with its diverse use cases, extensive support, and consideration of software qualities, offered a great opportunity to leverage a pre-existing framework to build upon [59]. In the end, the Spring Web MVC Framework seemed to be the best choice as a foundation for creating the FooSH prototype.

Spring Web MVC

The *Spring Web MVC framework* is a model-view-controller (MVC) framework out of the Spring ecosystem [49, 58]. As the name suggests, it is based on the MVC architectural pattern and helps developers build RESTful applications. Using a `DispatcherServlet`[2] as the *Front Controller*, the framework delegates received requests to specific handlers (controllers) and returns a response (see Figure 6.2). First, the corresponding *Controller* handles the request by executing the necessary code to create a response in the form of a *model* and returns it to the *Front Controller*. There, the response's data is rendered with the help of a *View Template*, using one of Spring's template engines, and returned to the client.

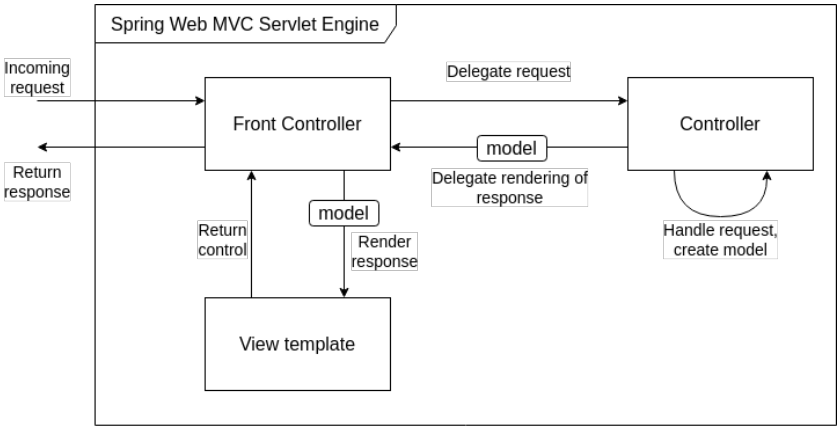


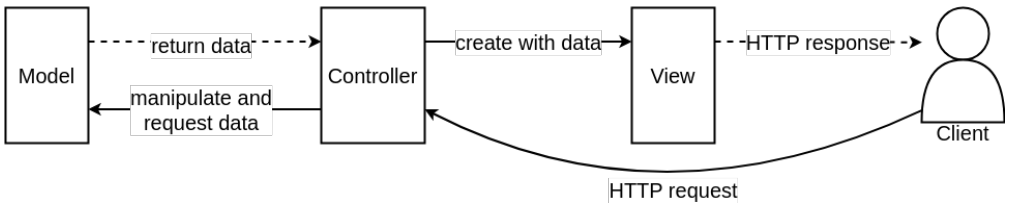
Figure 6.2: Spring Web MVC Servlet Engine [49]

[1] `https://openhab.org`
[2] See the Spring Documentation [50] for more details

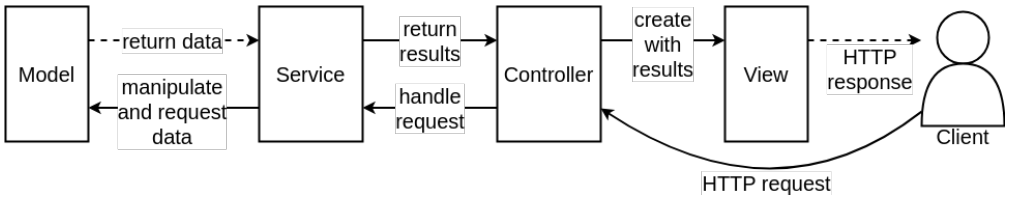
MVCS

To further enforce the separation of concern principle of the framework and promote REST's layered constraint, the MVC pattern is extended with services, resulting in the MVCS pattern. As depicted in Figure 6.3, the Model-View-Controller-Service (MVCS) pattern (adapted for an HTTP web application) adds a *Service* layer between the *Controller* and *Model* layer of the classic MVC pattern. Instead of directly retrieving resource data, the *Controller* now interacts with one or multiple *Services* to handle incoming HTTP requests and creates a *View* based on the *Services*' return values. Here, the *Services* retrieve and manipulate the *Model* by means of resources and their representations, and a *View* consists of a JSON representation of a resource.

Having chosen an MVC architectural pattern and introducing an additional *Service* layer, FooSH not only complies with the Layered System constraint imposed by REST but also contributes to improving its modifiability and portability. The implementation of layers allows the system to be easily changed and adapted to fit different requirements or, if needed, accommodate software/hardware changes while only working on one layer and keeping the other layers untouched and intact.



(a) MVC pattern, adapted from [57]



(b) MVCS pattern

Figure 6.3: MVC vs. MVCS pattern comparison

6.2.2 FooSH Architecture Model

The container diagram displayed in Figure 6.4 further describes the system environment by giving insight into the different responsibilities of various containers and their relationships to and interactions with each other. It becomes clear that the prototype consists of three parts: the *API Application*, the *Prediction Model*, and the *Local Storage*.

API Application Based on a Spring Web MVC application, the API Application provides means for smart home users to manage their outcome-oriented smart home functionalities. These include connecting to and fetching from the smart home system, creating and managing variables and prediction models, and requesting a smart home instruction sequence for accomplishing variable value goals.

Prediction Model For determining a control sequence, the user constructs a prediction model using an appropriate technology for a selection of prediction methods and incorporates it into the API Application. Based on the configured prediction method, it can then be used to determine an adequate smart home configuration to reach a specified value.

Local Storage The API application writes and reads its data to and from the local storage to avoid data loss after restarting the application. Hereby, all information regarding fetched smart devices, configured variables, and implemented prediction models are stored in local persistent storage.

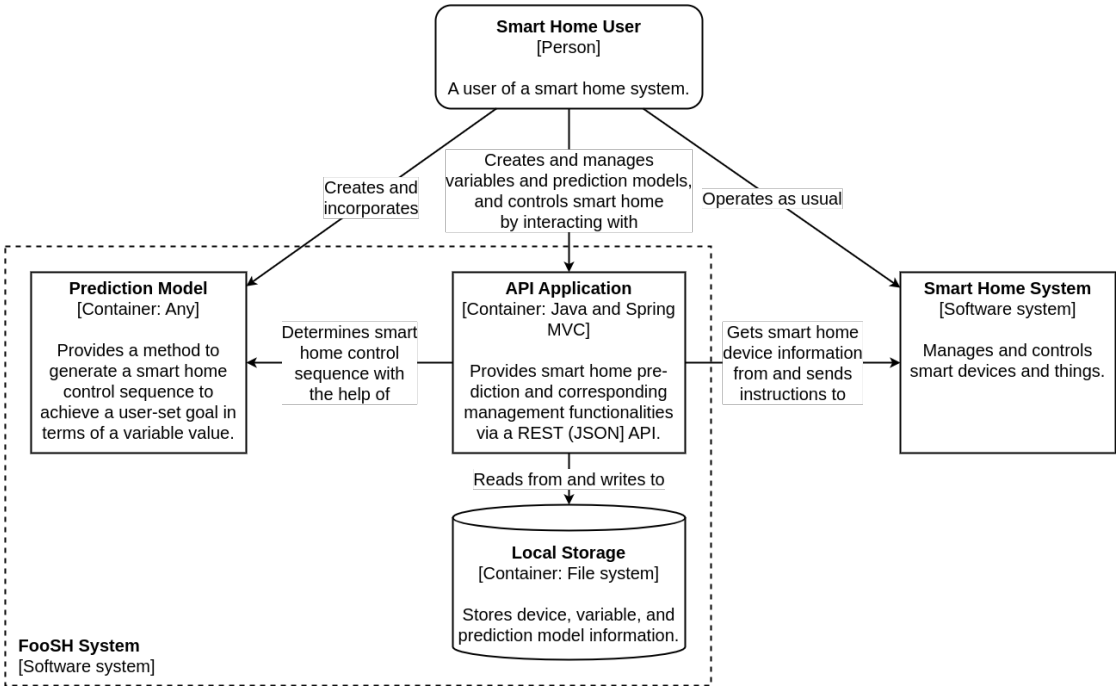


Figure 6.4: C4 Level 2 - Container Diagram

Subsequently, given the architecture container diagram in addition to the constraints imposed by Spring and MVCS, FooSH's high-level architecture and its role in the smart home context can be derived. Figure 6.5 portrays the prototype's general structure. Every HTTP request arrives at the *Spring Web MVC Front Controller* presented in Section 6.2.1 and, depending on the route, delegates it to the corresponding *Controller*. The delegation of requests and the remaining components are structured, with one exception, according to the MVCS pattern. One might notice that the depicted architecture does not include a MVCS *View* (layer). There is no explicit *View* since it is implicitly passed on from the *Service* to the *Controller* and returned to the user in form of a JSON resource representation. Referencing Figure 6.2, one can say, that FooSH just omits the *View template* and returns the somewhat "raw" *model* data.

It includes additional interfaces that are necessary for accommodating

- the communication with the *Smart Home API* for information retrieval and instruction execution, and
- reading and writing framework-relevant data to and from *Local Storage*.

It is important to note that the *Prediction Model* showcased in Figure 6.4 is, from now on, not treated as a standalone container but as a model contained within the MVCS model layer. Thus, when speaking of a prediction model, it should be considered as such if not stated otherwise.

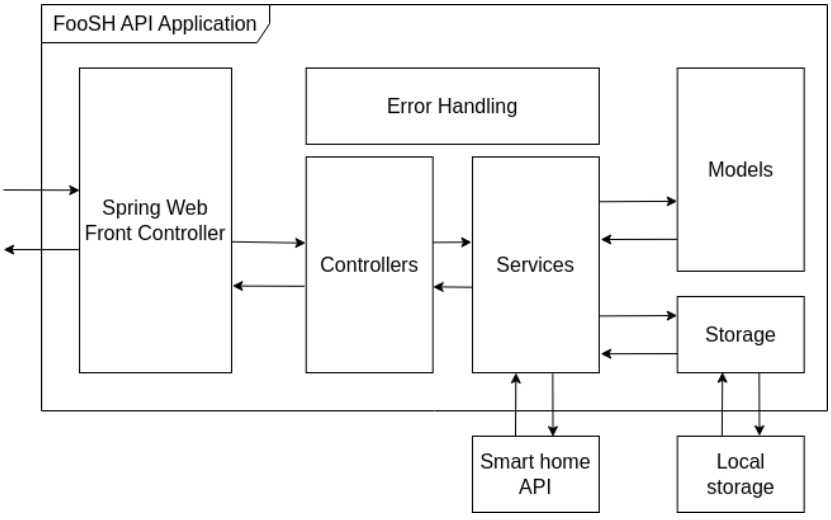


Figure 6.5: High level API application architecture

6.2.3 Components

In the following, with the high-level architecture in place, the different components are defined and characterized. For this, the C4 Component Diagram, as depicted in Figure 6.6, is used. The diagram deviates from an ordinary C4 level 3 diagram, for it is in the form of a UML component diagram to convey the relationships and constraints imposed by Spring Web MVC more clearly. In this format, the component diagram again showcases the extent to which the selected Spring framework provides RESTful functionalities and to what degree pre-implemented classes, interfaces, and structures are used.

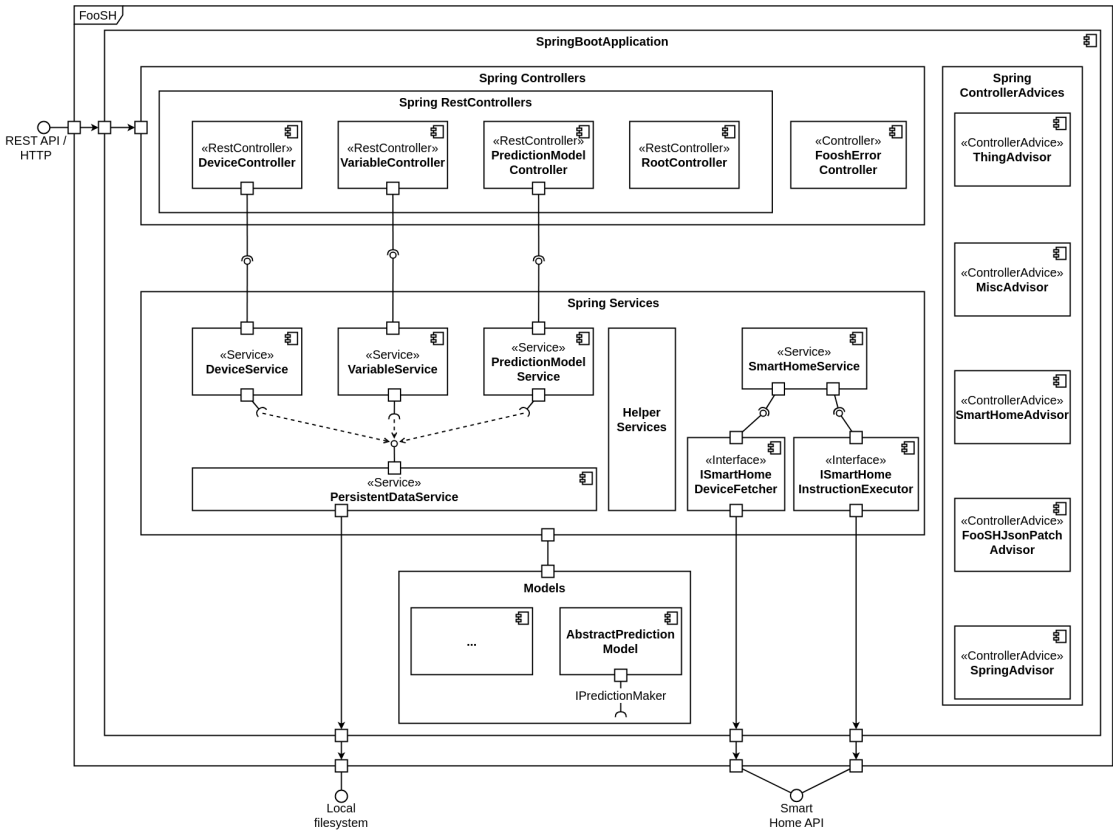


Figure 6.6: C4 Level 3 - Component Model

SpringBootApplication

The entire FooSH application is encapsulated in a *SpringBootApplication*[3], which is necessary to build and invoke a Spring-based software product. It comprises all other (Spring) components, including *Spring Controllers*, *Spring Services*, *(Data) Models*, and *Spring ControllerAdvices*. It also acts as the entry point for any REST API call and HTTP request since it passes them to the *DispatcherServlet* and consequently delegates each request to the corresponding *Spring Rest Controller*.

Spring Controllers

A component within the *Spring Controllers* grouping is either denoted by Spring's `@Controller`[3] or `@RestController`[3] annotation, making it a web controller. Each controller also uses the `@RequestMapping`[3] annotation to define the specific path (e.g., `/api/devices/`) for which it is responsible and capable of handling requests. Given these two annotations, FooSH (Spring MVC) is now able to effectively deal with incoming web requests by delegating them to the corresponding Spring Controller.

DeviceController

The *DeviceController* is a `RestController`, responsible for all requests regarding smart device management. Depending on the used HTTP method, the handling of every request is left to the corresponding *Spring Service*: the *DeviceService*. Lastly, the controller returns the response produced by the service to the client.

The *VariableController* and *PredictionModelController* are defined equally, as they are also `RestController`s and use their respective services, *VariableService* and *PredictionModelService*, to handle requests.

RootController

The *RootController* is responsible for serving requests on the paths `/` and `/api`. It always (for every `GET` request) responds with a list of references to the root paths of the *DeviceController*, *VariableController*, and *PredictionModelController*. This behavior helps to enforce HATEOAS, and it provides a comprehensive as well as informative "landing page" which is - especially for new users - very helpful for navigating the API.

Section 6.3.1 describes HATEOAS and its implementation in this prototype's context in more detail.

[3] See Spring's documentation for more information.

FooshErrorController

Spring Boot provides a default whitelabel error page that, whenever there is an unhandled exception or in case of a `404 Page Not Found` error, is shown. It is overwritten by the *FooshErrorController* to allow for creating a custom error page and custom responses. The latter, in particular, is necessary to make sure that generated responses contain a Problem Detail[4] to conform to the rest of the application's error handling.

Spring ControllerAdvices

Components marked with Spring's `@ControllerAdvice`[3] allow developers to intercept and handle errors. The system around ControllerAdvices is constructed such that if an error is thrown anywhere in the application and, if implemented correctly, instead of needing to handle the error at its source, a function in the corresponding ControllerAdvice is called. This enables centralized error handling, structuring the code more clearly and simplifying future adjustments. Consequently, the system, or in this case, the error handling mechanisms, are found in one place, having the same format and structure, thus ensuring easy access and allowing for straightforward and unified modifications throughout.

Section 6.3.2 discusses the topic of error handling in more detail.

Spring Services

A Spring component annotated with the general-purpose stereotype `@Service` is, in general, considered a Spring Service [45]. The documentation states that a Spring Service has no fixed definition and that (developer) teams may define their own semantics concerning the annotation. In this work, a component annotated with `@Service` is considered a standalone interface, providing specific functionalities throughout the application as part of the MVCS service layer.

DeviceService

The *DeviceService* provides methods to manipulate and extract device information from the list of fetched smart devices, which becomes available after using the *SmartHomeService*.

Again, the *VariableService* and *PredictionModelService* are defined similarly, as they provide functionalities specific to their respective data types: variables and prediction models.

[4] As defined by RFC 7807 [38]

It is important to mention that there are more interactions between services than depicted in Figure 6.6. Except for the exemplary depiction of the *PersistentDataService* interactions, all inter-service interactions are not shown, mainly because of readability reasons.

PersistentDataService

The *PersistentDataService* can be used to write information about the managed devices, variables, and prediction models to persistent storage. After restarting the application, it checks whether there are any saved files. If the service finds previously stored information, it loads it to be used again in the upcoming application session. Writing to and reading from persistent storage detaches the model information from a session, and no data is lost if the server restarts. This not only improves the user experience as the user does not need to configure FooSH after every restart but also reduces the network and smart home API load by avoiding superfluous calls to the API.

SmartHomeService

The *SmartHomeService* manages one instance of an *ISmartHomeDeviceFetcher* and one instance of an *ISmartHomeInstructionExecutor*. Both interfaces communicate with the Smart Home API, where the former provides methods to retrieve smart device information, and the latter forwards and excutes smart home instructions. Having the abstraction of these two interfaces, one is able to connect and communicate with any smart home, as long as it provides an appropriate API.

Helper Services

The prototype contains three additional services, summarized as *Helper Services*:

ListService. The *ListService* manages a list of devices, variables, and prediction models. Other services are designed to use this and only this service to retrieve and manipulate relevant information.

IdService. The *IdService* provides ID- and UUID-related functionalities, like checking whether a given String is a valid UUID.

LinkBuilderService. Constructing the links for HATEOAS-conform JSON responses requires repetitive and frequent constructions of URIs and link collections. The *LinkBuilderService* aims to prevent code repetition and to provide a central interface for creating HATEOAS link (collections).

Models

FooSH contains various data types and models whose description would exceed this thesis' scope. Nonetheless, the most important model, the *AbstractPredictionModel*, is presented in the following paragraph.

AbstractPredictionModel

One of the most crucial parts of the entire framework is the modularity, modifiability, and interoperability in terms of adding and switching between prediction models. For this reason, the *AbstractPredictionModel* is designed in a way that it can be switched out, and links to variables and smart devices can be changed - everything during runtime. It also allows the developer to use and implement arbitrary prediction methods to further increase the range of applicable use cases.

6.3 Implementation

After discussing the design choices and presenting the prototype's architecture, the following sections go into more detail about the actual implementation details and the realization of the preceding prototype specification. One can gain additional insights from FooSH's code base[5] that are not described either due to necessity or space and time constraints.

6.3.1 REST API

With its resource-oriented architecture, REST introduces resources in the form of models in the MVCS model layer, each with its own URI. Table 6.1 displays an overview of the three distinct resources of FooSH: devices, variables, and prediction models. Here, `{id}` is used as a placeholder for a resource identifier.

Every path, with a few exceptions, supports all ordinary HTTP methods in an intuitive way, including `GET`, `POST`, `PUT`, `PATCH`, and `DELETE`, to enforce REST's uniform interface constraint and increase the API's learnability. These exceptions include the root paths and the `PUT` method. The root paths `/` and `/api` only support `GET`, as their purpose is to act as a landing page without any inherent functionality. Even though `PUT` is supported, it is not allowed to be used. Whenever someone tries to send a `PUT` request, a `405 Method Not Allowed` is sent, based on the following two arguments: (i) We wanted the API operation to stay atomic. Therefore, a user is obliged to use `DELETE` and `POST`

[5] `https://github.com/MalteJosten/foosh`

instead to replace a resource. (ii) Sometimes, replacing a resource is not wanted. That is the case for the devices that are externally defined by the smart home or the prediction models that are not (yet) set up to be manipulated after their initial implementation.

Another exception or particularity, to be more exact, is FooSH's behaviour regarding the `DELETE` operation. Whenever the user tries to delete one of the collections, a save file is created (with the additional extension `.old`) before deleting the file stored in persistent storage. This way, the collection's old state can be recovered if necessary. Hence, the prototype becomes more reliable and robust against (erroneous) user behaviour. In addition, this measure also improves the user experience by providing a - to be fair - relatively laborious way of undoing and restoring a deleted collection.

Table 6.1: Overview and description of available FooSH API paths

Path	**Description**
`/`	Redirects to `/api`.
`/api`	Landing page of the API. Gives an overview over the following paths.
Devices	
`devices/`	The collection of smart home devices.
`devices/{id}`	An individual smart home device.
Variables	
`vars/`	The collection of environment variables.
`vars/{id}`	An individual environment variable.
`vars/{id}/devices/`	The collection of an environment variable's linked smart devices.
`vars/{id}/models/`	The collection of an environment variable's linked prediction models.
Prediction Models	
`models/`	The collection of prediction models.
`models/{id}`	An individual prediction model.
`models/{id}/mappings/`	The collection of parameter mappings of an prediction model.

A detailed list of all endpoints[6] with possible HTTP response codes and further request parameters is given in Appendix A.

[6] An endpoint is the combination of a path, e.g., `api/devices/`, and the HTTP request method, e.g., `GET`

FooSH comes with an additional oddity in that it uses a special document type for its HTTP `PATCH` requests: JSON Patch documents.

JSON Patch

Using JSON Patch Documents within HTTP `PATCH` requests allows targeted modifications of individual fields of a resource. However, RFC 6902 does not include any functionalities to restrict access or modification neither for the `op` or `path` fields. Even though there are some Java libraries for applying and executing JSON Patch documents on resources, they also provide no restriction functionalities. But those restrictions are especially important for preserving the external resources' integrity, i.e., that some fields of devices and prediction models remain untouched. And for those fields that can be altered, only a selection of operations shall be allowed to, for example, avoid the deletion of a device's name or crucial information about a prediction model. Simply removing the ability to use HTTP `PATCH` requests was no viable alternative either, for allowing users to make minor changes to the resources during runtime bolsters FooSH's usability and user experience. Therefore, based on RFC 6902, an own handling of JSON Patch documents was written (FooSHJsonPatch), where, depending on the resource, only some paths (resource properties) with selected operations are allowed. The list of affected resources and the associated operations and fields are displayed in Table 6.2.

Table 6.2: Overview of restrictions imposed by FooSHJsonPatch

URI	Path	Operation(s)	Description
`devices/{id}`	`/name`	`REPLACE`	Edit field `AbstractDevice.name`
`vars/{id}`	`/name`	`REPLACE`	Edit field `Variable.name`
`vars/{id}/devices/`	`/`	`ADD`	Add a device-ID to the list to `Variable.deviceIds`
	`/{uuid}`	`REMOVE`	Remove the device-ID {uuid} from `Variable.deviceIds`
`vars/{id}/models/`	`/{uuid}`	`ADD`, `REMOVE`, `REPLACE`	Add/Remove/Replace a link to the model with id {uuid}
`models/{id}`	`/name`	`REPLACE`	Edit field `AbstractPrediction-Model.name`
`models/{id}/mappings/`	`/{uuid}`	`ADD`, `REMOVE`, `REPLACE`	Add/Remove/Replace a link to the variable with id {uuid}

HATEOAS

With HATEOAS (Hypermedia As The Engine Of Application State), the server provides clients (users) with dynamically generated hyperlinks that describe the resource's relations with and to other resources as well as available operations to interact with these resources [19, 31]. This way, the user does not need any a priori knowledge of the prototype's API and should be able to navigate it solely based on the provided hypermedia controls. Implementing HATEOAS contributes to two major points:

1. HATEOAS supports the *Uniform Interface* constraint introduced by Fielding [14] as it realizes, as its name suggests, one of the interface constraints: hypermedia as the engine of application state. The other interface constraints are fulfilled by employing unique identifiers for each resource and passing JSON documents for representing and HTTP methods for modifying and accessing a resource (see Section 6.3.1).
2. HATEOAS is part of the requirement for reaching Level 3 of the Richardson Maturity Model presented in Section 3.2.

Since no standard clearly describes how HATEOAS should be implemented, we implemented the format used by Microsoft in their *Web API Design Best Practices* document [31]. In contrast to the format introduced by Fowler [15], it not only includes the relation name and corresponding URI but also additional fields that inform the user about the available HTTP methods and accepted media types. Listing 6.1 depicts the body of the response to an HTTP `GET` request for retrieving information about the variable *brightness*. In addition to the variable data (provided by the field *variable*), the response includes a *_links* block containing the hypermedia controls. Each link has a *relation* field, describing the resource's relation to other resources with their respective URI provided by the *link* field. The *types* and *action* fields give insight into what media types are accepted by which HTTP method and what HTTP methods are actually available, respectively. This *_links* block, i.e., the hypermedia controls, must be, and are, included in every response message, consequently having a decisive positive effect on the user experience, for it greatly improves the framework's learnability and operability.

Listing 6.1: FooSH HATEOAS Response Excerpt

```
GET /api/vars/brightness HTTP/1.1

HTTP/1.1 200 OK
Content-Type:  application/json

{
    "variable": {
        "id": "56cf5a57-d4bb-465c-8f46-c174921ff35f",
        "name": "brightness",
        "modelIds": [],
        "deviceIds": ["fed5db42-0bf5-44d0-8bfe-89e530a6aa9f"]
    },
    "_links": [
        {
            "relation": "selfStatic",
            "link": "http://localhost:8080/api/vars/56cf5a57-d4bb-465c-8f46-
                c174921ff35f",
            "action": "GET",
            "types": []
        },...
        {
            "relation": "selfName",
            "link": "http://localhost:8080/api/vars/brightness",
            "action": "GET",
            "types": []
        },...
        {
            "relation": "device",
            "link": "http://localhost:8080/api/devices/fed5db42-0bf5-44d0-8bfe-89
                e530a6aa9f",
            "action": "GET",
            "types": []
        },
        {
            "relation": "device",
            "link": "http://localhost:8080/api/devices/fed5db42-0bf5-44d0-8bfe-89
                e530a6aa9f",
            "action": "PATCH",
            "types": ["application/json-patch+json"]
        },
        {
            "relation": "variables",
            "link": "http://localhost:8080/api/vars/",
            "action": "GET",
            "types": []
        },...
    ]
}
```

6.3.2 Error Handling

Drew [13] and Hsieh *et al.* [22] showed that implementing and maintaining sophisticated error handling measures leads to better user experience, increased robustness of the software, and enables thorough debugging and troubleshooting by providing the user with comprehensible feedback and a working software - even when encountering error states. Good error handling also improves the software's reliability by making it more robust against (internal) failures and incorrect use. For these reasons, FooSH implements various error handlers in combination with commonly used and, in general, helpful standards while considering exception-handling goals, as defined by Chen *et al.* [12].

As already mentioned during the description of the *Spring ControllerAdvices* in Section 6.1, Spring enables centralized error management, allowing for a more sophisticated error handling and subsequently enforcing and supporting numerous software qualities. Thus, we further extend the range of improved software qualities by using Problem Details (RFC 7807) (see Section 4.3 for more details) as the standard response for users in reaction to an error. They help to convey more specific information about the cause of a problem or error than the simple HTTP status code that only provides a rough description (in the form of an error code) of what caused an error. Fortunately, Spring already provides a Problem Detail implementation and corresponding methods to integrate them into an existing project [51]. They are designed according to RFC 7807 and are used in the intended way, except for the fact that, in FooSH, the field `type` is left blank due to time constraints, and the field `title` is used to communicate the exception name to allow for better debugging and troubleshooting. It is important to note that the former adaption (empty `type` field, i.e., having the value 'about:blank'), according to RFC 7807, indicates that the problem has no detailed or necessary to mention semantics. Although FooSH uses an empty type field but still provides further information about the problems, it technically violates the standard and confusion could arise. To compensate for the actual specification of the `type`, the `title` field is used as described above. Since the principles of HATEOAS also apply for error responses, FooSH additionally provides a block of links for every problem detail.

The exemplary JSON payload in Listing 6.2 depicts the response for an HTTP `GET` request on the path `api/vars/abc` with no currently defined variables. It correctly states the exception that caused the error (*VariableNotFoundException*, as there is no variable with the name 'abc') as well as the corresponding HTTP status code (`404 Not Found`). The Problem Detail also lists the HATEOAS links to the related resources. Here, the only related resource is the collection of variables on path `api/vars/`.

Listing 6.2: Exemplary FooSH Error Response

```
GET /api/vars/abc HTTP/1.1

HTTP/1.1 404 Not Found
Content-Type:  application/problem+json

{
  "type": "about:blank",
  "title": "VariableNotFoundException",
  "status": 404,
  "detail": "Could not find variable with name 'abc'!",
  "instance": "/api/vars/abc",
  "_links": [
    {
      "relation": "variables",
      "link": "http://localhost:8080/api/vars/",
      "action": "GET",
      "types": []
    },
    {
      "relation": "variables",
      "link": "http://localhost:8080/api/vars/",
      "action": "POST",
      "types": ["application/json"]
    }
  ]
}
```

Using already established standards and technologies offers two main advantages: First, broadly accepted methods are assumed to be sophisticated enough to be used by many people and, therefore, seem to work correctly. It also implies that the chance of the developer needing to learn new principles decreases as they might already be familiar with them, improving the framework's learnability.

The software shows acceptable recoverability and fault tolerance as, if an error occurs, whether while parsing a request or during internal processes, it not only tries to go back to a safe state (if necessary) but also stays operational. In general, FooSH follows the following steps in case of an error:

1. An error occurs.
2. The error is detected and handed to the corresponding Spring advisor.
3. The system is recovered to the most recent safe state, e.g., reversing the action/change that triggered the error.
4. The user is informed about the error and, if available, possible reasons and, as previously mentioned, a link block to conform to HATEOAS is provided.

6.3.3 Testing

Similar to the arguments for error handling, Horgan *et al.* [21] concluded that testing also enhances and reinforces software qualities. Testing supports these SQs as they reveal

the degree to which (i) the software works as intended without showing undesirable side effects, (ii) the error handling is working correctly, and (iii) the software acts predictably and maturely. The ability to thoroughly write numerous (good) tests for a developed software product indirectly proves its testability in the context of maintainability - and thus establishes yet another desired software quality.

Java and Spring form a good starting point for testing purposes as Spring integrates the popular JUnit testing framework[7] to enable unit and integration testing of Spring Boot and Spring MVC-specific features that include but are not limited to Spring RestControllers.

To thoroughly test the developed REST API, 237 integration tests were implemented. They cover all 50 available endpoints, validating the API's responses to as many diverse requests as possible. Here, the process of validating means to compare the actual response with the desired response. If the actual response matches the desired one, the test passes. This way, the general behaviour, the correct error handling as well as the edge cases are tested and covered. It is important to note that one needs to have a working smart home system in place. That is because some tests have to use the smart home API for covering device fetching and smart home instruction execution. However, the smart home API was not mocked due to time constraints, requiring an actual smart home system for the integration tests.

The implemented unit tests (60 in total) primarily cover self-written algorithms, such as the request and name change validations. Unit test coverage is only limited to these methods as functionalities provided by external libraries/frameworks are considered sufficiently tested, and further self-implemented, partly trivial functions were disregarded due to time constraints and necessity.

6.3.4 Developer Guide

This section serves the purpose of addressing technical prerequisites and offering a guide for developers on what classes and functions need to be implemented.

Technical Prerequisites

FooSH is based on a Java 17 Spring Boot 3.1.1 project, using Maven[8] for dependency management. Therefore, a corresponding Java version, e.g., OpenJDK 17 as well as Maven, has to be installed.

[7] `https://junit.org/`
[8] `https://maven.apache.org/`

AbstractDevice

The abstract class `AbstractDevice` needs to be implemented to fit the needs of the smart device representation enforced by the smart home, potentially augmenting the implementation with additional smart home-specific data. Such data may include its name or API URL.

AbstractPredictionModel

For now, the prototype does not administer the ability to create a prediction model during runtime. Instead, the developer has to define and implement a class realizing the `AbstractPredictionModel`. FooSH does not restrict the implementation of the custom prediction model classes, meaning that the developer has complete freedom in their decision-making. Prediction-making can therefore either be done internally as a part of FooSH or externally by calling and utilizing external code. The `AbstractPredictionModel` implements the `IPredictionMaker` interface, which provides the `makePrediction()` method used for accepting the prediction's goal value and returning a list of smart home instructions for achieving that value. It defaults to returning an empty instruction list, resulting in leaving the smart home as it is, effectively not reacting to any requests. If the developer wishes for appropriate handling of requests and finding viable smart home configurations, they are advised to overwrite `makePrediction()`, customizing it to their prediction model's demands.

ISmartHomeDeviceFetcher and ISmartHomeInstructionExecutor

FooSH's connection to the smart home system significantly depends on the pre-existing smart home API techniques, necessitating the implementation of the `ISmartHomeDeviceFetcher` and `ISmartHomeInstructionExecutor` interfaces. Without their implementations, FooSH is neither able to gather any smart device information needed for its operation nor to apply any changes to the smart home.

6.4 Usage

With everything set up, i.e., a defined and incorporated prediction model, and after "connecting" FooSH to the already operating smart home API, one is able to interact with FooSH's web API, configuring, managing, and employing environment variables to alter the smart home's state in a goal-oriented manner. To fully utilize FooSH and invoke their first goal-oriented request, a user is required to follow a minimum of five steps:

Step 1: Fetch smart devices

First, the user needs to gather the currently active smart devices from the smart home API by submitting a `POST` request to `/api/devices/` with the option to provide further details, e.g., authentication credentials, if needed.

```
POST /api/devices/ HTTP/1.1
Accept:  application/json

{
    "details": {
        "token": "foo",
        "user": "bar"
    }
}
```

Step 2: Define environment variable

Then, the user has to create an environment variable with a `POST` request on `/api/vars/`, given a variable name that is not yet used, e.g., *brightness*.

```
POST /api/vars/ HTTP/1.1
Accept:  application/json

{
    "name": "brightness"
}
```

Step 3: Assign device(s) to variable

In the third step, the user assigns an arbitrary number of smart devices to the previously created environment variable with `POST /api/vars/{id}/devices/`. The request's body contains the list of IDs of the devices that can be retrieved with `GET /api/devices/` and should be assigned to the *brightness* variable.

```
POST /api/vars/devices/ HTTP/1.1
Accept:  application/json

{
    "deviceIds": [
        "91e0b147-3e7b-4e7a-90ca-f392453937a9",
        "4314c0b2-749d-447f-b807-6984556486a8"
    ]
}
```

Step 4: Link variable with prediction model

The last thing to do before being able to initiate a prediction is to link an environment variable with a prediction model. This way, the user defines the mapping of the input parameters of the prediction model to the actual smart devices of the environment variable. A variable can be linked to a prediction model by either sending an HTTP `POST` request to `/api/vars/{id}/models/` or `/api/models/{id}/mappings/`.

```
POST /api/model/my-model/mappings HTTP/1.1
Accept:  application/json

{
    "variableId": "35d6461e-a3fd-4d45-ab80-92657d4a9f06",
    "mappings": [
        {
            "parameter": "x1",
            "deviceId": "91e0b147-3e7b-4e7a-90ca-f392453937a9"
        },
        {
            "parameter": "x2",
            "deviceId": "4314c0b2-749d-447f-b807-6984556486a8"
        }
    ]
}
```

Step 5: Make a prediction

Posting an HTTP `POST` to `/api/vars/{id}`, the user can finally kick off a prediction process for an environment variable. In the example below, the user utilizes the previously linked prediction model *my-model* to generate a smart home configuration that attains a brightness of 50 (%). Since the field *execute* is marked as `true`, the generated configuration is converted into appropriate smart home instructions, which are sent to the smart home API, actually changing the smart home's state, thereby achieving the desired brightness of 50 (%).

```
POST /api/vars/brightness HTTP/1.1
Accept:  application/json

{
    "modelId": "my-model",
    "value": 50,
    "execute": true
}
```

Once an environment variable and prediction model are established, they are continuously available for further requests and determining corresponding smart home configurations, assuming a consistent smart home composition.

Chapter 7

Evaluation

The evaluation first addresses a proof of concept to determine the prototype's functionality and the degree to which it applies to established smart home systems. It then deals with the prototype itself: how does it answer or solve the research questions presented in Section 2.3, and which benefits or limitations arise when applied to a smart home system.

7.1 Proof of Concept

The following proof of concept was conducted to show the functional suitability of FooSH. It was set to be based on the desire to automate the brightness control on a desk to allow for reliably compensating for different external factors and giving the smart home user a consistent experience.

As mentioned before, using FooSH without a prediction model defies its purpose. Hence, the first step in preparing to deploy FooSH in an existing smart home system is to construct an appropriate prediction model. For properly constructing a prediction model, one needs some data to base the prediction method on. That is why a measurement setup was built to capture randomly generated smart home usage data. The next section deals with the details of the measurement setup, and Section 7.1.2 describes the acquired data that is used by the selected prediction model discussed in Section 7.1.3. The last two parts deal with the implementation of the prediction model using the FooSH framework and evaluating the concept's validity.

7.1.1 Setup

The existing smart home runs on openHAB and comprises a desk lamp and a Shelly Smart Plug S[1], which allows for controlling the light (turning it on and off). At the time of carrying out the proof of concept, no smart lamp was available. For this reason,

[1] `https://kb.shelly.cloud/knowledge-base/shelly-plus-plug-s-1`

the smart plug and lamp are used as replacements for one smart lamp to yield the same results. Consequently, limited to this scenario, the prediction model's goal is to determine when and under what circumstances the smart plug, i.e., the lamp, should be turned on, given a desired brightness value at the desk.

In order to collect (artificially) generated data, a *Measurement Hub* and *Measurement Station* were used to (i) randomly switch the smart plug on and off, (ii) record the resulting brightness at the desk, and (iii) collect the captured values in combination with the corresponding smart home configuration. The physical setup is depicted in Figure 7.1, showing the placement of the lamp in relation to the measurement station *MS*, as well as the source of natural light: a wide window front to the right side of the desk.

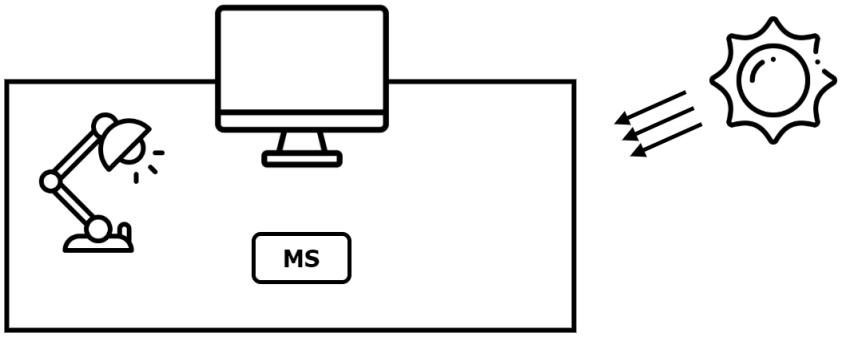


Figure 7.1: Data Collection and Measurement Setup

Measurement Hub

The measurement hub is a Raspberry Pi 4B[2], running two programs, fulfilling the following two purposes:

1. It controls the smart plug, turning the plug on and off at random to generate extensive and varied data.
2. It hosts and acts as a collection point for the measurement station to send its measurements to and stores them, in combination with the corresponding smart home configuration, in a MongoDB database.

The script for changing the smart plug's state communicates with the openHAB API, and every 120 seconds, it, based on a PRNG (pseudo-random number generator), either turns the plug on or off, simultaneously turning on or off the light as well. Whenever the hub receives new data points from the measurement station, it fetches the smart devices' states from the openHAB API and stores all gathered information (brightness

[2] `https://www.raspberrypi.com/products/raspberry-pi-4-model-b/`

measurement, smart home configuration, and timestamp) in a MongoDB database. This way, one can later use the measurement data to create and potentially train a prediction model.

Measurement Station

The measurement station deals with the problem of capturing the brightness levels present at the desk. A common, even though not reliable, solution requires a light dependent resistor, also known as photoresistor. Given the fact that a photoresistor changes its resistance depending on the amount of light hitting the sensor and by applying voltage to the resistor, one can measure the light intensity, i.e., brightness, based on the voltage being passed through.

With the circuit shown in Figure 7.2, one can capture the brightness in one area in a room. The circuit consists of a Raspberry Pi Pico W[3], a $10k\Omega$ resistor, and a photoresistor. All components were mounted on a breadboard according to the schematic, resulting in the conversion of the analog value generated by the photoresistor to a digital 16-bit signal at the corresponding GPIO pin. The sampling rate was, according to the Nyquist-Shannon sampling theorem, set to 60 seconds. This allows the Raspberry Pi Pico to successfully capture the brightness as a digital value, allowing it to be sent to the measurement hub for further processing.

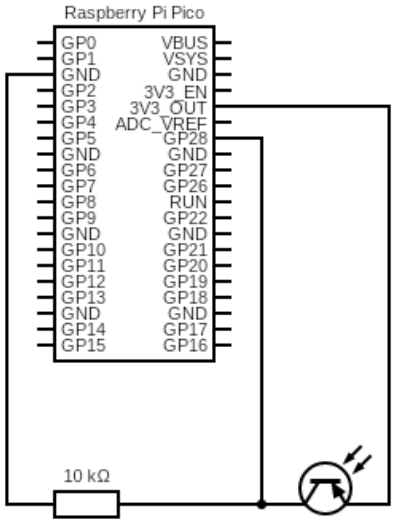
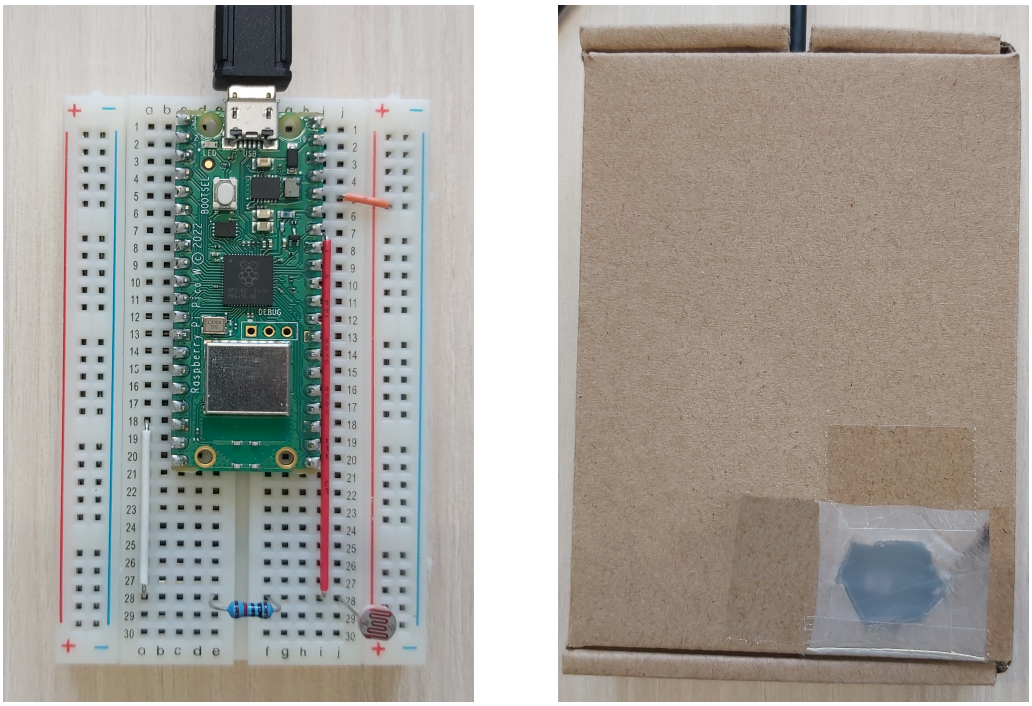


Figure 7.2: Measurement Station Circuit Schematic

[3] https://www.raspberrypi.com/products/raspberry-pi-pico/

One major problem when measuring brightness with a photoresistor is the fact that the resistor only captures the light punctually. To allow for more ambient light to be detected by the resistor, one can typically use specifically produced hardware, like Fresnel lenses. They collect light from a broader area and focus it on one point: the photoresistor. Since there was no Fresnel or similar lens available, and since the design and construction of a proper case for the measurement station would have exceeded the time constraints for this work, an intermediate solution was applied. This solution includes putting the breadboard with all the wired-on hardware into an opaque container, only leaving out a hole for the resistor, covered in a diffused material (see Figure 7.3). The semi-transparent material acts as a diffuser to smooth out the captured light and improve the blending of different light sources.



(a) Breadboard measurement station

(b) Measurement Station with light diffusion cover

Figure 7.3: Measurement Station Setup

For further implementation details, either regarding the hub or the station, one can look into the code in the corresponding Github repository[4].

[4] `https://github.com/MalteJosten/foosh-measurement`

7.1.2 Acquired Data

The collected data is stored in a MongoDB database, a NoSQL database. Listing 7.1 presents the content of an exemplary MongoDB document (without irrelevant MongoDB-generated fields). It holds a `timestamp`, encoded in UTC in milliseconds, representing the moment when the measurement was taken. Thus, it is necessary to keep this in mind for future reproductions of the experiment or usage of the collected data, e.g., training a prediction model. It additionally stores the measured brightness value in the field `value`, and the corresponding smart home configuration, i.e., the states of all smart devices, is stored in the `items` array. The measurements are stored in their original "raw" form, to enable future adaptions to make use of the unmodified values. This maximizes the potential for a range of possible new use cases and applications.

Listing 7.1: MongoDB Entry

```
{
  "timestamp":  1692343421703,
  "value": 53517,
  "items": [
    {
      "link": "http://192.168.108.103:8080/rest/items/plug1",
      "state": "ON",
      "editable": true,
      "type": "Switch",
      "name": "plug1",
      "label": "shelly-plug-1",
      "tags": ["Office"],
      "groupNames": ["Office"]
    },
    {
      "link": "http://192.168.108.103:8080/rest/items/foo",
         :
      "groupNames": ["Office"]
    },
    ...
  ]
}
```

The brightness measurement experiment ran for three consecutive days, and its result is plotted in Figure 7.4. Here, we differentiate between data points with the light switched on (yellow) and data points with the light switched off (blue). This way, one can clearly see the differences in the light affecting the brightness - especially at night. Based on the setup depicted in Figure 7.1, it is predictable that the lamp has nearly no effect on the overall brightness during the day. That is because the room is mainly illuminated by sunlight, and the lamp's light only becomes noticeable after sunset. The variations during the day are caused by different weather conditions (sunny, cloudy, etc.). As previously mentioned, the brightness is converted to a 16-bit integer number, resulting in a possible value range of 0-65535. This range is therefore used as the y-axis of the plot.

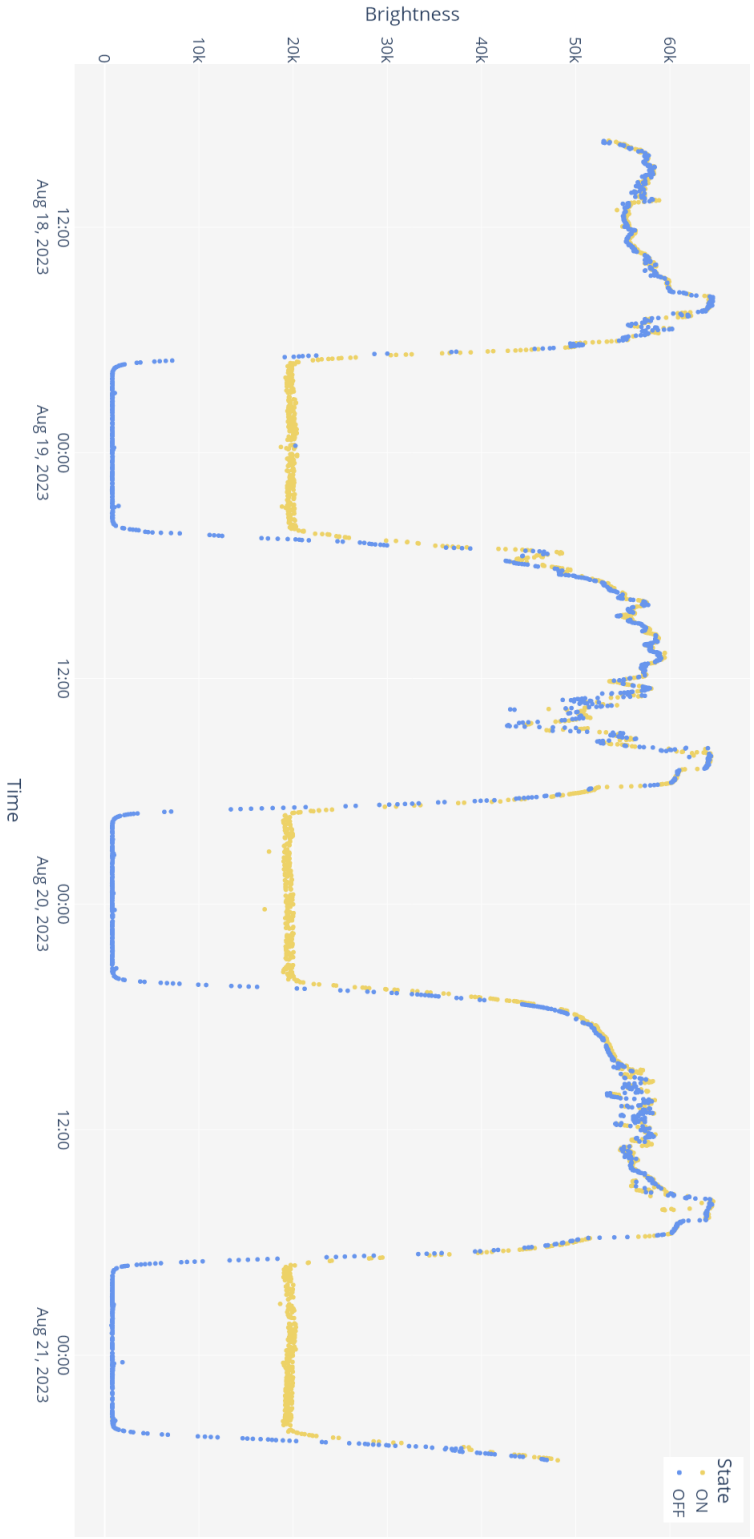


Figure 7.4: Raw brightness measurement data

7.1.3 Prediction Model

The approaches presented in Chapter 3 introduced various prediction techniques and showed only a small selection of possible uses for (popular) machine learning or logical methods. However, not every prediction method fits every problem. We decided that there is no need for a complex prediction method as developing approximation functions can be done by hand and suffices for the purpose of this simple proof of concept. For this to work and enable a more intuitive user experience, the initial value range of 0-65535 was mapped to 0-100. This way, the user can define a more natural goal in the form of providing a brightness percentage.

The first step was to divide the data set into two distinct data sets: one containing only brightness values for when the lamp was turned on (ON-State data) and a second one with only brightness values for when the lamp was turned off (OFF-State data). Then, each data set was smoothed using an average sliding window to discard any outliers and possible measurement inaccuracies. In addition, this made the subsequent steps easier. Figure 7.5 shows the smoothed graph for the OFF-State data.

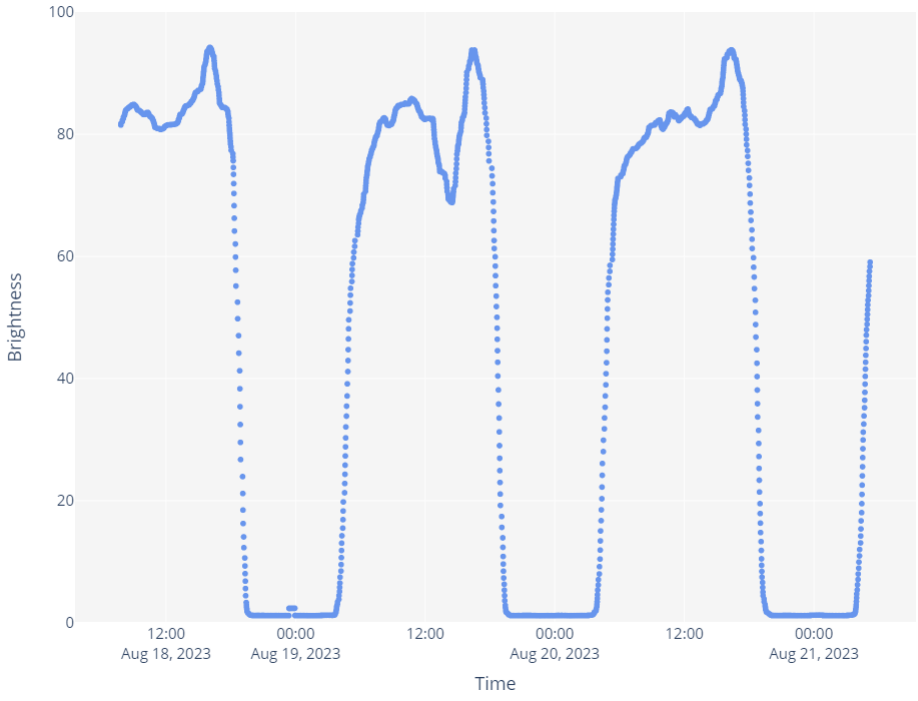


Figure 7.5: Smoothed, scaled OFF-State data

Next, the measurement data was divided into 24-hour sections, ranging from 00:00 to 23:59. Using a sophisticated prediction technique like symbolic regression would have also required dividing the data into multiple sections to be able to effectively determine an approximation for a day's worth of brightness. But in contrast to simple approximations (lower degree polynomial functions), symbolic regression would have needed further training without guaranteeing accurate results. Thus, if a simpler solution also yields sufficient results, it is unnecessary to consult complex methods. We defined the different sections by looking at the entire graph, identifying points at which the graph's trend noticeably changes during each 24-hour day. The divisions and resulting intervals are marked in Figure 7.6 with black lines, overlaying the actual brightness values of the third day for comparison.

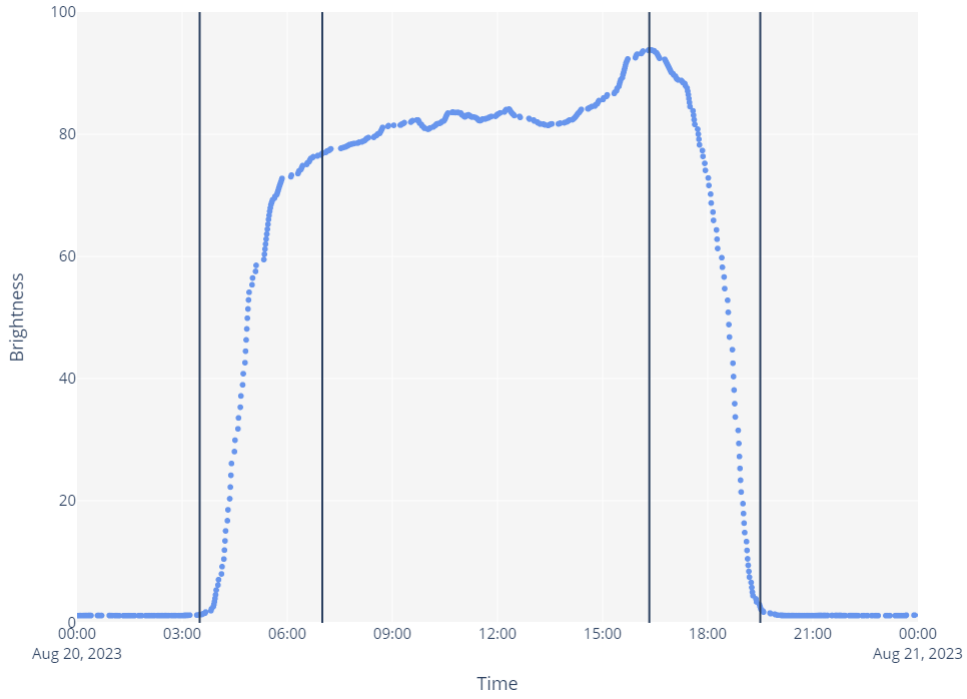


Figure 7.6: OFF-State data with marked intervals

The third step was to determine the polynomial functions that give a sufficient approximation of the actual brightness curve for an entire day. The interval boundaries defined in the preceding step were used as the x-values for finding the points (x_n, y_n) needed for deriving the respective function. Since one can derive a linear (quadratic) function given two (three) points (x_1, y_1), (x_2, y_2), (x_3, y_3), the average y-value, i.e., brightness value for each known interval boundary, was calculated using Table 7.1. Each row of the table represents one interval boundary, with the columns *Time* and t containing the time of the day in military time and the time of the day in minutes, respectively. The following four columns, *Day 1* to *Day 4*, describe the brightness value on each day at the time t, and their average is summarized in the last column.

The average values were then used as the y-values of each point, completing each (x, y) tuple, i.e., a Time-Brightness pair. Subsequently, with the help of the now fully defined pairs, the approximation functions f_1 - f_5 and g_1 - g_5, depicted in the *Function* column of Table 7.2, could be derived.

Table 7.1: Interval bound values estimation

Time	**t**	**Day 1**	**Day 2**	**Day 3**	**Day 4**	**Average**
OFF-State						
19:30	1170		1.85	1.73	2.41	2.00
00:00	0		1.21	1.19	1.27	1.22
03:30	210		1.28	1.35	1.29	1.31
07:00	420		76.98	76.93		76.96
16:20	980	92.88	93.74	93.81		93.48
ON-State						
19:37	1177		29.18	28.52	28.46	28.72
00:00	0		28.95	28.53	28.74	28.74
03:45	225		29.14	28.91	29.26	29.10
07:15	435		79.53	77.83		78.68
15:55	955	93.03	89.00	93.34		91.79

Table 7.2: Approximation function derivation

Interval	Start Value	End Value	Function
OFF-State			
00:00 - 03:30	1.22	1.31	$f_1(t) = 1.25$
03:30 - 07:00	1.31	76.96	$f_2(t) = -0.00171541t^2 + 1.44095238t - 225.64$
07:00 - 16:20	76.96	93.48	$f_3(t) = 0.0295t + 64.57$
16:20 - 19:30	93.48	2.00	$f_4(t) = -0.00253407t^2 + 4.96678116t - 2340.2428$
19:30 - 00:00	2.00	1.22	$f_5(t) = -0.0029t + 5.38$
ON-State			
00:00 - 03:45	28.74	29.10	$g_1(t) = 0.0016t + 28.74$
03:45 - 07:15	29.10	78.68	$g_2(t) = -0.00108262t^2 + 0.95054677t - 129.9650$
07:15 - 15:55	78.68	91.79	$g_3(t) = 0.0252t + 67.7130$
15:55 - 19:37	91.79	28.72	$g_4(t) = -0.00127972t^2 + 2.44427603t - 1075.3518$
19:37 - 00:00	28.72	28.74	$g_5(t) = 28.72$

Finally, depicted in Figure 7.7, the derived approximation functions can be used as a composite function for the 24-hour brightness estimation. These functions are equal to the following function f_{approx} and describe the brightness with the light turned off[5], using the previously derived sub-functions:

$$f_{approx}(t) = \begin{cases} f_1(t) & , 0 \leq t \leq 210 \\ f_2(t) & , 210 \leq t \leq 420 \\ f_3(t) & , 420 \leq t \leq 980 \\ f_4(t) & , 980 \leq t \leq 1170 \\ f_5(t) & , 1170 \leq t \leq 1439 \end{cases}$$

[5]The function g_{approx} was defined equally.

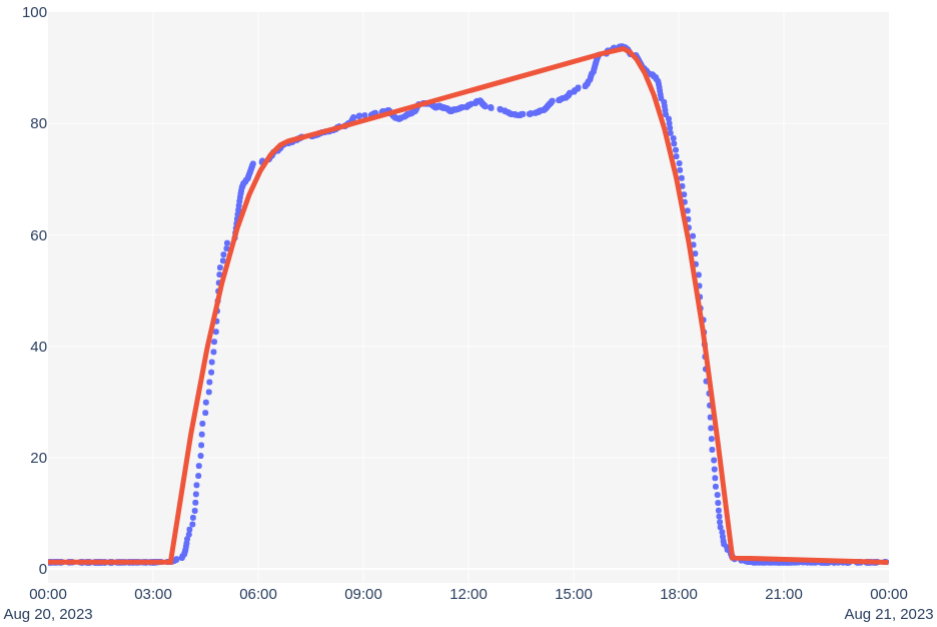


Figure 7.7: OFF-State data with approximation functions

With the approximation function in place, the idea to use f_{approx} as a threshold for the decision to turn on the light emerged. The resulting algorithm acts in a way that every brightness value request that contains a brightness less than the approximated brightness (using f_{approx}) can be met without turning on the light. Consequently, for every request exceeding the OFF-State brightness value, the lamp is turned on. This way, the prediction model always tries to ensure reaching the desired goal with the least effort, i.e., only turning on the light when it is really necessary.

7.1.4 Implementation

First, a *Device* data type, inheriting the *AbstractDevice* class, was defined to include all necessary information provided by the smart home that is needed for this use case. To successfully fetch the smart devices from the OpenHAB API and send and execute smart home instructions, the interfaces *ISmartHomeDeviceFetcher* and *ISmartHomeInstructionExecutor* were implemented, respectively.

Finally, based on the already presented prediction model, the *AbstractPredictionModel* was used to create a custom prediction model class, including the prediction function *makePrediction()*, shown in Algorithm 1. It takes a target value v and calculates, based

on the time of the day, the current brightness level (with the smart lamp turned off) using f_{approx}. If the computed value is less or equal to the target value, the smart home instruction(s) to turn the lamp off is generated since the ambient light apparently suffices to achieve the desired brightness. However, if the target value exceeds the prediction value, the instruction for turning on the lamp is generated. In the end, the function *makePrediction()* produces a sequence of smart home instructions (in this case, only containing one instruction) to modify the smart home, consequently achieving the previously specified target value.

Algorithm 1 Prediction making and smart home instruction generation

```
procedure makePrediction(targetValue : v)
    instr ← [ ]
    t ← time
    if f_approx(t) ≤ v then
        instr ← instr + turn_light_off
    else
        instr ← instr + turn_light_on
    end if
    return instr
end procedure
```

7.1.5 Validation & Evaluation

Various tests were conducted to validate the implementation and correctness of the custom prediction model and its incorporation into FooSH, as well as FooSH's integration into the existing smart home system. These tests covered two different parts:

1. Test whether the smart home could be properly integrated and interacted with.
2. Test whether the prediction model yielded satisfying results, i.e., serve its purpose.

The former was tested by starting the application and retrieving the list of devices. The resulting list matched the list of smart devices present in the smart home that were also registered in openHAB, essentially affirming the correct smart device retrieval. Validating the instruction execution was done in combination with the latter part by inducing multiple prediction requests and comparing the resulting smart home behaviour with its expected behaviour (either turning on or off the smart lamp). All requested predictions were handled correctly and produced a satisfying result, causing, if necessary, an appropriate change in the smart home to reach the initially specified target brightness value. Here, *satisfying results* are results that achieve the desired outcome with minimal effort, i.e., only turning on the light if really necessary.

It is important to note that the presented prediction model introduces seasonal inaccuracies, as the measurement period only covered three days in August. Therefore, the collected data and the resulting approximation function(s) should not be applied for prediction-making in, e.g., January, as the weather and, most importantly, the brightness levels during the day vary immensely. The approximation function would always yield a brightness level of 93.48 at 16:20 (see Table 7.1). This value may be correct for an August day but not for a day in January, as during winter, it usually gets darker sooner, and the natural light emitted by the sun is much lower. Hence, the validation only applies to a (short) timeframe after the initial measurement period.

In conclusion, the developed prediction model sufficiently approximates the original brightness (graphs). Compared to the original brightness measurements, it exhibits an MSE (Mean Squared Error) of 29.422 and an MSLE (Mean Squared Logarithmic Error) of 0.081. Given these metrics and just by looking at Figure 7.7, one can make out some inaccuracies, which, however, do not impede the approximation function's/prediction model's meaningfulness, as humans are unlikely to notice minor brightness differences. Therefore, slight deviations in the approximation by the prediction model relative to the ground truth (represented by the measurements) are negligible, at least for this scenario. Nonetheless, prediction models should be thoroughly validated and evaluated in more detail, especially regarding the question of how accurate the approximation needs to be and how accurate it actually is before deploying them with FooSH. As described previously, the developed prediction model struggles with approximating seasonal data. In general, the biggest challenges for prediction models that deal with smart home sensor data seem to be capturing and imitating the degree of seasonality or periodicity, as various factors are time-, weather-, or season-dependent. Additionally, the proof of concept shows deficits regarding adaptability to new circumstances, like changes in the smart home setup. Looking at the literature, FooSH is not the only goal-oriented approach that deals with the challenge of the smart home's flexibility.

7.2 Discussion

In the following discussion, we first cover and discuss how and to which extent the research questions introduced in Section 2.3 were answered. Then, FooSH's limitations and drawbacks when deploying in a real-world scenario are highlighted and finally compared to similar approaches from Chapter 3.

7.2.1 Research Questions

(RQ 1) How to deal with and connect to a wide variety of smart home systems?

As described during the initial solution proposal in Chapter 5, the prototype is designed so that it docks onto the existing smart home without it noticing. Deploying FooSH does not disturb the running system and does not require any alterations or modifications of the existing smart home system. If successfully integrated into the smart home, FooSH provides several interfaces to retrieve the smart home's device information and execute commands, essentially giving the user the possibility to control smart devices with FooSH as a proxy. The provided (Java) interfaces (see Section 6.3 for further details) ensure that FooSH has access to all the integral information for its proper operation (list of smart devices and the capability to alter the smart home's state, i.e., controlling a smart device). They additionally present the developer with nearly total freedom for tailoring them to their individual needs and ensuring their compatibility with, in theory, every smart home system[6].

Different smart homes have different ways of representing a smart device. To accommodate the various device representations, FooSH has an *AbstractDevice* class that includes the necessary information for every device and provides a uniform interface with functions to access its data. Implementing a class that inherits the *AbstractDevice* class forms the base for further adaptions, depending on the developer's and the system's needs (see Section 7.1.4 for an exemplary implementation of a custom device class).

One can say, based on the preceding measures implemented by FooSH, that the exact type of connection to the extant smart home is left to the developer, as FooSH merely administers the environment by defining indispensable functions. This environment guarantees the incorporation of FooSH in the smart home system without interfering with or restricting either the smart home's or FooSH's operability and functionality. FooSH's operability and other software qualities are evaluated in great detail in the following section dealing with research question RQ 3.

[6] It is debatable whether FooSH can be used within **any** smart home. Further tests or studies are required.

(RQ 2) How to deal with and allow incorporation of different prediction models?

The developed prototype allows the incorporation of pre-trained and ready-to-use prediction models, which can utilize an arbitrary prediction technique, determining a smart home configuration that achieves a specific environment variable value. Here, the only requirement for a prediction model is to eventually return a sequence of smart home instructions (which could be empty) that change the smart home's state and its devices, ultimately reaching the previously determined smart home configuration and achieving the desired goal.

FooSH provides features to theoretically link multiple variables to one prediction model. As discussed before, it may seem unreasonable to use one prediction model for different environment variables. At last, the user is left with the decision to make use of this feature since, in some scenarios, they possibly still want to employ it. This way, FooSH aims to improve its reusability and interoperability of components (here: prediction models and variables) without restricting the user and their potential intentions and needs.

The requirements for a solution (stated in Section 2.2) also have the objective of dealing with and solving potential conflicts between prediction methods or, to be more exact, between their respective prediction-making and the resulting smart home configurations. FooSH has no explicit measures to mediate or solve said conflicts. Instead, it processes multiple prediction requests sequentially. This way, two (or more) prediction processes can not directly interfere with each other, and a prediction request is handled based on the configuration induced by the previous one. Thus, the responsibility for dealing with conflicts is transferred to the user, increasing the user's cognitive load and consequently restraining the user experience. These impediments contrast with the actual goals of smart homes: improving comfortability and being convenient.

In summary, the prototype maintains features for including various prediction models and linking them to environment variables. They can then be used to determine a smart home configuration that achieves a certain user-defined environment variable value. These features can be used flexibly, and their implementations make them open for future adjustments or extensions. However, there is still a need for improving conflict resolution to sustainably support the user experience and allow for sophisticated handling of potentially conflicting variable goals.

(RQ 3) Which software qualities are crucial for developing a sustainable and useful solution? What can be done to accomplish and enforce selected software qualities?

Multiple SQMs were presented in Chapter 4, from which ISO/IEC 9126 seems to be a promising software quality model but as ISO/IEC 25010 is its revised successor, we will choose the former as the software quality model for this thesis. Despite the critique of

Boehm, we stick to our decision of choosing ISO/IEC 25010. Additionally, the restricted access and low popularity of SQOTA do not meet our requirements for a software quality model, and it is therefore not an option.

In addition to narrowing it down to primarily only considering one model, we further want to point out the relevant quality attributes of the quality product model for this work. Considering that the framework only connects the smart home with a prediction model, provides a REST API (see Section 6.2 for more architectural details), and is the result of the first step of becoming a sophisticated framework for outcome-oriented smart home systems, we want to omit the qualities *performance efficiency* and *security*.

In the following, FooSH is evaluated individually regarding every quality defined by ISO/IEC 25010, including the additional quality scalability. But since it was developed as a prototypical framework that shall be extended by other software engineers and smart home enthusiasts, we are paying more attention and devoting more time to the product quality attributes and only succinctly discuss the quality in use attributes.

Functional Suitability The *functional suitability* was thoroughly covered in the preceding section. It showed that the developed prototype fulfills the goals defined in Section 2.2 by providing a selection of implementable interfaces to connect to the existing smart home system, and its architecture allows developers to assimilate various prediction models.

Performance Efficiency The individual implementation(s) of the prediction model(s) are the crucial parts concerning *performance efficiency* and are not part of the framework itself. They are developed and inserted by the developer who uses the framework and consequently outside our control and this evaluation. However, since FooSH itself is built with Spring which naturally pays close attention to performance, it also exhibits a similarly high degree of all-round performance [59].

Compatibility Due to its nature of being an extension, FooSH does not interfere with the existing smart home system and only extends and utilizes already available smart home services. Hence, it is able to co-exist with other (software) systems. Additionally, FooSH's RESTful web API allows other programs and users to access its functionalities, and the generalized smart home interfaces also strengthen the overall *interoperability* with external systems.

Usability FooSH's code base is thoroughly documented, supplying (new) developers with detailed descriptions of how FooSH works internally and can be used. Both the popularity of REST and the information provided by HATEOAS help counteract the initial hurdle of starting with FooSH. The simplicity of and familiarity with the CRUD operations (except for JSON Patch), likely originating from interacting and using other web APIs, further constitutes the prototype's overall *operability*. If the developer is unfamiliar with web APIs or is inexperienced in working with RESTful APIs, the error handling presented in Section 6.3.2 ensures sufficient *user error protection*. To aid the prevention of inducing further errors, future work on

the prototype should address the missing Problem Detail error descriptions (contained in field `type` and discussed in Section 6.3.2), consequently offering detailed descriptions and potential causes for a raised exception/error.

Reliability Based on the following various factors, the *reliability* of FooSH comes out to be quite sophisticated. FooSH's *fault tolerance* is supported by the expansive error handling capabilities described in Section 6.3.2 and validated by the extensive test suite depicted in Section 6.3.3. Not permanently deleting, e.g., a device list from FooSH's persistent storage, but instead keeping it around for later recovery positively impacts the *recoverability* and allows a user to restore (falsely) deleted data. The same is true for the use of JSON Patch, as erroneous JSON Patch documents are validated before execution and, if necessary, the safe state of a resource is re-established. In case some exceptions are not covered by the prototype's error handling, Spring tries to ensure the application's *fault tolerance* and *availability* with its internal workings that also detect errors and act accordingly by sending error responses to the client without sacrificing the overall *reliability*.

Security Mainly due to time constraints, features and mechanisms to accommodate *security* were considered out of scope for this thesis. Nonetheless, security is especially crucial in the context of smart home systems as the user's privacy and data need to be protected. It therefore has to be addressed in future matters when working with and expanding FooSH.

Maintainability Different factors play into the evaluation of the *maintainability* characteristic. Because of this, Table 7.3 gives a brief overview of said factors in regard to the sub-characteristics they influence the most. The separation of concerns and the demarcation between components induced by REST's layered constraint as well as the MVCS architectural pattern benefits all sub-characteristics and the *maintainability* as a whole. The second REST constraint that improves the software's *maintainability* is the uniform interface constraint, as it, based on its unitary (web) interface, allows for fast and straightforward modifications along with simplifying its *analysability* and *testability*. In addition, the implementation details discussed in Section 6.3 also show that the implemented (abstract) classes and Java interfaces make easy reuse and adaption possible. Moreover, the centralized exception handling presented in Section 6.3.2 simplifies and ultimately enhances FooSH's *analysability* and *modifiability* since it is all in one place, similarly designed and built. Lastly, Spring provides mechanisms to effectively test the developed web API and ordinary code using the popular testing framework JUnit. As Section 6.3.3 showed, every API endpoint was successfully tested, and further code was covered by standard unit tests. We presume that the not yet tested parts could, if necessary, also be easily tested.

Portability FooSH's *adaptability* leaves something to be desired, for when the smart home hardware changes (e.g., adding a new smart light), existing prediction models need to be re-trained/re-generated using newly collected usage data. With the current design, the prototype does not know about any changes occurring in the

Table 7.3: Factors affecting FooSH's maintainability

	Modularity	**Reusability**	**Analysability**	**Modifiability**	**Testability**
REST layered constraint / MVCS	✓	✓	✓	✓	✓
REST uniform interface constraint			✓	✓	✓
FooSH implementation details		✓		✓	
Spring/FooSH error management			✓	✓	
Spring/Java tests					✓

smart home without manually fetching the device list. It is assumed that if someone fancies installing and using FooSH, a working smart home system is already set up. The only two prerequisites left to fulfill to successfully install and ultimately utilize FooSH are to (1) provide a platform that is capable of running the Java application and (2) set up at least one prediction model. Depending on the previously employed system, it should be relatively easy to replace an existing software solution with FooSH as it does not interfere with the smart home system. Additionally, based on its properties for providing a web API, its interface controls are straightforward, easy to understand and interact with. Furthermore, FooSH exhibits significant potential for the range of possible areas of application (see Chapter 5), rendering it deployable not only for smart homes but for various use cases.

Scalability The choice of the REST architecture and RESTful web API promotes the scalability of the developed system since the imposed constraints considerably emphasize *scalability* [14]. Spring also offers various mechanics and best practices to support scaling Spring Boot applications and associated software [44].

Effectiveness The prototype allows for querying prediction requests, but it has no direct influence on their results since the results heavily depend on the employed prediction model. Additionally, FooSH provides interfaces for executing a sequence of smart home instructions, realizing the generated smart home configuration, and achieving the initial goal value. Nonetheless, the generation/prediction process still lies outside the prototype's scope. Consequently, its *effectiveness* cannot reliably be determined without an explicit prediction model on hand. The framework's *effectiveness* therefore significantly depends on the use case, i.e., the utilized prediction model.

Efficiency Determining a product's *efficiency* sets its *effectiveness* in relation to the needed resources. Thus, the same limitations for ascertaining the *effectiveness* also apply to the *efficiency*. However, if the prototype's software qualities are evaluated regarding a specific use case, the following factors could be considered as resources: the time it takes to carry out the prediction, consumed hardware and

software resources, the number of required user interactions to reach a goal, or the cognitive load of the user, i.e., how much does the user have to think about the correct way for achieving their goal.

Satisfaction, Freedom of Risk, Context Coverage To be able to reasonably assess the degree of *satisfaction*, *freedom of risk*, and *context coverage*, further studies and use cases are needed. That is because, as the name *Quality of Use* suggests, one needs actual developer and user feedback for a proper evaluation. Therefore, it is advisable to let other developers employ and integrate FooSH into real-world smart home systems. With the help of their experiences and collected user feedback, one should be able to make meaningful statements regarding the foregoing software qualities. Until then, the question of the qualities' sophistication unfortunately has to be postponed.

Since software qualities are not the only factor playing a role in determining if a software product is sustainable or usable, we also want to evaluate FooSH's maturity regarding its web API. We think this is another relevant point for the overarching evaluation as a badly implemented web, or in this case, RESTful API negatively affects the entire product's sustainability and usability. For this reason, we want to classify the implemented RESTful API that was described in Section 6.3.1 using the WS^3 Maturity Model to primarily assess its maturity (which gives a good indication of its sustainability and usability) but also to create better and more accurate comparability with other or future approaches. As described in Chapter 4 (and depicted in Figure 3.4), the WS^3 Maturity Model uses three dimensions for classifying a web API, which, in the case of FooSH, leads to a categorization of D4-S2-P1.

For the first dimension - the design dimension - the prototype fulfills its requirements for the fourth level as they are analogous to the Richardson Maturity Model in which FooSH's web API reaches the fourth and highest maturity level. That is the case, as the API, as thoroughly described in Section 6.2 and Section 6.3, exhibits all the necessary properties. For one, it uses more than a single URI since it manages various resources with distinct API endpoints (Level 0, Level 1), and allowing the client to call it using any HTTP method (Level 2). Moreover, with the implementation of HATEOAS and providing hypermedia controls, the API satisfies the preliminaries for Level 3.

The implemented REST API or, to be more exact, the HATEOAS messages not only semantically describe each resource data by the nature of JSON documents but also the resource's relationships with the provided link blocks in each response, as described in Section 6.3.1. Thus, FooSH fulfills the prerequisites for the maximum *Semantic Dimension*: *Linked Data*.

In contrast, FooSH only reaches the first level of the *Profile Dimension* (*Interaction Profile*) since the link blocks provided by the API's HATEOAS messages include a description regarding every available option, such as HTTP methods or accepted media types (see Section 6.3.1 for more details on the implementation of HATEOAS and its

usage in FooSH). However, the API does not provide any instructions in what order calls need to be done to, for example, rename an environment variable nor describe pre- or post-conditions for the different features.

7.2.2 Limitations

Looking at the prototype's architecture, its implementation, and the preceding evaluation of the research questions, a few limitations and restrictions arise when deploying FooSH in a real-world system. The following list contains the most impactful and restricting limitations and drawbacks.

(L1) FooSH is a framework intended to be used as an extension for an existing smart home. Because it is a framework, it still needs to be programmatically integrated and configured by other developers or smart home enthusiasts. Thus, it is unsuitable for out-of-the-box deployment and requires a somewhat technically skilled person to be set up.

(L2) As of now, the prototype provides no features for creating or training prediction models during runtime. It is only capable of working with already pre-trained and implemented prediction models. This limitation can be explained by the fact that smart homes are considered part of the IoT domain, where computationally limited hardware is most commonly deployed. It is therefore not advisable to use this hardware, for example, to train a CNN, which should then be part of a prediction model.

(L3) The current state of the framework does not allow for considering home automations that might interfere or aid with reaching a user-specified environment variable value. FooSH only takes into account the smart devices "as they are" and disregards any automations or (time-based) rules.

(L4) As discussed during the software quality evaluation, FooSH lags in terms of adaptability. In case something changes inside the smart home, e.g., if a smart device is added or removed, the integrity of a prediction model or environment variable cannot be guaranteed as an originally relevant device could be removed, unbeknown to FooSH. Even if such a change is detected, both the affected environment variable(s) and the prediction model(s) must be adapted accordingly. Looking at (L2), this cannot be done during runtime, as the prediction model might need some form of re-training or similar modifications.

(L5) If a prediction model produces a smart home configuration and the corresponding instructions, the user has, except for toggling the `execute` flag, no opportunity to give feedback or react to the result. They consequently cannot influence or adapt the prediction model's outcome in a reliable or user-friendly way.

7.2.3 Comparison to Related Work

The creators of the ACHE system [32] probably also recognized the problem of simulating or predicting the entire smart home environment, including its external factors, for which they, equally to FooSH, distinguish between different application domains (environment variables), effectively subdividing the aforementioned problem. Both approaches also require re-training if the smart home setup changes, as they depend on pre-configured policies or prediction models. However, FooSH does not restrict its users to use one of the policies, or in this case, prediction models. Instead, it allows for any model to be integrated and used to reach a specific environment variable value.

With *Sasha* [27], King *et al.* introduced a sophisticated and flexible framework for smart home assistants, especially concerning its engaging feedback loop. Unfortunately, it also limits its use cases as it is purely developed with vocal interactions and its role as a voice assistant in mind. Additionally, unlike FooSH, it does not provide a universal interface by which it could be integrated into and employed in even more contexts. We nonetheless need to admit that *Sasha*'s weakness concerning compatibility, induced by its restrictions as an NLP-using voice assistant, is simultaneously one of its advantages over FooSH, as it can handle natural language instructions (coming from the smart home user). On the contrary, FooSH can only work with discrete and factual instructions, such as "Set Brightness to 80%," and fails to comprehend or even accept instructions like "Help me wake up." Looking at (L5), one could use the authors' approach for allowing user-induced feedback as inspiration for adding feedback features for FooSH - eliminating a major limitation regarding FooSH's usability and user-friendliness.

With *DGOC*, Palanca *et al.* introduce a quite complex and invested to set up, goal-oriented architecture for smart homes. This complexity might stem from their intentions of transforming their system into a working operating system. Another factor might be the need to define all the agents together with their services and plans, and link them to the physical (smart) devices. The need to provide (and implement) functionalities for extracting and detecting the goal from possibly colloquial user interactions, as their approach assumes that the already extracted goal is used as its input, plays another vital role in that context. Looking at it this way, FooSH provides a more user-friendly system that is more straightforward to set up. With its lack of adaptability described in (L4), FooSH lags behind the *DGOC* architecture as it is capable of reconfiguring and adapting itself to different circumstances, like removing or adding smart devices. Unfortunately, the paper does not go into detail on how this self-adaptability is reached, necessitating further research into this topic.

The declarative programming approach by Bisicchia *et al.* [7] presents a promising solution for the challenge of solving conflicts between multiple goals and their respective solution "plans." This approach exhibits sophisticated principles with which it is able to deal with conflicts way more efficiently than FooSH. FooSH only avoids said conflicts

by processing them sequentially, essentially alleviating potential discrepancies, whereas their framework approaches it by using mediation policies to find a satisfying solution for all parties. Therefore, it should be considered to integrate Bissichia's framework into FooSH to make use of the provided conflict resolution, consequently transforming FooSH to be more resilient and user-friendly.

In summary, one can say that although FooSH shows some limitations and deficits, it still deals with and answers all research questions appropriately, contributing to the current status quo of smart home research. Contrasting the presented related work, FooSH allows the integration of arbitrary methods, be it machine learning methods, methods based on artificial intelligence models, or service-oriented selection mechanisms (as in [36]). Additionally, the related literature mostly uses their various "prediction" methods to primarily mediate between or find a fitting execution plan instead of using observed user behavior. FooSH therefore makes use of data that does not need to be explicitly generated, making the manual work of filling some form of knowledge base obsolete.

Chapter 8

Conclusion

We set out to design, create, and implement a software product that fills the current gap in universally applicable goal-oriented smart home systems to bring back the initial user-centric goals of comfortability and convenience. For this reason, a prototype was developed that fulfills the requirements for a sustainable and long-lasting goal-oriented smart home framework by exhibiting distinctive software qualities while paving the path for future extensions and modifications. It also showed that the design decisions and its architecture, especially the implemented interfaces, allow developers to integrate the framework into the majority of state-of-the-art smart home systems and use FooSH with an arbitrary prediction method, i.e., goal-resolution mechanism. We constructed a proof of concept that demonstrates FooSH's ability to be deployed into an existing smart home system as well as the simplicity with which new prediction models can be incorporated and utilized.

Even though the prototype has a few limitations and imposes some restrictions regarding its adaptability to changes in the smart home, the current state of the framework can, nonetheless, be seen as a successful attempt to contribute to the current research field of smart homes. It represents a good starting point for further research and expansion to further improve the user experience and satisfaction with smart homes.

8.1 Future Work

A possible approach to remove the limitations described in (L2) could introduce a feature set that allows clients to create and, if necessary, train prediction models during runtime. One would need to investigate in more detail whether the local hardware suffices for potentially complex and resource-intensive operations or if outsourcing the training process to a cloud service, for example, would yield satisfying results while exonerating the native system.

Being one of the major weaknesses, the goal of increasing the framework's adaptability is self-explanatory. Consequently, one could try to integrate adaptability-improving features from the related work and evaluate whether they actually improve FooSH's

adaptability or if further research needs to be done and new mechanisms need to be developed.

Implementing a user feedback loop poses another interesting starting point for further improvements of the framework, as the user can now actively influence and correct the computed smart home configuration and instructions. The work of King *et al.* [27] could be used as inspiration as it already introduced a sophisticated and evaluated interaction cycle that allows the smart home user to interact with the system, taking initiative in influencing the process of achieving the user goals.

Security is a crucial factor in smart homes as it deals with sensitive user data, pointing out the importance of data privacy and protection. Since security aspects were omitted for now, one of the next steps in expanding FooSH should be to implement and consider security measures to protect smart home users and their data. It should not be possible for unauthorized parties to, for example, access the smart home details, and they should not be able to cause changes in the smart home or control smart devices. Because of this, one should research viable options and evaluate whether they apply to the smart home context without negatively affecting the user experience.

Appendix A

REST API Endpoints

Table A.1: List of REST API entpoints for route `/api/devices/...`

Method	Description	Header field(s)	Accepted Mediatype	Response Code(s)
`/api/devices/`				
`GET`	Retrieve the list of smart home devices.	-	*	`200 OK`

`POST`	Fetch the list of registered smart devices from the smart home API.	-	`application/json`	`201 CREATED`
				`409 CONFLICT`
				`415 UNSUPPORTED MEDIATYPE`
				`500 INTERNAL SERVER ERROR`
				`502 BAD GATEWAY`
				`504 GATEWAY TIMEOUT`

`PUT`	Method is not allowed.	-	*	`405 METHOD NOT ALLOWED`
`PATCH`	Method is not allowed.	-	*	`405 METHOD NOT ALLOWED`
`DELETE`	Delete the local list of smart home devices.	-	*	`200 OK` `500 INTERNAL SERVER ERROR`
⋮	⋮	⋮	⋮	⋮

Table A.1: List of REST API entpoints for route `/api/devices/...`

Method	Description	Header field(s)	Accepted Mediatype	Response Code(s)
`/api/devices/{id}`				
`GET`	Retrieve the smart home device with the ID or name `id`.	-	*	`200 OK` `404 NOT FOUND`
`POST`	Method is not allowed.	-	*	`404 NOT FOUND` `405 METHOD NOT ALLOWED`
`PUT`	Method is not allowed.	-	*	`404 NOT FOUND` `405 METHOD NOT ALLOWED`

`PATCH`	Change the name of a device using a Json Patch Document.	-	`application/json-patch+json`	`200 OK`
				`400 BAD REQUEST`
				`404 NOT FOUND`
				`409 CONFLICT`
				`415 UNSUPPORTED MEDIATYPE`
				`500 INTERNAL SERVER ERROR`

`DELETE`	Method is not allowed.	-	*	`404 NOT FOUND` `405 METHOD NOT ALLOWED`

Table A.2: List of REST API entpoints for route `/api/vars/...`

Method	Description	Header field(s)	Accepted Mediatype	Response Code(s)
`/api/vars/`				
`GET`	Retrieve the list of defined smart home variables.	-	*	`200 OK`

`POST`	Create and name a/multiple new smart home variable(s).	`batch=true|false`	`application/json`	`201 CREATED`
				`400 BAD REQUEST`
				`409 CONFLICT`
				`415 UNSUPPORTED MEDIATYPE`
				`500 INTERNAL SERVER ERROR`

`PUT`	Method is not allowed.	-	*	`405 METHOD NOT ALLOWED`
`PATCH`	Method is not allowed.	-	*	`405 METHOD NOT ALLOWED`
`DELETE`	Delete the list of defined smart home variables.	-	*	`200 OK` `500 INTERNAL SERVER ERROR`
`/api/vars/{id}`				
`GET`	Retrieve the smart home variable with the ID or name `id`.	-	*	`200 OK` `404 NOT FOUND`

`POST`	Generate (and execute) a set of smart home instructions to reach a given value for the variable with the ID or name `id`.	-	`application/json`	`200 OK`
				`400 BAD REQUEST`
				`404 NOT FOUND`
				`415 UNSUPPORTED MEDIATYPE`

`PUT`	Method is not allowed.	-	*	`405 METHOD NOT ALLOWED`
⋮	⋮	⋮	⋮	⋮

Table A.2: List of REST API entpoints for route `/api/vars/...`

Method	Description	Header field(s)	Accepted Mediatype	Response Code(s)

`PATCH`	Change the name of the variable with the ID `id` using a Json Patch Document.	-	`application/json patch+json`	`200 OK`
				`400 BAD REQUEST`
				`404 NOT FOUND`
				`409 CONFLICT`
				`415 UNSUPPORTED MEDIATYPE`
				`500 INTERNAL SERVER ERROR`

`DELETE`	Delete the variable and all its links with other *things* with the ID or name `id`.	-	$*$	`200 OK`
				`404 NOT FOUND`
				`500 INTERNAL SERVER ERROR`

`/api/vars/{id}/devices/`				
`GET`	Retrieve the list of smart home devices that are linked to the smart home variable with the ID or name `id`.	-	$*$	`200 OK` `404 NOT FOUND`

`POST`	Link a list of devices to the variable with the ID or name `id`.	-	`application/json`	`200 OK`
				`400 BAD REQUEST`
				`404 NOT FOUND`
				`409 CONFLICT`
				`415 UNSUPPORTED MEDIATYPE`
				`500 INTERNAL SERVER ERROR`

`PUT`	Method is not allowed.	-	$*$	`405 METHOD NOT ALLOWED`
⋮	⋮	⋮	⋮	⋮

Table A.2: List of REST API entpoints for route `/api/vars/...`

Method	Description	Header field(s)	Accepted Mediatype	Response Code(s)

`PATCH`	Modify the list of linked devices for the variable with the ID `id` using a Json Patch Document.	-	`application/json-patch+json`	`200 OK`
				`400 BAD REQUEST`
				`404 NOT FOUND`
				`409 CONFLICT`
				`415 UNSUPPORTED MEDIATYPE`
				`500 INTERNAL SERVER ERROR`

`DELETE`	Delete the list of devices that are linked to the variable with the ID or name `id`.	-	$*$	`200 OK`
				`404 NOT FOUND`
				`500 INTERNAL SERVER ERROR`

`/api/vars/{id}/models/`				
`GET`	Retrieve the list of linked prediction models to the smart home variable with the ID or name `id`.	-	$*$	`200 OK` `404 NOT FOUND`

`POST`	Link a prediction model to the variable with the ID or name `id`.	-	`application/json`	`200 OK`
				`400 BAD REQUEST`
				`404 NOT FOUND`
				`409 CONFLICT`
				`415 UNSUPPORTED MEDIATYPE`
				`500 INTERNAL SERVER ERROR`

`PUT`	Method is not allowed.	-	$*$	`405 METHOD NOT ALLOWED`
⋮	⋮	⋮	⋮	⋮

Table A.2: List of REST API entpoints for route `/api/vars/...`

Method	Description	Header field(s)	Accepted Mediatype	Response Code(s)

`PATCH`	Modify the list of linked prediction models for the variable with the ID `id` using a Json Patch Document.	-	`application/json-patch+json`	`200 OK`
				`400 BAD REQUEST`
				`404 NOT FOUND`
				`409 CONFLICT`
				`415 UNSUPPORTED MEDIATYPE`
				`500 INTERNAL SERVER ERROR`

`DELETE`	Delete the list of prediction models that are linked to the variable with the ID or name `id`.	-	*	`200 OK`
				`404 NOT FOUND`
				`500 INTERNAL SERVER ERROR`

Table A.3: List of REST API entpoints for route `/api/models/...`

Method	Description	Header field(s)	Accepted Mediatype	Response Code(s)
`/api/models/`				
`GET`	Retrieve the list of pre-defined prediction models.	-	*	`200 OK`
`POST`	Method is not allowed.	-	*	`405 METHOD NOT ALLOWED`
`PUT`	Method is not allowed.	-	*	`405 METHOD NOT ALLOWED`
`PATCH`	Method is not allowed.	-	*	`405 METHOD NOT ALLOWED`
`DELETE`	Method is not allowed.	-	*	`405 METHOD NOT ALLOWED`
`/api/models/{id}`				
`GET`	Retrieve the prediction model with the ID or name `id`.	-	*	`200 OK` `404 NOT FOUND`
`POST`	Method is not allowed.	-	*	`405 METHOD NOT ALLOWED`
`PUT`	Method is not allowed.	-	*	`405 METHOD NOT ALLOWED`

`PATCH`	Change the name of the prediction model with the ID `id` using a Json Patch Document.	-	`application/json-patch+json`	`200 OK`
				`400 BAD REQUEST`
				`404 NOT FOUND`
				`409 CONFLICT`
				`415 UNSUPPORTED MEDIATYPE`
				`500 INTERNAL SERVER ERROR`

`DELETE`	Method is not allowed.	-	*	`405 METHOD NOT ALLOWED`
⋮	⋮	⋮	⋮	⋮

Table A.3: List of REST API entpoints for route `/api/models/...`

Method	Description	Header field(s)	Accepted Mediatype	Response Code(s)
`/api/models/{id}/mappings/`				
`GET`	Retrieve the list of parameter mappings and linked smart home variables for the prediction model with the ID or name `id`.	-	*	`200 OK` `404 NOT FOUND`

`POST`	Link a variable with corresponding parameter mappings to the prediction model with the ID or name `id`.	-	`application/json`	`200 OK`
				`400 BAD REQUEST`
				`404 NOT FOUND`
				`409 CONFLICT`
				`415 UNSUPPORTED MEDIATYPE`
				`500 INTERNAL SERVER ERROR`

`PUT`	Method is not allowed.	-	*	`405 METHOD NOT ALLOWED`

`PATCH`	Change one or multiple parameter mappings and linked variables using a Json Patch Document.	-	`application/json-patch+json`	`200 OK`
				`400 BAD REQUEST`
				`404 NOT FOUND`
				`409 CONFLICT`
				`415 UNSUPPORTED MEDIATYPE`
				`500 INTERNAL SERVER ERROR`

`DELETE`	Delete the list of parameter mappings and variables that are linked to the prediciton model with the ID or name `id`.	-	*	`200 OK`
				`404 NOT FOUND`
				`500 INTERNAL SERVER ERROR`

Appendix B

ISO/IEC 25010 Software Quality Descriptions

Table B.1: Product Quality [53]

(Sub-)Characteristic	Description
Functional Suitability	
Functional Completeness	The degree to which the set of functions covers all the specified tasks and user objectives
Functional Correctness	The degree to which a product or system provides the correct results with the needed degree of precision.
Functional Appropriateness	The degree to which the functions facilitate the accomplishment of specified tasks and objectives.
Performance Efficiency	
Time Behaviour	The degree to which the response and processing times and throughput rates of a product or system, when performing its functions, meet requirements.
Resource Utilization	The degree to which the amounts and types of resources used by a product or system, when performing its functions, meet requirements.
Capacity	The degree to which the maximum limits of a product or system parameter meet requirements.
Compatibility	
Co-existence	The degree to which a product can perform its required functions efficiently while sharing a common environment and resources with other products, without detrimental impact on any other product.
Interoperability	The degree to which two or more systems, products or components can exchange information and use the information that has been exchanged.
⋮	⋮

Table B.1: Product Quality [53]

(Sub-)Characteristic	Description
Usability	
Appropriateness Recognizability	The degree to which users can recognize whether a product or system is appropriate for their needs.
Learnability	The degree to which a product or system can be used by specified users to achieve specified goals of learning to use the product or system with effectiveness, efficiency, freedom from risk and satisfaction in a specified context of use.
Operability	The degree to which a product or system has attributes that make it easy to operate and control.
User Error Protection	The degree to which a system protects users against making errors.
User Interface Aesthetics	The degree to which a user interface enables pleasing and satisfying interaction for the user.
Accessibility	The degree to which a product or system can be used by people with the widest range of characteristics and capabilities to achieve a specified goal in a specified context of use.
Reliability	
Maturity	The degree to which a system, product or component meets needs for reliability under normal operation.
Availability	The degree to which a system, product or component is operational and accessible when required for use.
Fault Tolerance	The degree to which a system, product or component operates as intended despite the presence of hardware or software faults.
Recoverability	The degree to which, in the event of an interruption or a failure, a product or system can recover the data directly affected and re-establish the desired state of the system.
Security	
Confidentiality	The degree to which a product or system ensures that data are accessible only to those authorized to have access.
Integrity	The degree to which a system, product or component prevents unauthorized access to, or modification of, computer programs or data.
⋮	⋮

Table B.1: Product Quality [53]

(Sub-)Characteristic	Description
Non-reudiation	The degree to which actions or events can be proven to have taken place so that the events or actions cannot be repudiated later.
Accountability	The degree to which the actions of an entity can be traced uniquely to the entity.
Authenticity	The degree to which the identity of a subject or resource can be proved to be the one claimed.
Maintainability	
Modularity	The degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components.
Reusability	The degree to which an asset can be used in more than one system, or in building other assets.
Analysability	The degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified.
Modifiability	The degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality.
Testability	The degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met.
Portability	
Adaptability	The degree to which a product or system can effectively and efficiently be adapted for different or evolving hardware, software or other operational or usage environments.
Installability	The degree of effectiveness and efficiency with which a product or system can be successfully installed and/or uninstalled in a specified environment.
Replaceability	The degree to which a product can replace another specified software product for the same purpose in the same environment.

Table B.2: Quality in Use [53]

(Sub-)Characteristic	Description
Effectiveness	The accuracy and completeness with which users achieve specified goals.
Efficiency	The resources expended in relation to the accuracy and completeness with which users achieve goals.
Satisfaction	
Usefulness	The degree to which a user is satisfied with their perceived achievement of pragmatic goals, including the results of use and the consequences of use.
Trust	The degree to which a user or other stakeholder has confidence that a product or system will behave as intended.
Pleasure	The degree to which a user obtains pleasure from fulfilling their personal needs.
Comfort	The degree to which the user is satisfied with physical comfort.
Freedom from Risk	
Economic Risk Mitigation	The degree to which a product or system mitigates the potential risk to financial status, efficient operation, commercial property, reputation or other resources in the intended contexts of use.
Health and Safety Risk Mitigation	The degree to which a product or system mitigates the potential risk to people in the intended contexts of use.
Environmental Risk Mitigation	The degree to which a product or system mitigates the potential risk to property or the environment in the intended contexts of use.
Context Coverage	
Context Completeness	The degree to which a product or system can be used with effectiveness, efficiency, freedom from risk and satisfaction in all the specified contexts of use.
Flexibility	The degree to which a product or system can be used with effectiveness, efficiency, freedom from risk and satisfaction in contexts beyond those initially specified in the requirements.

Bibliography

[1] *2021 State of the API Report — API Technologies*. 2021. URL: https://www.postman.com/state-of-api/2021/api-technologies/ (visited on 09/13/2023).

[2] *2023 State of the API Report — API Technologies*. 2023. URL: https://www.postman.com/state-of-api/api-technologies/ (visited on 09/13/2023).

[3] Sultan Almuhammadi and Majeed Alsaleh. "Information Security Maturity Model for Nist Cyber Security Framework". In: *Computer Science & Information Technology (CS & IT)*. Academy & Industry Research Collaboration Center (AIRCC), Feb. 2017, pp. 51–62. ISBN: 978-1-921987-62-5. DOI: 10.5121/csit.2017.70305.

[4] Amazon Web Services, Inc. *RPC vs REST - Difference Between API Architectures - AWS*. URL: https://aws.amazon.com/compare/the-difference-between-rpc-and-rest/ (visited on 09/15/2023).

[5] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. 2. ed., 1. print. SEI Series in Software Engineering. Addison-Wesley, 2003. ISBN: 978-0-321-15495-8.

[6] Kang Bing et al. "Design of an Internet of Things-based smart home system". In: *2011 2nd International Conference on Intelligent Control and Information Processing*. Vol. 2. 2011, pp. 921–924. DOI: 10.1109/ICICIP.2011.6008384.

[7] Giuseppe Bisicchia, Stefano Forti, and Antonio Brogi. *A Declarative Goal-oriented Framework for Smart Environments with LPaaS*. June 2021. DOI: 10.48550/arXiv.2106.13083. eprint: 2106.13083 (cs, eess). (Visited on 11/08/2023).

[8] B W Boehm, J R Brown, and M Lipow. "Quantitative Evaluation of Software Quality". In: Proceedings of the 2nd International Conference of Software Engineering, 1976.

[9] Barry Boehm. "Improving and Balancing Software Qualities". In: *Proceedings of the 38th International Conference on Software Engineering Companion*. ICSE '16. New York, NY, USA: Association for Computing Machinery, May 2016, pp. 890–891. ISBN: 978-1-4503-4205-6. DOI: 10.1145/2889160.2891049. (Visited on 09/04/2023).

[10] Roberta Calegari et al. "Logic Programming as a Service (LPaaS): Intelligence for the IoT". In: *2017 IEEE 14th International Conference on Networking, Sensing and Control (ICNSC)*. 2017, pp. 72–77. DOI: 10.1109/ICNSC.2017.8000070.

[11] Joseph P. Cavano and James A. McCall. "A Framework for the Measurement of Software Quality". In: *Proceedings of the Software Quality Assurance Workshop on Functional and Performance Issues* -. Not Known: ACM Press, 1978, pp. 133–139. DOI: 10.1145/800283.811113.

[12] Chien-Tsun Chen et al. "Exception Handling Refactorings: Directed by Goals and Driven by Bug Fixing". In: *Journal of Systems and Software* 82.2 (Feb. 2009), pp. 333–345. ISSN: 0164-1212. DOI: 10.1016/j.jss.2008.06.035. (Visited on 10/27/2023).

[13] Steven J. Drew and K. John Gough. "Exception Handling: Expecting the Unexpected". In: *Computer Languages* 20.2 (May 1994), pp. 69–87. ISSN: 0096-0551. DOI: 10.1016/0096-0551(94)90015-9. (Visited on 10/27/2023).

[14] Roy Thomas Fielding. "Architectural Styles and the Design of Network-based Software Architectures". PhD thesis. Irvine: University of California, 2000.

[15] Martin Fowler. *Richardson Maturity Model*. Mar. 2010. URL: https://martinfowler.com/articles/richardsonMaturityModel.html (visited on 08/31/2023).

[16] Tamas Galli, Francisco Chiclana, and Francois Siewe. "Software Product Quality Models, Developments, Trends, and Evaluation". In: *SN Computer Science* 1.3 (May 2020), p. 24. ISSN: 2661-8907. DOI: 10.1007/s42979-020-00140-z.

[17] Taqiyah Khadijah Ghazali and Nur Haryani Zakaria. "Security, comfort, healthcare, and energy saving: A review on biometric factors for smart home environment". In: *Journal of Computers* 29.1 (2018), pp. 189–208.

[18] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. 2. ed., internat. ed. Upper Saddle River, NJ: Prentice Hall, Pearson Education, 2003. ISBN: 978-0-13-099183-6.

[19] *HATEOAS*. URL: https://en.wikipedia.org/wiki/HATEOAS (visited on 11/06/2023).

[20] James Herbsleb et al. "Software Quality and the Capability Maturity Model". In: *Communications of the ACM* 40.6 (June 1997), pp. 30–40. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/255656.255692.

[21] J.R. Horgan, S. London, and M.R. Lyu. "Achieving software quality with testing coverage measures". In: *Computer* 27.9 (1994), pp. 60–69. ISSN: 1558-0814. DOI: 10.1109/2.312032.

[22] Chin Yun Hsieh et al. "Identification and Refactoring of Exception Handling Code Smells in Javascript". In: *Journal of Internet Technology* 18.6 (2017), pp. 1461–1471. ISSN: 1607-9264. DOI: 10.6138/JIT.2017.18.6.20160118. (Visited on 10/27/2023).

[23] *Hypertext Transfer Protocol – HTTP/1.1*. Standard RFC 2616. IETF, 1999. URL: https://www.rfc-editor.org/rfc/rfc2616 (visited on 09/13/2023).

[24] IBM. *WS-Security*. July 2023. URL: https://www.ibm.com/docs/en/app-connect/11.0.0?topic=security-ws (visited on 09/15/2023).

[25] *JavaScript Object Notation (JSON) Patch*. Standard RFC 6902. IETF, 2013. URL: https://www.rfc-editor.org/rfc/rfc6902 (visited on 10/28/2023).

[26] Bob Jones. *REST Architecture*. June 2020. URL: https://www.redhat.com/en/blog/rest-architecture (visited on 09/04/2023).

[27] Evan King et al. "Sasha: Creative Goal-Oriented Reasoning in Smart Homes with Large Language Models". In: arXiv:2305.09802 (May 2023). DOI: 10.48550/arXiv.2305.09802. eprint: 2305.09802. (Visited on 11/07/2023).

[28] P.B. Kruchten. "The 4+1 View Model of Architecture". In: *IEEE Software* 12.6 (Nov. 1995), pp. 42–50. ISSN: 1937-4194. DOI: 10.1109/52.469759.

[29] Tim Lindholm et al. *Java Virtual Machine Specification*. Feb. 28, 2013. URL: https://docs.oracle.com/javase/specs/jvms/se7/html/index.html (visited on 10/26/2023).

[30] Gabriele Lobaccaro, Salvatore Carlucci, and Erica Löfström. "A Review of Systems and Technologies for Smart Homes and Smart Grids". In: *Energies* 9.5 (May 2016), p. 348. ISSN: 1996-1073. DOI: 10.3390/en9050348. (Visited on 09/28/2023).

[31] Microsoft. *RESTful web API design*. 2023. URL: https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design (visited on 06/30/2023).

[32] Michael C Mozer. "The Neural Network House: An Environment that Adapts to Its Inhabitants". In: ().

[33] Martin Nally. *REST vs. RPC: What Problems Are You Trying to Solve with Your APIs?* Oct. 2018. URL: https://cloud.google.com/blog/products/application-development/rest-vs-rpc-what-problems-are-you-trying-to-solve-with-your-apis (visited on 06/29/2023).

[34] Martin Nally. *gRPC vs REST: Understanding gRPC, OpenAPI and REST and When to Use Them in API Design*. Apr. 2020. URL: https://cloud.google.com/blog/products/api-management/understanding-grpc-openapi-and-rest-and-when-to-use-them (visited on 06/29/2023).

[35] Javier Palanca, Vicente Julian, and Ana García-Fornes. "A Goal-Oriented Execution Module Based on Agents". In: *2011 44th Hawaii International Conference on System Sciences*. 2011, pp. 1–10. DOI: 10.1109/HICSS.2011.14.

[36] Javier Palanca et al. "Designing a Goal-Oriented Smart-Home Environment". In: *Information Systems Frontiers* 20.1 (Feb. 2018), pp. 125–142. ISSN: 1572-9419. DOI: 10.1007/s10796-016-9670-x. (Visited on 08/31/2023).

[37] JoonSeok Park et al. "CASS: A Context-Aware Simulation System for Smart Home". In: *5th ACIS International Conference on Software Engineering Research, Management & Applications (SERA 2007)*. Aug. 2007, pp. 461–467. DOI: 10.1109/SERA.2007.60.

[38] *Problem Details for HTTP APIs*. Standard RFC 7807. IETF, 2016. URL: https://www.rfc-editor.org/rfc/rfc7807 (visited on 09/22/2023).

[39] *Prolog*. Standard ISO/IEC 13211-1:1995. ISO (International Organization for Standardization), 1995. URL: https://www.iso.org/standard/21413.html (visited on 11/20/2023).

[40] Red Hat. *REST vs. SOAP*. Apr. 2023. URL: https://www.redhat.com/en/topics/integration/whats-the-difference-between-soap-rest (visited on 09/15/2023).

[41] Leonnard Richardson. *Act Three: The Maturity Heuristic*. Conference Presentation. Stanford, Jan. 2009. (Visited on 09/13/2023).

[42] Ivan Salvadori and Frank Siqueira. "A Maturity Model for Semantic RESTful Web APIs". In: *2015 IEEE International Conference on Web Services*. June 2015, pp. 703–710. DOI: 10.1109/ICWS.2015.98.

[43] Lalatendu Satpathy. "Smart housing: technology to aid aging in place-new opportunities and challenges". In: (2006).

[44] *Scaling and Parallel Processing*. URL: https://docs.spring.io/spring-batch/docs/current/reference/html/scalability.html (visited on 11/14/2023).

[45] *Service (Spring Framework 6.0.12 API)*. URL: https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/Service.html (visited on 09/20/2023).

[46] Simon Brown. *The C4 model for visualising software architecture*. URL: https://c4model.com/ (visited on 09/19/2023).

[47] Brijendra Singh and Suresh Prasad Kannojia. "A Review on Software Quality Models". In: *2013 International Conference on Communication Systems and Network Technologies*. Apr. 2013, pp. 801–806. DOI: 10.1109/CSNT.2013.171.

[48] *Software engineering - Product quality - Part 1: Quality model*. Standard ISO/IEC 9126-1:2001. ISO (International Organization for Standardization), 2001. URL: https://www.iso.org/standard/22749.html (visited on 09/08/2023).

[49] Spring. *Spring Web MVC Framework*. URL: https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/mvc.html (visited on 09/18/2023).

[50] *Spring Framework Documentation*. URL: https://docs.spring.io/spring-framework/reference/index.html (visited on 11/06/2023).

[51] Rossen Stoyanchev and Juergen Hoeller. *ProblemDetail (Spring Framework 6.0.13 API)*. URL: https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/http/ProblemDetail.html (visited on 10/28/2023).

[52] *Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - Guide to SQuaRE*. Standard ISO/IEC/IEEE 25000:2014(en). ISO (International Organization for Standardization), 2014. URL: https://www.iso.org/obp/ui/en/#iso:std:iso-iec:25000:ed-2:en (visited on 09/08/2023).

[53] *Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models*. Standard ISO/IEC 25010:2011(en). ISO (International Organization for Standardization), 2011. URL: https://www.iso.org/obp/ui/en/#iso:std:iso-iec:25010:ed-1:v1:en (visited on 09/08/2023).

[54] *Systems and software engineering - Vocabulary*. Standard ISO/IEC/IEEE 24765:2017(en). ISO (International Organization for Standardization), 2017. URL: https://www.iso.org/obp/ui/en/#iso:std:iso-iec-ieee:24765:ed-2:v1:en (visited on 09/08/2023).

[55] *Uniform Resource Identifier (URI): Generic Syntax*. Standard RFC 3986. IETF, 2005. URL: https://www.rfc-editor.org/rfc/rfc3986 (visited on 09/13/2023).

[56] Colin C. Venters et al. "Software sustainability: Research and practice from a software architecture viewpoint". In: *Journal of Systems and Software* 138 (2018), pp. 174–188. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2017.12.026. URL: https://www.sciencedirect.com/science/article/pii/S0164121217303072.

[57] verma_anushka. *Benefit of using MVC*. Apr. 15, 2023. URL: https://www.geeksforgeeks.org/benefit-of-using-mvc/ (visited on 09/19/2023).

[58] VMWare Inc. *Spring*. URL: https://spring.io (visited on 09/19/2023).

[59] VMWare Inc. *Why Spring?* URL: https://spring.io/why-spring (visited on 09/20/2023).

[60] *W3C Semantic Web Frequently Asked Questions*. 2009. URL: https://www.w3.org/2001/sw/SW-FAQ (visited on 09/13/2023).

[61] World Wide Web Consortium. *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)*. Apr. 2007. URL: https://www.w3.org/TR/soap12/ (visited on 09/15/2023).

Further Sources

Some figures use images and symbols from third parties which require attribution. They are listed in the following:

- flatflaticon.com, by *SatawatDesign*: Desk Lamp
  Usage(s): Figure 7.1
- flatflaticon.com, by *Handicon*: Monitor
  Usage(s): Figure 7.1
- flatflaticon.com, by *Freepik*: Sun
  Usage(s): Figure 7.1

Versicherung an Eides Statt

Ich versichere an Eides statt durch meine untenstehende Unterschrift,

- dass ich die vorliegende Arbeit - mit Ausnahme der Anleitung durch die Betreuer - selbstständig ohne fremde Hilfe angefertigt habe und
- dass ich alle Stellen, die wörtlich oder annähernd wörtlich aus fremden Quellen entnommen sind, entsprechend als Zitate gekennzeichnet habe und
- dass ich ausschließlich die angegebenen Quellen (Literatur, Internetseiten, sonstige Hilfsmittel) verwendet habe und
- dass ich alle entsprechenden Angaben nach bestem Wissen und Gewissen vorgenommen habe, dass sie der Wahrheit entsprechen und dass ich nichts verschwiegen habe.

Mir ist bekannt, dass eine falsche Versicherung an Eides Statt nach §156 und §163 Abs. 1 des Strafgesetzbuches mit Freiheitsstrafe oder Geldstrafe bestraft wird.

Duisburg, 1. Dezember 2023 _____

(Ort, Datum) (Vorname Nachname)