Master's Thesis

Design and Implementation of a Template-Based System for Flexible Smart Home Automation

by

Luis Thiele

submitted in partial fulfillment of the requirements
for the degree of Master of Science
in Computer Science

Department of Computer Science
Paris Lodron Universität Salzburg
Salzburg, Austria

supervised by

Univ.-Prof. Dr. Wolfgang Pree

June 8th, 2025

Abstract

The proliferation of smart home devices has created a paradox: while the potential for sophisticated automation is immense, the tools available to users often fail to bridge the gap between power and accessibility. This thesis confronts this challenge through the design and implementation of a flexible, template-based automation system for the so-called Honua smart home platform. It begins by analyzing the critical limitations of a first-generation, rigid ”Rules” engine, which bound automation logic in a restrictive one-to-one relationship with devices, failing to accommodate complex scenarios or diverse user capabilities.

The core contribution of this work is the development of a novel automation paradigm centered on reusable templates. This system effectively decouples the complexity of creating automation logic from the simplicity of its application. It empowers advanced users to construct powerful, multi-step automation templates with nested conditions and multiple actions, while enabling novice users to instantiate these templates through a simple, parameter-based interface. This approach is demonstrated through the implementation of several real-world automation scenarios, from simple conditional lighting to complex energy surplus management.

Furthermore, this thesis details a significant architectural refactoring of the underlying platform, replacing a brittle dual-backend infrastructure with a robust, single-server model running on a Raspberry Pi. Secure remote access, a critical challenge for self-hosted systems, is achieved through a custom-built SSH tunneling solution, eliminating the need for a public cloud gateway. The entire system, implemented in Go and deployed as a containerized Docker stack, presents a complete and practical solution that markedly enhances the flexibility, reusability, and user-friendliness of smart home automation.

Zusammenfassung

Die zunehmende Verbreitung von Smart-Home-Geräten hat ein Paradoxon geschaffen: Während das Potenzial für hochentwickelte Automatisierung immens ist, gelingt es den verfügbaren Werkzeugen oft nicht, die Lücke zwischen Leistungsfähigkeit und Zugänglichkeit zu schließen. Diese Masterarbeit stellt sich dieser Herausforderung durch den En-twurf und die Implementierung eines flexiblen, templatebasierten Automatisierungssystems für die sogenannte Honua Smart-Home-Plattform. Sie beginnt mit der Analyse der entscheidenden Einschränkungen einer starren "RegelEngine der ersten Generation, die Automatisierungslogik in einer restriktiven Eins-zu-eins-Beziehung an Geräte band und somit weder komplexe Szenarien noch unterschiedliche Benutzerfähigkeiten berücksichtigen konnte.

Der Kernbeitrag dieser Arbeit ist die Entwicklung eines neuartigen Automatisierungsparadigmas, das auf wiederverwendbaren Templates basiert. Dieses System entkoppelt effektiv die Komplexität der Erstellung von Automatisierungslogik von der Einfachheit ihrer Anwendung. Es ermöglicht fortgeschrittenen Benutzern, leistungsstarke, mehrstufige Automatisierungsvorlagen mit verschachtelten Bedingungen und mehreren Aktionen zu erstellen, während es Anfängern gestattet, diese Vorlagen über eine einfache, parameterbasierte Schnittstelle zu instanziieren. Dieser Ansatz wird durch die Implementierung mehrerer realer Automatisierungsszenarien demonstriert, von einfacher konditionaler Beleuchtung bis hin zum komplexen Management von Energieüberschüssen.

Darüber hinaus beschreibt diese Arbeit ein signifikantes architektonisches Refactoring der zugrundeliegenden Plattform, bei dem eine fehleranfällige duale Backend-Infrastruktur durch ein robustes Einzelserver-Modell ersetzt wird, das auf einem Raspberry Pi läuft. Der sichere Fernzugriff, eine kritische Herausforderung für selbstgehostete Systeme, wird durch eine maßgeschneiderte SSH-Tunneling-Lösung realisiert, wodurch die Notwendigkeit eines öffentlichen Cloud-Gateways entfällt. Das gesamte System, implementiert in Go und als containerisierter Docker-Stack bereitgestellt, stellt eine vollständige und praxisnahe Lösung dar, welche die Flexibilität, Wiederverwendbarkeit und Benutzerfreundlichkeit der Smart-Home-Automatisierung entscheidend verbessert.

AI Disclosure Statement

In the course of writing this Master's thesis, I employed the artificial intelligence (AI)-based tools ChatGPT-4.5 from OpenAI and Gemini Pro 2.5 from Google for support.

I utilized the AI to generate text for Chapter 1, in which commonly known technologies are described, and for the abstract.

All content generated by the AI was critically reviewed, placed into its technical context, and revised where necessary by me. The entirety of all statements, arguments, as well as the selection and evaluation of the sources, remains my own responsibility.

The use of these tools was reflective and undertaken with an awareness of the technical, ethical, and scientific limitations of generative AI systems.

Contents

1 Introduction

In the rapidly evolving landscape of the Internet of Things (IoT), the smart home has emerged as a focal point of innovation, promising a future of enhanced convenience, efficiency, and security. However, the current ecosystem is often fragmented, with a multitude of devices from various manufacturers, each requiring its own application and configuration process. This complexity can present a significant barrier to entry for non-technical users, undermining the very convenience that smart technology aims to provide.

Honua is a sophisticated yet user-centric application designed to address these challenges directly. It offers a unified and streamlined platform for smart home automation, prioritizing intuitive operation and effortless configuration. By abstracting the underlying technical complexities, Honua provides a seamless user experience, empowering individuals to easily manage their smart devices and create powerful, personalized automation routines. This introduction details the robust technological foundation upon which Honua is built, explaining the role and synergy of each component in delivering a powerful and accessible smart home solution.

1.1 Tech-Stack

The architecture of Honua is a carefully curated ecosystem of modern, high-performance tools and technologies. The backend, which forms the core of the system, is deployed as a cohesive Docker stack on a low-power Raspberry Pi. This backend leverages the comprehensive Home Assistant API to interface with a wide array of smart sensors and control devices. The primary logic is implemented in Go, a language renowned for its efficiency and concurrency, with data persistence handled by a dual-database strategy employing both PostgreSQL and MongoDB. The frontend is a native Android application built with Flutter (Dart), ensuring a responsive and modern user interface. Communication between the frontend and backend is achieved through gRPC, a high-performance framework that utilizes Protocol Buffers for defining a clear and strongly-typed API contract.

1.1.1 Home Assistant

Home Assistant is a home automation platform that serves as the foundational layer for Honua. Launched in 2013, it has grown into one of the most powerful and flexible smart home solutions available, supported by a massive and vibrant global community of developers and enthusiasts. Its primary mission is to provide a unified system for controlling all aspects of a smart home, with a steadfast commitment to local control and user privacy [1].

Core Philosophy: Local Control and Privacy Unlike many commercial smart home hubs that rely on cloud servers, Home Assistant is designed to run locally on hardware within the user's own network, such as a Raspberry Pi or a dedicated home server. This “local-first” approach is a cornerstone of its philosophy and offers several critical advantages:

- Reliability: Since the core logic runs locally, the smart home continues to function perfectly even if the internet connection goes down. Automations will still trigger, and devices can be controlled without issue.
- Speed: Commands are sent directly from the local server to the devices on the same network, resulting in near-instantaneous response times without the latency of a round-trip to a corporate cloud server.
- Privacy: Perhaps most importantly, all sensitive data about the user’s home—when they are present, which lights are on, camera feeds—remains securely within their own network. This gives the user complete ownership and control over their personal information.

Integrations The single greatest strength of Home Assistant is its immense ecosystem of integrations. The platform acts as a universal translator for smart devices, capable of communicating with a staggering number of products from hundreds of different brands. Thanks to its open-source nature, the community has contributed integrations for over 2,500 different devices and services, covering every conceivable category:

- Lighting: Philips Hue, Lifx, Nanoleaf, WLED
- Climate: Nest, Ecobee, Tado, generic climate controllers
- Media & Entertainment: Sonos, Spotify, Plex, Kodi, Samsung TV
- Security: Alarm systems, security cameras, motion sensors, door locks
- Energy: Solar panel inverters, smart plugs with energy monitoring

Home Assistant unifies these disparate ecosystems, allowing devices that were never designed to work together to be seamlessly incorporated into powerful automations.

The Role of Home Assistant in Honua While Home Assistant provides its own user interface (known as Lovelace) and a robust automation engine, Honua is designed to offer a uniquely curated and streamlined user experience. Instead of attempting to replicate the monumental effort of integrating with thousands of devices, Honua strategically leverages Home Assistant’s power.

In the Honua architecture, Home Assistant acts as a hardware abstraction layer. It does the heavy lifting of device discovery, communication, and standardization. Honua then interacts with Home Assistant’s stable and well-documented Application Programming Interface (API). This approach provides several key benefits:

- Universal Compatibility: By using the Home Assistant API, Honua instantly gains access to every device that Home Assistant supports, both now and in the future.
- Focus on User Experience: By delegating the low-level device control to Home Assistant, the development of Honua can focus entirely on what it does best: creating an intuitive interface, a superior automation-building experience, and a polished mobile application.
- Stability and Reliability: Honua builds upon a mature, time-tested platform known for its stability, offloading the core operational risks to a system trusted by millions.

Essentially, Honua serves as a sophisticated and user-centric "head" for the powerful Home Assistant "body". It replaces the default user interface and automation implementation with its own specialized logic, while relying on the underlying Home Assistant platform to serve as a reliable and all-encompassing bridge to the physical hardware in the user's home.

1.1.2 Go

The entire backend logic of Honua is written in Go (Golang). Developed at Google and released in 2009, Go was conceived as a pragmatic response to the growing complexity of modern software development. It is a statically typed, compiled language designed to build software that is simple, highly efficient, and exceptionally reliable, making it a premier choice for demanding backend systems [2].

Strengths Go's design philosophy can be summarized as "less is more". It intentionally omits many of the complex features found in other object-oriented languages like C++ or Java, resulting in a smaller language specification and a cleaner, more readable syntax. This simplicity is not a limitation but a powerful feature, as it leads to code that is easier to write, review, and maintain over the long term.

Its key strengths are particularly suited to the needs of a smart home application:

- Elite Concurrency: Go's most celebrated feature is its built-in model for concurrency. Instead of traditional, heavy operating system threads, Go uses goroutines, which are extremely lightweight threads managed by the Go runtime. It's feasible to have hundreds of thousands of goroutines running simultaneously. Communication between these goroutines is safely handled through channels, which prevent the race conditions that plague other concurrent programming models.
- Compiled Performance: Go is a compiled language, translating source code directly into machine code. This results in performance that approaches the speed of C or C++, while offering memory safety and garbage collection. For Honua, this means the system can process sensor data, evaluate

complex automation rules, and respond to user commands with minimal delay.

- Effortless Deployment: A Go program compiles into a single, statically linked binary file. This file contains the application and all its dependencies, with no need for interpreters or virtual machines on the target machine. This makes deployment incredibly straightforward—a single file can be copied to the server and executed.

Go's Role in the Honua Architecture Within Honua, Go's capabilities are leveraged to create a backend that is both powerful and robust. The concurrency model is a perfect match for the chaotic nature of a smart home environment, where dozens of events might occur at once. A motion sensor trigger, a user tapping a button in the app, a scheduled automation firing, and a thermostat reporting a new temperature can all be handled concurrently as independent goroutines, ensuring the system remains responsive and stable under load. The compiled binary is placed within a minimal Docker container, creating a highly efficient and portable service that is perfectly suited for running on the resource-conscious Raspberry Pi.

1.1.3 gRPC & protoc

For all communication between the Flutter mobile application (frontend) and the Go backend, Honua utilizes gRPC (Google Remote Procedure Calls). Developed and open-sourced by Google, gRPC is a modern, high-performance framework designed to create efficient and reliable connections between services. It represents a significant evolution from traditional REST APIs, offering superior speed, stronger guarantees, and more flexible communication patterns [3].

Protocol Buffers At the heart of gRPC is Protocol Buffers (protoc), a powerful and language-agnostic mechanism for serializing structured data. Unlike REST APIs that typically use text-based JSON, gRPC uses Protobuf to encode data into a compact binary format. This has profound implications for performance:

- Size and Speed: Protobuf messages are significantly smaller and faster to serialize and deserialize than their JSON counterparts. This reduces network bandwidth consumption and saves CPU cycles, which is especially critical for battery-powered mobile devices.

- Strong Typing and a Clear Contract: The structure of all API calls and data messages is defined in a central .proto file. This file acts as an unambiguous, language-agnostic contract. The protoc compiler then uses this file to automatically generate the necessary client-side code in Dart (for Flutter) and server-side code in Go. This strong typing eliminates an entire class of data-mismatch errors and makes the API robust against changes.

Communication Beyond simple request-response, gRPC excels by supporting advanced, real-time communication patterns. Its most powerful feature for an application like Honua is bidirectional streaming. This allows for a persistent, two-way communication channel to be established between the server and the app.

This capability is the key to Honua's live, responsive feel. When a device state changes—for example, a physical light switch is flipped—the backend can instantly push this update through the open stream to the app. The app doesn't need to constantly poll the server asking updates. This proactive push model is far more efficient and provides the instantaneous UI feedback that users expect from a modern smart home application.

1.1.4 PostgreSQL

To cache critical states, Honua relies on PostgreSQL, an exceptionally powerful and reliable open-source object-relational database system. With over three decades of active development, PostgreSQL has earned an unparalleled reputation for its adherence to standards, feature robustness, and data integrity, making it a trusted choice for mission-critical applications [4].

Relational Model PostgreSQL is built on the relational model, which organizes data into tables with predefined columns and data types. This structured approach is ideal for data where consistency and integrity are non-negotiable.

- ACID Compliance: PostgreSQL is fully ACID compliant (Atomicity, Consistency, Isolation, Durability). This is a set of guarantees ensuring that database transactions are processed reliably. For Honua, it means that when a device state change is recorded, it is guaranteed to be saved correctly and permanently, preventing data corruption or loss.
- Data Integrity: The rigid schema ensures that data is always clean and predictable. A temperature reading is always stored as a number, and a timestamp is always a valid time. This makes querying and aggregating data for historical analysis—such as creating a chart of a room's temperature over the last 24 hours—both simple and trustworthy.

PostgreSQL's Role in Honua Within the Honua architecture, PostgreSQL serves as the official system of record for time-series and state data. Every event, from a door sensor being triggered to a smart plug reporting its energy consumption, can be logged with a precise timestamp. This provides a rich, queryable history of everything that has happened in the home. By caching the most recent state of all devices, the system can quickly serve this information to the app without needing to query every device individually, significantly improving the responsiveness of the user interface.

1.1.5 MongoDB

Complementing the structured storage of PostgreSQL, Honua employs MongoDB to handle data that is dynamic, complex, and doesn't fit neatly into the rigid rows and columns of a relational database. MongoDB is a leading document-oriented NoSQL database that offers immense flexibility in how it stores information, making it a perfect fit for user-generated and configuration-heavy content [5].

Document Model Instead of tables, MongoDB stores data in BSON (Binary JSON) documents. These documents are self-describing, hierarchical data structures that can include nested documents and arrays. This model provides what is often called a "flexible schema".

- Adaptability: The structure of a document can be changed at any time. New fields can be added, or old ones removed, without requiring a complex and disruptive database migration. This is ideal for a system in active development, where features and their underlying data needs may evolve.
- Natural Data Mapping: Many objects in programming languages, like a complex configuration object in Go, map naturally to a single JSON-like document. Storing this object in MongoDB is as simple as serializing it, which greatly simplifies the application code compared to shredding the object across multiple relational tables.

MongoDB's Role in Honua MongoDB is the designated repository for all user-specific configurations that demand flexibility. Its primary use cases in Honua are:

- Dashboard Layouts: Every user's dashboard is unique. One user might have a simple list of buttons, while another might have a complex, multi-tabbed grid with graphs, sliders, and custom widgets. Storing this entire layout as a single, nested document in MongoDB is incredibly natural and efficient.
- Automation Templates: Honua's automation templates can have varying numbers of parameters, optional fields, and complex logical structures. The document model effortlessly accommodates this variability, allowing for rich and powerful templates to be serialized, stored, and accessed without constraint.

1.1.6 Docker

The entire Honua backend—the Go application, the PostgreSQL and MongoDB databases, and all their dependencies—is packaged and deployed using Docker. Docker is a revolutionary platform that enables applications to be run in isolated environments called containers. This approach fundamentally changes how software is built, shipped, and run, bringing industrial-scale consistency and efficiency to projects of any size [6].

Containers: Lightweight, Portable, and Efficient A container bundles an application's code with all the libraries and dependencies it needs to run. This self-contained unit can then be run on any machine that has Docker installed, regardless of its underlying operating system or configuration.

- Consistency and Portability: Docker solves the classic "it works on my machine" problem. A containerized application runs in an identical environment whether it's on a developer's Windows laptop, a testing server, or the production Raspberry Pi. This eliminates a vast array of potential bugs caused by environment differences.
- Lightweight Virtualization: Unlike traditional Virtual Machines (VMs) that virtualize an entire hardware stack and run a full guest operating system, containers virtualize at the OS level, sharing the host machine's kernel. This makes them incredibly lightweight, allowing them to start in seconds and consume far fewer resources in terms of CPU and RAM.

Docker's Role in Honua For the Honua project, Docker and its companion tool, Docker Compose, provide the backbone for deployment and management. The entire backend is defined in a single `.yaml` file, which describes the set of services to run: one container for the Go application, one for the PostgreSQL database, and one for the MongoDB database.

This architecture provides immense operational benefits. Deploying the entire system from scratch becomes as simple as running a single command. Updating the application is just as easy: pull the new image for the Go service and restart the stack. This containerized approach ensures that the Honua backend is robust, easy to manage, and simple to replicate, forming a professional and resilient deployment strategy.

1.1.7 Raspberry Pi

The designated hardware platform for the Honua backend is a Raspberry Pi, a series of low-cost, credit-card-sized single-board computers (SBCs). Originally created by the Raspberry Pi Foundation to promote computer science education, these tiny yet capable devices have ignited a movement among makers, hobbyists, and even industrial users, proving to be the perfect platform for countless dedicated computing tasks [7].

Single-Board Computer An SBC, as the name implies, integrates all the essential components of a functional computer—CPU, memory, networking, and I/O ports like USB—onto a single circuit board. This compact and integrated design makes it perfect for embedded applications where size, cost, and power consumption are critical factors.

The Raspberry Pi has established itself as the de facto standard in this space due to:

- Low Power and Cost: A modern Raspberry Pi consumes a tiny amount of electricity (typically 3-5 watts), making it dramatically cheaper to run 24/7 than a traditional desktop or server. Its low purchase price makes it highly accessible for any home project.
- Sufficient Performance: While not designed for heavy desktop computing, modern iterations of the Raspberry Pi feature powerful multi-core ARM processors and several gigabytes of RAM. This provides more than enough horsepower to run the entire containerized Honua stack—including Home Assistant, the Go backend, and multiple databases—smoothly and efficiently.
- Vast Community and Ecosystem: The Raspberry Pi is supported by one of the largest and most active communities in computing. Any question, problem, or project idea has likely been documented in extensive tutorials, forums, and videos, making setup and troubleshooting incredibly straightforward.

The Raspberry Pi's Role in Honua The Raspberry Pi is the physical home for Honua. It is the ideal "set it and forget it" device. It can be tucked away on a shelf and run silently and continuously, serving as the reliable brain of the smart home. The backend is deployed on a Raspberry Pi running the standard Raspberry Pi OS (a derivative of Debian Linux), a stable and well-supported operating system. The combination of the Pi's efficiency and Docker's containerization creates a home server solution that is powerful, affordable, and exceptionally easy to maintain.

1.2 Honua: Initial Project Implementation

This section provides a comprehensive overview of the Honua application's architecture and feature set at the commencement of this project. It details the initial design of the backend infrastructure, the capabilities of its first-generation automation engine, and a critical analysis of the inherent limitations that necessitated the subsequent re-architecture and development efforts.

1.2.1 Initial Architecture: The Public vs. Private Backend Model

At its inception, Honua employed a complex, dual-backend architecture designed to overcome a common and significant hurdle in self-hosted applications: providing reliable remote access to a user's home network. Many home internet providers use techniques like Carrier-Grade NAT (CGNAT) or assign dynamic IP addresses, making traditional port forwarding an unreliable, technically challenging, and often insecure solution for the average user. To circumvent this,

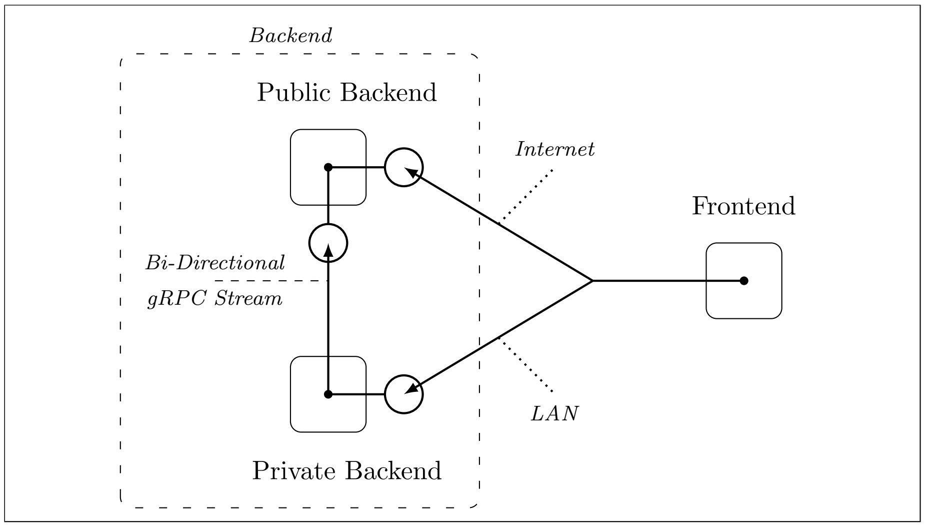
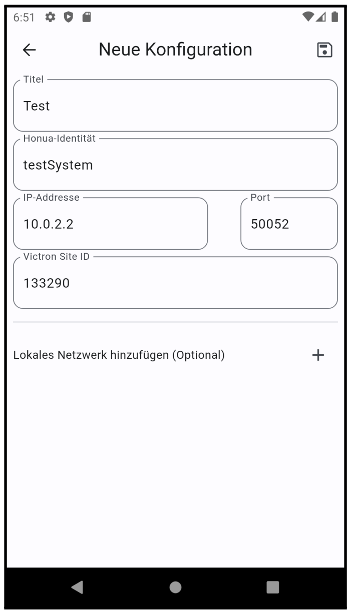


Figure 1: Private & Public Backend

the system was split into two distinct components: a “private” backend running locally within the user’s home and a “public” backend deployed in a cloud environment.

- The Private Backend (Local Controller): This component was deployed on the Raspberry Pi within the user’s home network. Its sole purpose was to be the “edge controller”, communicating directly with the Home Assistant API to query sensor states and execute commands on local smart devices. It possessed an intimate knowledge of the home’s state but was, by design, unreachable from the outside internet.
- The Public Backend (Cloud Gateway): This component was a separate application deployed on a cloud provider, giving it a stable, publicly accessible IP address. It served as the central “rendezvous point” for the system. Its primary responsibilities were to authenticate the frontend mobile application, receive commands from the user, and act as a message broker and state cache.

The communication flow between these components was orchestrated to bypass the need for an inbound connection to the user’s home. The private backend would establish a persistent, outbound connection (e.g., a long-lived WebSocket or a long-polling HTTP request) to the public backend. This connection acted as a secure tunnel. When a user issued a command from the mobile app, the request would travel to the public backend. The public backend would then send the command down the already established tunnel to the correct private backend, which would then execute it locally. State changes detected by the private backend were pushed up to the public backend through the same tunnel


6:51

Neue Konfiguration

Titel: Test

Honua-Identität: testSystem

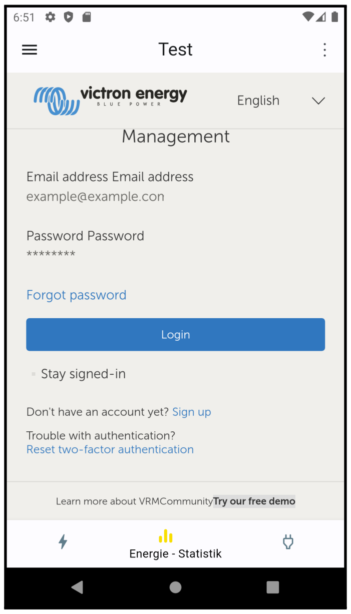
IP-Addresse: 10.0.2.2

Port: 50052

Victron Site ID: 133290

Lokales Netzwerk hinzufügen (Optional)

(a) Login Screen


6:51

Test

victron energy BLUE POWER

English

Management

Email address Email address
example@example.con

Password Password
****

Forgot password

Login

Stay signed-in

Don't have an account yet? Sign up

Trouble with authentication?
Reset two-factor authentication

Learn more about VRMCommunityTry our free demo

Energie - Statistik

(b) Victron Login Screen

Figure 2: Honua Frontend Login Screens

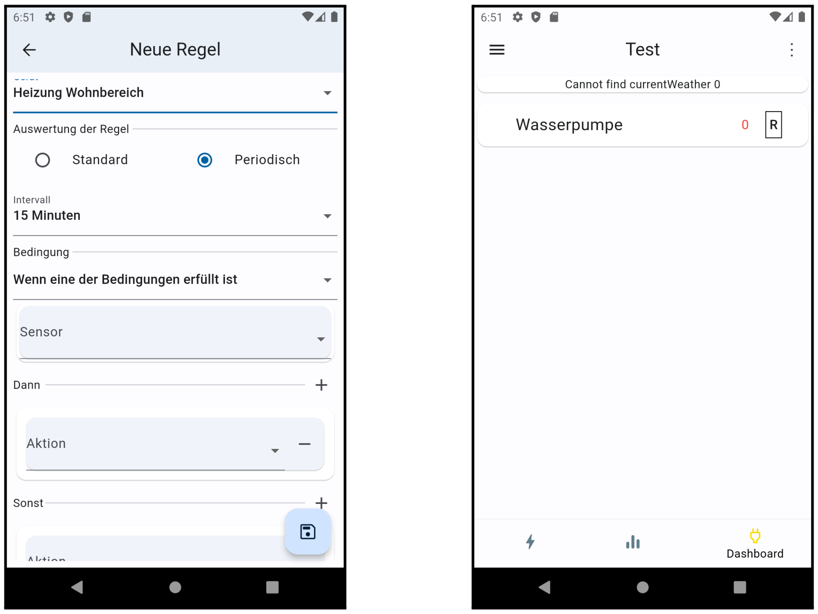
to be cached and relayed to the app.

This entire intricate workflow is illustrated in Figure 1. The diagram shows the server listeners for both backends, with the public listener being directly accessible from the internet, whereas the private listener is confined to the LAN. The arrows depict the active connection requests that form the communication chain, from the user's app to the cloud and finally down to the local Raspberry Pi.

1.2.2 The Original Frontend User Interface

The initial Honua system was a complete solution, comprising not only the backend infrastructure but also a fully functional frontend mobile application for the Android platform, developed using the Flutter framework. This application served as the primary user-facing component, providing the means to monitor, control, and automate smart home devices. The following figures illustrate the key screens of this initial application, demonstrating its capabilities and user interaction model.

The entry point to the application is the login screen, shown in Figure 2a. This screen prompted the user for credentials to connect to their Honua instance. Notably, the IP address `10.0.2.2` is a special alias within the Android emulator environment. It is used during development to redirect network traffic from the



(a) New Rule Screen (b) Rules Overview Screen

Figure 3: Honua Frontend Rules Screens

emulated phone to the loopback interface (`localhost`) of the host computer running Android Studio. This allows for seamless testing of the client-server communication without requiring a physical device on the local network.

Upon successful login, the user was presented with the rules overview screen, depicted in Figure 3b. This screen acted as the central dashboard for monitoring and managing all connected devices and their associated automation rules. For each device, such as the “Wasserpumpe” (Water Pump) shown, the interface provided critical at-a-glance information. A color-coded numerical indicator displayed the device’s current state: a red `0` signifies that the device is off, whereas a green `1` would indicate it is on.

A key feature of this interface was the intuitive, dual-mode control system. A highlighted box around the R (for Rule) signifies that the device is currently in “Automatic” mode, where its state is dictated by the logic of its assigned rule. The user could, at any time, tap on the state indicator (0 or 1) to manually override the automation. Doing so would move the highlighted box to the state indicator, placing the device in ”Manual” mode. This toggle provided users with an immediate and powerful method to take direct control or re-engage rule-based evaluation as needed.

Figure 3a shows the rule configuration screen, where users could define the logic for the automation. The interface separated triggers into two distinct categories: “Standard” for event-based triggers (e.g., from a sensor change) and “Periodic” for time-based triggers, the latter of which is selected in the figure. For combining multiple conditions into a single outcome, the user could select a logical connective from a dropdown menu, with “OR” being chosen in this example. The screen then prompts the user to populate the essential components of the rule: defining the input sensor(s) that drive the logic, and specifying the corresponding “Then” and “Else” action blocks to be executed.

Beyond direct device automation, the initial application also integrated with third-party systems to provide a more holistic view of the home environment. Figure 2b displays the login screen for Victron Energy, a manufacturer of power management systems. This screen served as a gateway to the Victron Remote Management (VRM) portal. After authenticating, a user could access a wealth of detailed telemetry regarding their home’s energy ecosystem, including data on solar power production, battery state of charge, and overall energy consumption, directly within the Honua application.

1.2.3 First-Generation Automation: The “Rules” System

The initial version of Honua already supported automation through a system referred to as “Rules”. This system was designed to be simple and approachable, providing a basic framework for creating cause-and-effect logic. Its structure, however, was highly rigid and defined by a strict set of constraints. Rule Structure and Binding

The core concept of the Rules system was a tight, one-to-one relationship between a rule and a device. For every controllable device (e.g., a lamp), a user could define exactly one rule. Conversely, that rule could only control that single device. This made the rule less of a standalone automation and more of an extended attribute of the device itself. At any time, the user could toggle the device between two modes: “Manual”, where the user had direct on/off control, and “Automatic”, where the device’s state was determined exclusively by the evaluation of its associated rule.

Components of a Rule Each rule was composed of a well-defined set of components:

- Triggers: These defined when a rule’s logic should be evaluated. Two types of triggers were supported:
  - Sensor Triggers: The rule evaluation was initiated whenever the state of a linked sensor changed (e.g., a motion sensor detecting movement).
  - Periodic Triggers: The rule evaluation was initiated at a fixed time interval (e.g., every five minutes), which was useful for checking conditions like ambient temperature.

- Conditions and Connectives: The core logic of the rule was a set of conditions that evaluated to true or false. For instance, a condition could be Time is after 10 PM or Window sensor is 'closed'. To combine multiple conditions into a single outcome, the user could select one logical connective: `AND`, `OR`, `NAND`, or `NOR`. This accumulator would process all conditions to yield a final boolean result.
- Actions ("Then"/"Else"): Based on the final outcome of the condition evaluation, one of two action lists would be executed.
  - "Then" Actions: This was a list of actions to perform if the accumulated condition was true. For example, Turn device on and Set brightness to 75
  - "Else" Actions: This was an alternative list of actions to perform if the accumulated condition was false. This was critical for creating reverting logic, such as Turn device off when motion was no longer detected.

Rules A list of rules, which were specifically part of the initial requirements.

- "Office Light": `IF` *Powergrid Available* `THEN` *Turn on Light* `ELSE` *Turn off Light*
- "Water Temp.": `IF` *Solar Surplus* > 30 and *Water Temp.* $< 60°C$ `THEN` *Turn on Water Heater*; *Wait 30min.* `ELSE` *Turn off Water Heater*
- "Water Pump": `IF` *Water Tank* $< 30\%$ `THEN` *Turn on Water Pump*; *Wait 30min.* `ELSE` *Turn off Water Pump*
- "Load Battery": `IF` *Current Time* $\in$ *[01:30h, 03:30h]* `THEN` *Turn on Battery Charger* `ELSE` *Turn off Battery Charger*
- "Bathroom Temp.": `IF` *Bathroom Temp.* $< 5°C$ `THEN` *Turn on Bathroom Heater*; *Wait 1h* `ELSE` *Turn off Bathroom Heater*

Component Types List Here is a list of component types for rules.

Trigger Types

- `ONEMIN`: Trigger every minute.
- `TWOMIN`: Trigger every two minutes.
- `FIVEMIN`: Trigger every five minutes.
- `TENMIN`: Trigger every ten minutes.
- `FIFTEENMIN`: Trigger every 15 minutes.
- `TWENTYMIN`: Trigger every 20 minutes.
- `TWENTYFIVEMIN`: Trigger every 25 minutes.

- `THIRTYMINMIN`: Trigger every 30 minutes.
- `FORTYFIVEMIN`: Trigger every 45 minutes.
- `ONEH`: Trigger every hour.
- `TWOH`: Trigger every two hours.
- `SIXH`: Trigger every six hours.

Condition Types

- `NUMERICSTATE`: Check if the numeric state of a sensor is above a configurable value, below a configurable value, or both.
- `STATE`: Check if the state of a sensor yields on or off.
- `TIME`: Check if the current time is after a configurable value, before a configurable value, or both.

Action Types

- `SERVICE`: Set the device state to a configurable state (on or off).
- `DELAY`: Locks the device state to its current state for a configurable amount of time.

1.2.4 Analysis of Issues and Limitations

While functional at a basic level, this initial implementation of Honua suffered from significant architectural and conceptual issues that limited its scalability, maintainability, and ultimate utility for the user. Architectural Deficiencies

The dual-backend architecture, while a clever workaround for port forwarding, introduced severe problems:

- Code Duplication and Maintenance Overhead: A significant portion of the application's logic-handling user authentication, device models, and rule structures—had to be implemented and maintained in two separate codebases (the public and private backends). Any change to a feature required coordinated updates in both places, doubling the development effort and dramatically increasing the risk of introducing bugs.
- Synchronization Complexity and Latency: The system's reliability was entirely dependent on the constant, stable connection between the two backends. This introduced noticeable latency, as a user command had to make a round trip from the app, to the cloud, down to the Pi, and then to the device. Furthermore, it created complex synchronization challenges and race conditions, where the state cached in the public backend could easily become out of sync with the true state of the devices in the home, leading to a confusing and unreliable user experience.

- Single Point of Failure and Cost: The public backend represented a centralized point of failure. If the cloud service experienced an outage, all users would lose remote control capabilities simultaneously. This architecture also incurred ongoing cloud hosting costs for servers, databases, and data transfer.

1.2.5 Inflexibility of the Rule System

The automation engine was fundamentally too static and failed to cover a wide range of common smart home scenarios:

- The Rigid One-to-One Binding: This was the most critical limitation. Users could not create a single automation to control multiple devices. A "Good Night" scene that turns off three lights, locks the door, and lowers the thermostat was impossible to implement in a single rule. A user would need to create a separate, nearly identical rule for each device, which was both tedious and inefficient.
- Lack of Reusability: Because a rule was tied to a specific device, there was no way to create a reusable automation template. If a user wanted the same motion-based lighting logic in three different rooms, they had to manually recreate the entire rule three times.
- Limited Expressiveness: The simple `IF/THEN/ELSE` structure was not powerful enough for more advanced logic. Scenarios requiring `ELSE IF` conditions, delays between actions (e.g., "turn off the fan 10 minutes after the light is turned off"), or the ability to trigger other rules were not supported. The system failed to provide the customization and power necessary to fulfill a user's unique and evolving automation requirements.

2 Template System

To overcome the issues of the current rule system, a new smart home automation system was proposed and implemented.

2.1 The Problem

To formerly define the problem this new system wants to solve, we first must acknowledge that there are a multitude of different users. There are users with limited technical capabilities who would require a very easy automation system similar to the current one, there are uses with very advanced technical capabilities who want a very advanced and customizable system, and there might be users with limited abilities who want a simple view on an advanced system another person has setup for them. So, this system must cover all different and individual automation needs from simple to advanced while also giving a view on it which is also anything from easy to expert.

2.2 The Solution

The solution is the introduction of rule templates. Users have the options of either picking a template and filling out the required parameters, creating a template themselves for multi-use, or fully creating a rule (by creating a template and immediately instantiating it).

Now you can have very easy and simple automation configuration while also supporting very advanced configurations. You also support the ability of an advanced user setting up templates for other users to use who have lesser technical capabilities.

2.3 Template Structure

We must differentiate between templates and template instances. Firstly, the structure of the former is explained, then it is expanded to show how it is instantiated.

Templates are made up of a list of rules and a list of configuration variables. A rule has a condition and a list of actions. The condition is checked every second and decides whether the actions are executed or not. When adding a condition or action it might require you to set identifiers for variables which have to be set on instantiation. Rules also require you to set their initial state (on or off) as rules can be active or inactive and thus require a starting state.

On template instantiation you are only required to pass a list of values for the configuration variables, i.e., the identifiers of conditions and actions earlier. Of course, these configuration values have to be the appropriate data type for the condition or action they were assigned to. Next, all rules are instantiated such that their state (on or off) is now properly tracked.

Templates and their instances are all managed by the template manager, which allows you to add, remove, and instantiate templates or delete such instances. It is directly accessible over gRPC and, as such, also supports querying a full list of templates and template instances.

2.3.1 Comparison

The older system which this template system is replacing is fully covered.

What was defined as trigger and condition in the old system is now replaced by the condition of templates. The periodic timed triggers can be replaced using cron conditions and the ability of having multiple conditions aggregated is also supported due to the recursive nature of the new condition system where all the aggregate types (`OR`, `AND`, `NOR`, `NAND`, and also `NOT`) are a condition themselves which can be nested as desired.

The ability to delay actions by locking their state change is fully covered by the ability to turn off rules until a certain amount of time passes.

Component Types List Here is a list of component types of the new template system.

Data Types

- `BOOL`: Always `true` or `false`.
- `INT`: A 64-bit signed integer value (can be 32-bit on certain systems)
- `CRON`: A cron object, which yields whether the current time triggers this object or not.
- `NUMERIC_SENSOR`: The ID of a sensor which yields the current numeric value of a sensor.
- `SENSOR`: The ID of a sensor which yields the current boolean value of a sensor.
- `ENTITY_DEVICE`: The ID of a device, which exposes the ability to turn the device on or off.

Condition Types

- `IMMEDIATE`: This condition requires a `BOOL` variable and is fulfilled when this variable yields `true`.
- `NOT`: This condition requires a sub-condition and simply yields the negation of the result of this sub-condition.
- `OR`: This condition requires a list of sub-conditions and yields the `OR`-aggregation of the results of these sub-conditions.
- `AND`: This condition requires a list of sub-conditions and yields the `AND`-aggregation of the results of these sub-conditions.
- `NUMERIC_SENSOR`: Requires a `NUMERIC_SENSOR` variable and two `OptionalFloat` (one max and one min) and yields whether the value of the sensor is above the min (if given) and below the max (if given).
- `SENSOR`: Requires a `SENSOR` variable and a `BOOL` variable and yields `true` if the state of the sensor yields the value of the boolean variable.
- `TIME`: Requires two `INT` variables where both describe minutes of a 24h cycle ([0, 60*24] = [0, 1440]) and yields `true` if the current day-minute is between these variables.
- `CRON`: Requires a `cron` variable and yields `true` if the current time triggers this cron object.

Action Types

- `ENTITY_ON`: Requires an `ENTITY_DEVICE` variable and turns this entity on when executed.

- `ENTITY_OFF`: Requires an `ENTITY_DEVICE` variable and turns this entity off when executed.
- `RULE_ON`: Requires a rule ID and turns this rule on when executed.
- `RULE_OFF`: Requires a rule ID and turns this rule off when executed.

2.4 Specific Automations

Here is a description on how to implement the specific requirements listed previously in the new template system. For readability purposes there are no page-breaks within an automation definition.

2.4.1 "Office Light"

Exact Definition:

- Variables:
  - `CRON` *`interval`*: When to check state.
  - `SENSOR` *`power_grid`*: Sensor ID of the availability of the power grid as boolean.
  - `ENTITY_DEVICE` *`light_switch`*: Device ID of the light switch.
- Rule 1: Initial state is on.
  Condition: `AND([`
  - `CRON(`*`interval`*`)`
  - `SENSOR(`*`power_grid`*` == true)`

  `])`
  Actions:
  - `ENTITY_ON(`*`light_switch`*`)`
  - `RULE_ON(2)`
  - `RULE_OFF(1)`
- Rule 2: Initial state is off.
  Condition: `AND([`
  - `CRON(`*`interval`*`)`
  - `SENSOR(`*`power_grid`*` == false)`

  `])`
  Actions:
  - `ENTITY_OFF(`*`light_switch`*`)`
  - `RULE_ON(1)`
  - `RULE_OFF(2)`

Notes This is a very simple exclusive two-state template, which corresponds to the states of the light switch. The interval time should not be too fine otherwise the light may start flickering in sync with this interval.

2.4.2 "Water Temp."

Exact Definition:

- Variables:
  - `CRON` *`interval`*: When to check state.
  - `CRON` *`min_heat_interval`*: Interval time with the minimum amount of running time for the heater. Example: `"*/5 * * * *"` always runs the heater for at least 5 minutes and up to a maximum of twice the interval (10 minutes).
  - `INT` *`hi_solar_surplus`*: Solar surplus amount in kW such that if the solar surplus is above this value the water heater may be turned on.
  - `INT` *`lo_water_temp`*: Temperature in degrees Celsius such that if the temperature inside the water tank falls below this value the water heater may be turned on.
  - `SENSOR` *`solar_surplus`*: Sensor ID of the solar power surplus in kW.
  - `SENSOR` *`water_temp`*: Sensor ID of the water temperature inside the water tank in degrees Celsius.
  - `ENTITY_DEVICE` *`water_heater`*: Device ID of the heater warming up the water tank.
- Rule 1: Initial state is on.
  Condition: `AND([`
  - `CRON(`*`interval`*`)`
  - `NUMERIC_SENSOR(`*`solar_surplus > hi_solar_surplus`*`)`
  - `NUMERIC_SENSOR(`*`water_temp < lo_water_temp`*`)`

  `])`
  Actions:
  - `ENTITY_ON(`*`water_heater`*`)`
  - `RULE_OFF(1)`
  - `RULE_ON(2)`
- Rule 2: Initial state is off.
  Condition: `CRON(`*`min_heat_interval`*`)`
  Actions:
  - `RULE_OFF(2)`
  - `RULE_ON(3)`
- Rule 3: Initial state is off.
  Condition: `CRON(`*`min_heat_interval`*`)`
  Actions:
  - `ENTITY_ON(`*`water_heater`*`)`
  - `RULE_OFF(3)`
  - `RULE_ON(1)`

Notes To ensure the water pump running for at least a certain amount of time this template was solved in three states. The first state (rule) ensures the pump is turned on and then switches to the second state. This state only ticks down the remaining time until `min_heat_interval` is triggered at which point the third state waits for another trigger. This guarantees a full cycle of the interval defined in `min_heat_interval`.

2.4.3 “Water Pump”

Exact Definition:

- Variables:
  - `CRON` *`interval`*: When to check state.
  - `CRON` *`min_pump_interval`*: Interval time with the minimum amount of running time for the pump. Example: `"*/30 * * * *"` always runs the pump for at least 30 minutes and up to a maximum of twice the interval (60 minutes).
  - `INT` *`lo_water_tank`*: Tank level in percent such that if the tank level falls below this value the pump is turned on.
  - `SENSOR` *`water_tank`*: Sensor ID of the water tank level in percent.
  - `ENTITY_DEVICE` *`water_pump`*: Device ID of the water pump.
- Rule 1: Initial state is on.
  Condition: `AND([`
  - `CRON(`*`interval`*`)`
  - `NUMERIC_SENSOR(`*`water_tank`*` < `*`lo_water_tank`*`)`

  `])`
  Actions:
  - `ENTITY_ON(`*`water_pump`*`)`
  - `RULE_OFF(1)`
  - `RULE_ON(2)`
- Rule 2: Initial state is off.
  Condition: `CRON(`*`min_pump_interval`*`)`
  Actions:
  - `RULE_OFF(2)`
  - `RULE_ON(3)`
- Rule 3: Initial state is off.
  Condition: `CRON(`*`min_pump_interval`*`)`
  Actions:
  - `ENTITY_ON(`*`water_pump`*`)`
  - `RULE_OFF(3)`
  - `RULE_ON(1)`

Notes This template essentially works the same way as the “Water Temp.” template.

2.4.4 “Load Battery”

Exact Definition:

- Variables:
  - `CRON` *`start_time`*: When to start charging.
  - `CRON` *`stop_time`*: When to stop charging.
  - `ENTITY_DEVICE` *`battery`*: Device ID of the charger to turn on.
- Rule 1: Initial state is on.
  Condition: `CRON(`*`start_time`*`)`
  Actions:
  - `ENTITY_ON(`*`battery`*`)`
  - `RULE_OFF(1)`
  - `RULE_ON(2)`
- Rule 2: Initial state is off.
  Condition: `CRON(`*`stop_time`*`)`
  Actions:
  - `ENTITY_OFF(`*`battery`*`)`
  - `RULE_ON(1)`
  - `RULE_OFF(2)`

Notes Alternatively, the `TIME` condition can be used for both rules.

2.4.5 "Bathroom Temp."

Exact Definition:

- Variables:
  - `CRON` *`interval`*: When to check state.
  - `INT` *`lo_temp`*: Temperature in degrees Celsius such that if the sensor temperature falls below this value the heater is turned on.
  - `INT` *`hi_temp`*: Temperature in degrees Celsius such that if the sensor temperature goes above this value the heater is turned off.
  - `SENSOR` *`temp`*: Sensor ID of the bathroom temperature in degrees Celsius.
  - `ENTITY_DEVICE` *`heater`*: Device ID of the heater to turn on.
- Rule 1: Initial state is on.
  Condition: `AND([`
  - `CRON(`*`interval`*`)`
  - `NUMERIC_SENSOR(`*`temp < lo_temp)`*

  `])`
  Actions:
  - `ENTITY_ON(`*`heater`*`)`
  - `RULE_OFF(1)`
  - `RULE_ON(2)`
- Rule 2: Initial state is off.
  Condition: `AND([`
  - `CRON(`*`interval`*`)`
  - `NUMERIC_SENSOR(`*`temp > hi_temp)`*

  `])`
  Actions:
  - `ENTITY_OFF(`*`heater`*`)`
  - `RULE_ON(1)`
  - `RULE_OFF(2)`

Notes To ensure the heater is not constantly being turned on and off again one should ensure $hi\_temp - lo\_temp \geq 1$.

2.5 Implementation Specifics

This section dives into the specific implementation of the template system in the Honua backend [8]. To reiterate, the code is written in Go and it uses both structs with receiver methods and generic interfaces. Often, it uses a type-based approach which is required for serialization. Generally speaking, the code was designed to be loosely coupled and easily extendable.

2.5.1 Templates & Rules

The entire rule system is executed by simply running the `Tick` method of a `TemplateManager` instance (of which there is a singleton) as referenced in Appendix A.21. It simply iterates over all template instances, checks whether or not they are active, and executes them if so. This method ensures execution exactly once per second and accounts for being called too quickly or too slowly.

It is very important to understand that the `Template` struct refers to a template—a blueprint—and `TemplateInstance` contains this blueprint together with all the configurations needed to run it as referenced in Appendix A.22. A template instance is executed by running the `Execute` method where all rules, which are active are executed. For this it logs first which rules are active and then executes all of them otherwise there could be a case where rule i changes the state of rule $i + 1$ during its iteration and when $i + 1$ is now checked this yields a different result than before.

For rules, there are again `Rule` and `RuleInstance` structs as referenced in Appendix A.19. The `Execute` method of a rule instance first checks whether or not the condition is satisfied and, if so, executes all actions.

An `IContext` instance (Appendix A.10) is created in the template instance on every execution that exposes access to different parts of the system, namely variables, the database, and the Home Assistant API and it is passed down to all lower components.

Actions are defined by the `IAction` interface and are type-based as referenced in Appendix A.1. Every action type is defined in the `.proto` file and registered in the `ActionFromService` function which is used for deserializing. The action method `ToService` is for serialization.

Conditions are also type-based. Each condition implements the `ICondition` interface in Appendix A.9. Again, every condition type is defined in the `.proto` file and registered in the `ConditionFromService` function.

Conditions often use context variables for configuration, which are defined in `ctxvar` as referenced in Appendix A.11. A context variable is either a static value represented by the struct `ImmediateCtxVar` (used to “hardcode” values into templates) or it is derived from the context where it is represented by the struct `ReferencedCtxVar` (used to make values in templates configurable).

Each context variable, of course, is defined on a data type. Data types are

```
docker build
--platform linux/amd64,linux/arm64
--tag TEMP_LAYERNAME
.
```

Figure 4: Building a Docker Image

```
docker tag TEMP_LAYERNAME USERNAME/LAYERNAME
```

Figure 5: Tagging a Docker Image

type-based singletons. They all have a defined type in the `.proto` file and all implement the `IDataType` interface (Appendix A.18). Every singleton is registered in the `GetDataType` function and has an additional optional convenience function to resolve context variables quickly (usually named `GetValue`).

3 Installation & Deployment

3.1 Raspberry Pi

To install the Honua docker stack on the Raspberry Pi you first must install Raspberry Pi OS on it. Setup WiFi access properly such that the Pi can connect to it and enable SSH. For SSH credentials only use lowercase letters for username and password as digits and any form of symbols causes issues.

After everything is done, connect the Pi to a power source. It should now automatically connect to WiFi and then be accessible via SSH. Using this access, you can now install Docker on it and start the Honua Docker stack as usual.

3.2 Creating Docker Images on Docker Hub

Firstly, you must have a Docker Hub account and be ready to build the image. So, to start you build it using the command shown in Figure 4. For the platforms, `amd64` is used to support widespread systems and `arm64` is required to be able to run on the Raspberry Pi. The `TEMP_LAYERNAME` is just the temporary name to identify this build on the local machine. The `.` is the folder, which to build (hence this works by running it inside the build folder).

Next, you must tag this image by using the command shown in Figure 6. `USERNAME` is your Docker Hub username and `LAYERNAME` is what its final

```
docker tag TEMP_LAYERNAME USERNAME/LAYERNAME
```

Figure 6: Tagging a Docker Image

```
docker push USERNAME/LAYERNAME
```

Figure 7: Pushing a Docker Image

```
docker compose up
```

Figure 8: Running a Docker Stack

identifier is going to be.

Lastly, push the image to Docker Hub using the command shown in Figure 7. Now, the image is available under that identifier for your Docker stacks.

3.3 Deploying the Docker Stack

To deploy and run a Docker stack all that is needed is the `.yaml` file that defines it and the command in either Figure 8 or Figure 9 where the `-d` option means "detached" from the current console (the command must be run next to the `.yaml` file). To stop it again simply run the command shown in Figure 10.

3.4 gRPC & Protocol Buffer (`protoc`)

To generate the protocol code for the Honua backend you have to successfully install protoc and then run the command shown in Figure 11 (this command can also be seen in the `MAKEFILE` file).

4 More Solutions

Here we explain other things which were done as part of this project but can be seen as separate sub-projects.

4.1 Template Frontend App Prototype

To present the template system a frontend prototype was created. This prototype mimics functional templates and lets you configure them. Figure 12a shows a simple overview over all configured and active rules. By clicking on that rule you get to the screen shown in Figure 12b which essentially opens a form based on the required input of the template described previously as "Office Light".

```
docker compose up -d
```

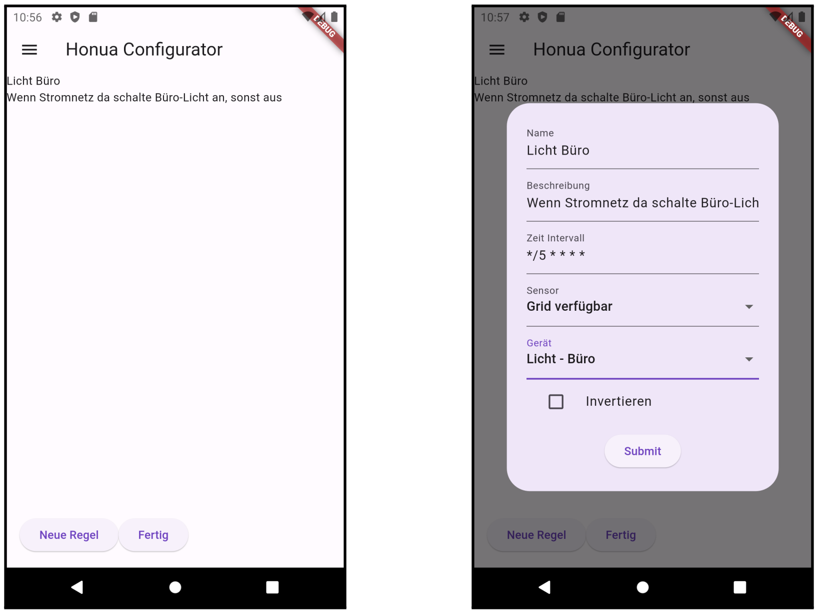
Figure 9: Running a Docker Stack in Detached Mode

```
docker compose down
```

Figure 10: Stopping a Docker Stack in Detached Mode

```
protoc
--go_out=.
--go-grpc_out=.
./resources/*.proto
```

Figure 11: Generating Honua `protoc` Code



Honua Configurator

Licht Büro
Wenn Stromnetz da schalte Büro-Licht an, sonst aus

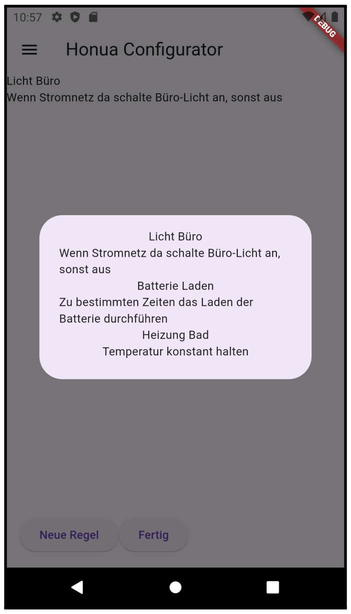
Honua Configurator

Licht Büro
Wenn Stromnetz da schalte Büro-Licht an, sonst aus

Name: Licht Büro
Beschreibung: Wenn Stromnetz da schalte Büro-Lich
Zeit Intervall: */5 * * * *
Sensor: Grid verfügbar
Gerät: Licht - Büro
Invertieren [ ]

(a) Overview Screen (b) Edit Template Screen

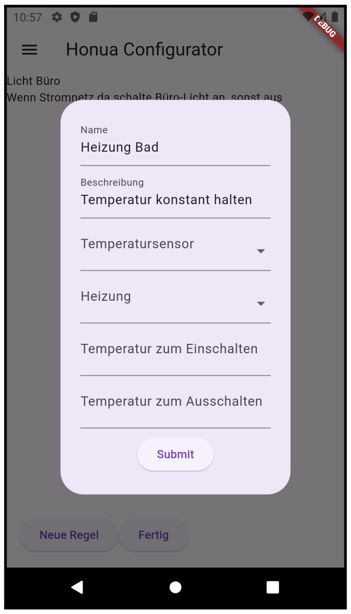
Figure 12: Prototype Overview & Edit Rule Screens



Licht Büro
Wenn Stromnetz da schalte Büro-Licht an, sonst aus

Licht Büro
Wenn Stromnetz da schalte Büro-Licht an, sonst aus
Batterie Laden
Zu bestimmten Zeiten das Laden der Batterie durchführen
Heizung Bad
Temperatur konstant halten

(a) New Rule, Select Template Screen



Licht Büro
Wenn Stromnetz da schalte Büro-Licht an, sonst aus

Name: Heizung Bad
Beschreibung: Temperatur konstant halten
Temperatursensor
Heizung
Temperatur zum Einschalten
Temperatur zum Ausschalten

(b) Configure Template Screen

Figure 13: Prototype New Rule Configuration

The first two fields are always “Name” and “Description” for identification reasons for the user. Next, we generate dynamical form fields based on the inputs (and their data types) required by the template. The first one is a cron input. Of course, one could implement a proper screen here for cron configuration but for sakes of a prototype this is sufficient. Next, we select sensors and devices. The list of sensors and devices to choose from is gathered when starting the app and logging in. Finally, there is an additional field not mentioned previously which represents a boolean (to invert the sensor state).

New rules can also be added as shown in Figure 13a which shows a template selection and leads to Figure 13b where you get to another form as shown before. The last two fields of this form are integer inputs.

This prototype serves to show how powerful this template system can be where very powerful and unique rule templates can be created but all that is shown to the user is a very simple configuration screen where each input entry is dynamically generated based on its data type.

4.2 cron Package

A separate and independent cron go package was created to add a standard cron job implementation to the project. It features wildcards ($\star$), value list separators (`,`), range of values (`-`), and step values (`/`) for minute, hour, day of

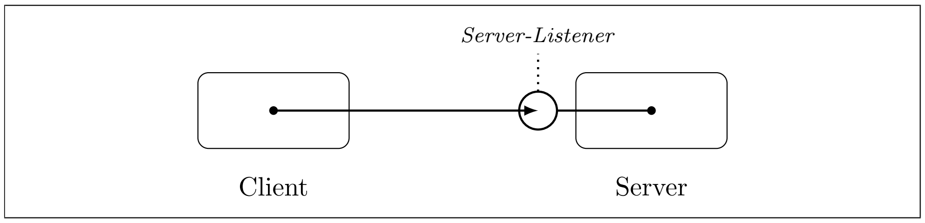


Figure 14: Client-Server Connection with Port Forwarding

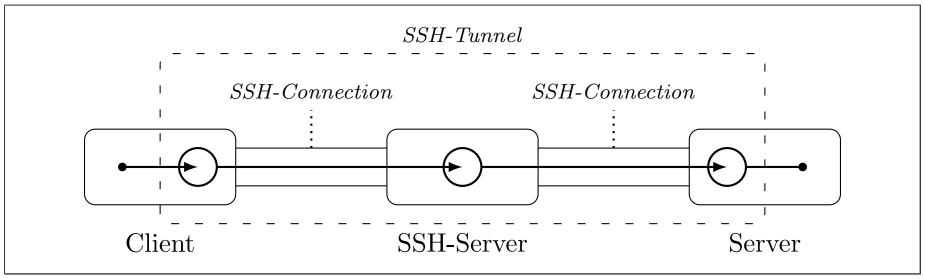


Figure 15: Client-Server Connection over SSH-Tunnel

month, month, and day of week. It supports deserializing cron strings to objects and vice versa and it works with go `time.Time` values [9].

`github.com/CAS-ual-TY/simple_go_cron` is the module name and the value used for importing the whole package.

4.3 SSH Tunnel Package & Image

`github.com/CAS-ual-TY/sshtunnel2` is another go package made specifically for this project. It allows you to create SSH tunnels to forward connections and circumvent port forwarding requirements. This way the entire public backend is not needed anymore as the frontend is now able to establish a connection to the private backend [10].

First, the server creates a reverse connection to an SSH server, i.e. it establishes an ordinary SSH connection with credentials and then it listens for a local connection on that SSH server. Anything sent to that connection is then transported via SSH to the server and forwarded to a target.

Once the server has established a listener on the SSH server, the client connects via SSH and listens locally (on the client) for a connection. Again, everything sent to that connection is forwarded to the SSH server into a local target, which in this case needs to be the listener of the server.

The application now simply targets the local SSH connection instead of the direct server address to reach the server. This allows the server to become reachable without any port forwarding whatsoever as shown in Figure 15.

The entire SSH functionality is also provided as docker images in the docker hub as `plusluist/sshtunnel2_client` (client to SSH server) and, of course, `plusluist/sshtunnel2_server` (SSH server to server). They are configured via the following environment variables:

- `SSHHOST`: The SSH server host.
- `SSHPORT`: The SSH server port.
- `SSHUSER`: The SSH user.
- `SSHPASSWORD`: The SSH password.
- `LPORT`: The local port, i.e. where the tunnel listens at on the client or where the tunnel forwards to on the server.
- `RPORT`: The SSH local connection port. Must be the same on client and server.
- `TARGET`: Where to forward to on the server (default is `localhost`) in case it is not local.

`plusluist/ssh_server` is another docker hub image which only provides a local SSH server to use the tunnel on.

4.4 Additional Honua Changes

As mentioned already, the entire public backend was removed and, as such, any code that functioned as synchronization between private and public backend.

It is also worth mentioning that the original rule system was first cleaned up and rewritten before it was eventually replaced by the template system entirely.

Additionally, a bunch of general improvements were made. The most prominent example is the proper listening to an interrupt signal by the operating system for shutdown.

4.5 Future Extension Possibilities

Finally, we discuss a bunch of extension possibilities.

Test Context The `IContext` interface could be changed such that conditions and actions do not access the database directly but rather do all state checks and modifications on the interface. This would allow a separate implementation of this context with emulated sensors and devices.

Context Variables Right now context variables only serve as settings for conditions and actions. Additional conditions and actions could be introduced which simply create, compare, and modify such variables. These variables could

```
ssh -N -L
9000:localhost:8080
user@sshserver.com -p 22
```

Figure 16: SSH Forward Tunnel Command on Debian

```
ssh -N -R
8080:localhost:9000
user@sshserver.com -p 22
```

Figure 17: SSH Reverse Tunnel Command on Debian

then be set to be persistent with restarts or not. This would allow things like counters or countdowns.

Lazy Evaluation on Aggregate Conditions Aggregate condition types could be expanded with another boolean variable which defines whether or not they are lazily evaluated (for the case where condition checks modify the state or force a reloading of cached states).

Optional Data Types Add proper support for optionals and register them as data type. Right now these are only used for the `TimeCondition` and can only be hardcoded in the template. The reason being that this condition type can be fully replaced by the `CronCondition`.

More State Refreshing Right now it is possible that multiple template instance ticks are being done without a state check in between. For this, the loop which controls template execution to happen exactly once per second needs to be executed higher in the structure to include state refreshing. The real question to ask would be whether or not a state check every second would actually be needed or not.

Frontend Views The backend was designed with the ability in mind to have different views on the same system. The best example for this is a graphical user interface that lets the user configure a week-day based schedule. In reality this schedule would be implemented using aggregation conditions in combination with cron conditions.

SSH Tunnel Improvement There are two different improvements that can be made to the SSH tunnel image of the Docker stack:

- The `TARGET` is only used server-side right now. Extend it such that on client-side this is the IP that is listened on. Systems can be connected to different networks and this would allow a precise choice (right now the client always listens on `localhost`).

- It might be possible to replace the entire Go application with pure SSH commands which are shown in Figures 16 and 17 where
  - `9000` is `LPORT`,
  - `localhost` is `TARGET`,
  - `8080` is `RPORT`,
  - `user` is `SSHUSER`,
  - `sshserver.com` is `SSHHOST`,
  - and `22` is `SSHPORT`.

  Of course, this includes the previously mentioned client-side improvement to `TARGET`.

  It is important to note that while it is easy to establish a tunnel using these commands, the difficulty in creating a Docker image based only on these commands is having it be resilient in case the SSH connection is lost. In such cases, it must be reestablished quickly and securely.

  The advantages would be having less code to maintain and, if the image is really able to reconnect on its own, it would be faster and more resilient (assuming that native Debian SSH code works better than custom Go code).

5 References

[1] Open Home Foundation. Documentation - Home Assistant.
`https://www.home-assistant.io/docs/`, 2025.

[2] The Go Authors. Documentation - The Go Programming Language.
`https://go.dev/doc/`, 2025.

[3] gRPC Authors. Documentation | gRPC.
`https://grpc.io/docs/`, 2025.

[4] The PostgreSQL Global Development Group. PostgreSQL: Documentation.
`https://www.postgresql.org/docs/`, 2025.

[5] MongoDB Inc. What is MongoDB? - Database Manual - MongoDB Docs.
`https://www.mongodb.com/docs/manual/`, 2025.

[6] Docker Inc. Docker Docs.
`https://docs.docker.com/`, 2025.

[7] Raspberry Pi Ltd. Raspberry Pi Documentation.
`https://www.raspberrypi.com/documentation/`, 2025.

[8] Jonas Bordewick, Luis Thiele. Honua Backend Source Code.
`https://github.com/CAS-ual-TY/honua_local_backend`, 2025.

[9] Luis Thiele. github.com/CAS-ual-TY/simple_go_gron go package.
`https://github.com/CAS-ual-TY/simple_go_cron`, 2025.

[10] Luis Thiele. github.com/CAS-ual-TY/sshtunnel2 go package.
`https://github.com/CAS-ual-TY/sshtunnel2`, 2025.

A Template System

A.1 action.go

```go
package rules

import (
        "errors"
        "github.com/CAS-ual-TY/honua_local_backend/service"
        "time"
)

type ActionType service.Action_ActionType

func (t ActionType) ToService() service.Action_ActionType {
        return service.Action_ActionType(t)
}

type IAction interface {
        GetType() ActionType
        Execute(ctx IContext, t time.Time) error
        ToService() (*service.Action, error)
}

type Action struct{}

type TypeBasedAction struct {
        Type ActionType
}

func (a *TypeBasedAction) GetType() ActionType {
        return a.Type
}

type SetEntityStateAction struct {
        TypeBasedAction
        Entity ICtxVar
}

func (a *SetEntityStateAction) Execute(ctx IContext, t time.Time) error {
        entity, err := DeviceDataType.GetValue(a.Entity, ctx)
        if err != nil {
                return err
        }
        if a.Type.ToService() == service.Action_ENTITY_ON {
                return ctx.GetAPI().HomeassistantGenericTurnOn(entity)
        } else if a.Type.ToService() == service.Action_ENTITY_OFF {
                return ctx.GetAPI().HomeassistantGenericTurnOff(entity)
        } else {
                return errors.New("invalid action")
        }
}

func (a *SetEntityStateAction) ToService() (*service.Action, error) {
        entity, err := a.Entity.ToService()
        if err != nil {
                return nil, err
        }
        if a.Type.ToService() == service.Action_ENTITY_ON {
```

```go
			return &service.Action{
				ActionType: service.Action_ENTITY_ON,
				Action: &service.Action_EntityAction{
					EntityAction: &service.EntityAction{
						EntityId: entity,
					},
				},
			}, nil
		} else if a.Type.ToService() == service.Action_ENTITY_OFF {
			return &service.Action{
				ActionType: service.Action_ENTITY_OFF,
				Action: &service.Action_EntityAction{
					EntityAction: &service.EntityAction{
						EntityId: entity,
					},
				},
			}, nil
		} else {
			return nil, errors.New("invalid action")
		}
}

func EntityStateActionFromService(a *service.Action) (*SetEntityStateAction, error) {
		entity, err := CtxVarFromService(a.GetEntityAction().GetEntityId())
		if err != nil {
			return nil, err
		}
		return &SetEntityStateAction{
			TypeBasedAction: TypeBasedAction{
				Type: ActionType(a.ActionType),
			},
			Entity: entity,
		}, nil
}

type SetRuleStateAction struct {
		TypeBasedAction
		RuleID uint32
}

func (a *SetRuleStateAction) Execute(ctx IContext, t time.Time) error {
		if a.Type.ToService() == service.Action_RULE_ON {
			return ctx.EnableRule(a.RuleID)
		} else if a.Type.ToService() == service.Action_RULE_OFF {
			return ctx.DisableRule(a.RuleID)
		} else {
			return errors.New("invalid action")
		}
}

func (a *SetRuleStateAction) ToService() (*service.Action, error) {
		if a.Type.ToService() == service.Action_RULE_ON {
			return &service.Action{
				ActionType: service.Action_RULE_ON,
				Action: &service.Action_RuleAction{
					RuleAction: &service.RuleAction{
						RuleId: a.RuleID,
					},
				},
			}, nil
```

```go
	} else if a.Type.ToService() == service.Action_RULE_OFF {
		return &service.Action{
			ActionType: service.Action_RULE_OFF,
			Action: &service.Action_RuleAction{
				RuleAction: &service.RuleAction{
					RuleId: a.RuleID,
				},
			},
		}, nil
	} else {
		return nil, errors.New("invalid action")
	}
}

func RuleActionFromService(a *service.Action) (*SetRuleStateAction, error) {
	return &SetRuleStateAction{
		TypeBasedAction: TypeBasedAction{
			Type: ActionType(a.ActionType),
		},
		RuleID: a.GetRuleAction().GetRuleId(),
	}, nil
}

func ActionFromService(a *service.Action) (IAction, error) {
	switch a.GetActionType() {
	case service.Action_ENTITY_ON:
		return EntityStateActionFromService(a)
	case service.Action_ENTITY_OFF:
		return EntityStateActionFromService(a)
	case service.Action_RULE_ON:
		return RuleActionFromService(a)
	case service.Action_RULE_OFF:
		return RuleActionFromService(a)
	}

	return nil, ErrUnknownType
}
```

A.2 condition_aggregate.go

```go
package rules

import (
	"github.com/CAS-ual-TY/honua_local_backend/service"
	"time"
)

type AggregateCondition struct {
	TypeBasedCondition
	Conditions []ICondition
}

func (c *AggregateCondition) CheckCondition(ctx IContext, t time.Time) (bool, error) {
	switch service.Condition_ConditionType(c.Type) {
	case service.Condition_OR:
		for _, cond := range c.Conditions {
			result, err := cond.CheckCondition(ctx, t)
			if err != nil {
				return false, err
			}
			if result {
				return true, nil
			}
		}
		return false, nil
	case service.Condition_AND:
		for _, cond := range c.Conditions {
			result, err := cond.CheckCondition(ctx, t)
			if err != nil {
				return false, err
			}
			if !result {
				return false, nil
			}
		}
		return true, nil
	default:
		return false, nil
	}
}

func (c *AggregateCondition) ToService() (*service.Condition, error) {
	conditions := make([]*service.Condition, len(c.Conditions))
	for _, co := range c.Conditions {
		cos, err := co.ToService()
		if err != nil {
			return nil, err
		}
		conditions = append(conditions, cos)
	}

	return &service.Condition{
		ConditionType: service.Condition_ConditionType(c.GetType()),
		Condition: &service.Condition_AggregateCondition{
			AggregateCondition: &service.AggregateCondition{
				SubConditions: conditions,
			},
		},
		}, nil
}

func AggregateConditionFromService(c *service.Condition) (*AggregateCondition, error)
↪	{
	conditions := make([]ICondition,
↪	len(c.GetAggregateCondition().GetSubConditions()),
↪	len(c.GetAggregateCondition().GetSubConditions()))
	for i, c1 := range c.GetAggregateCondition().GetSubConditions() {
		c2, err := ConditionFromService(c1)
		if err != nil {
			return nil, err
		}
		conditions[i] = c2
	}
	return &AggregateCondition{
		TypeBasedCondition: TypeBasedCondition{
			Condition: Condition{},
			Type:      ConditionType(c.ConditionType),
		},
		Conditions: conditions,
	}, nil
}
```

A.3 condition_cron.go

```go
package rules

import (
	"github.com/CAS-ual-TY/honua_local_backend/service"
	"time"
)

type CronCondition struct {
	Condition
	Cron ICtxVar
}

func (c *CronCondition) CheckCondition(ctx IContext, t time.Time) (bool, error) {
	cron, err := CronDataType.GetValue(c.Cron, ctx)
	if err != nil {
		return false, err
	}
	return cron.TimeMatches(t), nil
}

func (c *CronCondition) GetType() ConditionType {
	return ConditionType(service.Condition_CRON)
}

func (c *CronCondition) ToService() (*service.Condition, error) {
	cron, err := c.Cron.ToService()
	if err != nil {
		return nil, err
	}
	return &service.Condition{
		ConditionType: service.Condition_ConditionType(c.GetType()),
		Condition: &service.Condition_CronCondition{
			CronCondition: &service.CronCondition{
				Cron: cron,
			},
		},
	}, err
}

func CronConditionFromService(c *service.Condition) (*CronCondition, error) {
	cron, err := CtxVarFromService(c.GetCronCondition().GetCron())
	if err != nil {
		return nil, err
	}
	return &CronCondition{
		Condition: Condition{},
		Cron:      cron,
	}, nil
}
```

A.4 condition_immediate.go

```go
package rules

import (
	"github.com/CAS-ual-TY/honua_local_backend/service"
	"time"
)

type ImmediateCondition struct {
	Condition
	Bool ICtxVar
}

func (c *ImmediateCondition) CheckCondition(ctx IContext, t time.Time) (bool, error) {
	return BoolDataType.GetValue(c.Bool, ctx)
}

func (c *ImmediateCondition) GetType() ConditionType {
	return ConditionType(service.Condition_IMMEDIATE)
}

func (c *ImmediateCondition) ToService() (*service.Condition, error) {
	v, err := c.Bool.ToService()

	if err != nil {
		return nil, err
	}

	return &service.Condition{
		ConditionType: service.Condition_ConditionType(c.GetType()),
		Condition: &service.Condition_ImmediateCondition{
			ImmediateCondition: &service.ImmediateCondition{
				Bool: v,
			},
		},
	}, nil
}

func ImmediateConditionFromService(c *service.Condition) (*ImmediateCondition, error)
↪ {
	bool, err := CtxVarFromService(c.GetImmediateCondition().GetBool())
	if err != nil {
		return nil, err
	}
	return &ImmediateCondition{
		Condition: Condition{},
		Bool:      bool,
	}, nil
}
```

A.5 condition_not.go

```go
package rules

import (
	"github.com/CAS-ual-TY/honua_local_backend/service"
	"time"
)

type NotCondition struct {
	Condition
	SubCondition ICondition
}

func (c *NotCondition) CheckCondition(ctx IContext, t time.Time) (bool, error) {
	result, err := c.SubCondition.CheckCondition(ctx, t)
	if err != nil {
		return false, err
	}
	return !result, nil
}

func (c *NotCondition) GetType() ConditionType {
	return ConditionType(service.Condition_NOT)
}

func (c *NotCondition) ToService() (*service.Condition, error) {
	sub, err := c.SubCondition.ToService()
	if err != nil {
		return nil, err
	}
	return &service.Condition{
		ConditionType: service.Condition_ConditionType(c.GetType()),
		Condition: &service.Condition_NotCondition{
			NotCondition: &service.NotCondition{
				SubCondition: sub,
			},
		},
	}, nil
}

func NotConditionFromService(c *service.Condition) (*NotCondition, error) {
	sub, err := ConditionFromService(c.GetNotCondition().GetSubCondition())
	if err != nil {
		return nil, err
	}
	return &NotCondition{
		Condition:    Condition{},
		SubCondition: sub,
	}, nil
}
```

A.6 condition_numeric_sensor.go

```go
package rules

import (
	"github.com/CAS-ual-TY/honua_local_backend/service"
	"time"
)

type NumericSensorCondition struct {
	Condition
	NumericSensor ICtxVar
	Minimum       OptionalFloat
	Maximum       OptionalFloat
}

func (c *NumericSensorCondition) CheckCondition(ctx IContext, t time.Time) (bool, error) {
	sensor, err := NumericSensorDataType.GetValue(c.NumericSensor, ctx)
	if err != nil {
		return false, err
	}
	if c.Minimum.IsPresent() {
		if c.Minimum.Get() > sensor {
			return false, nil
		}
	}
	if c.Maximum.IsPresent() {
		if c.Maximum.Get() < sensor {
			return false, nil
		}
	}
	return true, nil
}

func (c *NumericSensorCondition) GetType() ConditionType {
	return ConditionType(service.DataTypeType_NUMERIC_SENSOR)
}

func (c *NumericSensorCondition) ToService() (*service.Condition, error) {
	sensor, err := c.NumericSensor.ToService()
	if err != nil {
		return nil, err
	}
	return &service.Condition{
		ConditionType: service.Condition_ConditionType(c.GetType()),
		Condition: &service.Condition_NumericStateCondition{
			NumericStateCondition: &service.NumericStateCondition{
				NumericSensor: sensor,
				Minimum:       c.Minimum.ToService(),
				Maximum:       c.Maximum.ToService(),
			},
		},
	}, nil
}

func NumericSensorConditionFromService(c *service.Condition) (*NumericSensorCondition, error) {
	numericSensor, err := CtxVarFromService(c.GetNumericStateCondition().GetNumericSensor())
```

```go
		if err != nil {
			return nil, err
		}
		minimum := OptionalFloatFromService(c.GetNumericStateCondition().Minimum)
		maximum := OptionalFloatFromService(c.GetNumericStateCondition().Maximum)
		return &NumericSensorCondition{
			Condition:     Condition{},
			NumericSensor: numericSensor,
			Minimum:       minimum,
			Maximum:       maximum,
		}, nil
}
```

A.7 condition_sensor.go

```go
package rules

import (
	"github.com/CAS-ual-TY/honua_local_backend/service"
	"time"
)

type SensorCondition struct {
	Condition
	Sensor        ICtxVar
	RequiredState ICtxVar
}

func (c *SensorCondition) CheckCondition(ctx IContext, t time.Time) (bool, error) {
	sensor, err := SensorDataType.GetValue(c.Sensor, ctx)
	if err != nil {
		return false, err
	}
	requiredState, err := BoolDataType.GetValue(c.RequiredState, ctx)
	if err != nil {
		return false, err
	}
	return sensor == requiredState, nil
}

func (c *SensorCondition) GetType() ConditionType {
	return ConditionType(service.DataTypeType_SENSOR)
}

func (c *SensorCondition) ToService() (*service.Condition, error) {
	sensor, err := c.Sensor.ToService()
	if err != nil {
		return nil, err
	}
	requiredState, err := c.RequiredState.ToService()
	if err != nil {
		return nil, err
	}
	return &service.Condition{
		ConditionType: service.Condition_ConditionType(c.GetType()),
		Condition: &service.Condition_StateCondition{
			StateCondition: &service.StateCondition{
				Sensor:        sensor,
				RequiredState: requiredState,
			},
		},
	}, nil
}

func SensorConditionFromService(c *service.Condition) (*SensorCondition, error) {
	sensor, err := CtxVarFromService(c.GetStateCondition().GetSensor())
	if err != nil {
		return nil, err
	}
	requiredState, err :=
	↪ CtxVarFromService(c.GetStateCondition().GetRequiredState())
	if err != nil {
		return nil, err
```

```
        }
        return &SensorCondition{
                Condition:     Condition{},
                Sensor:        sensor,
                RequiredState: requiredState,
        }, nil
}
```

A.8 condition_time.go

```go
package rules

import (
	"github.com/CAS-ual-TY/honua_local_backend/service"
	"time"
)

type TimeCondition struct {
	Condition
	IntMinuteStart ICtxVar
	IntMinuteEnd   ICtxVar
}

func (c *TimeCondition) CheckCondition(ctx IContext, t time.Time) (bool, error) {
	start, err := IntDataType.GetValue(c.IntMinuteStart, ctx)
	if err != nil {
		return false, err
	}
	end, err := IntDataType.GetValue(c.IntMinuteStart, ctx)
	if err != nil {
		return false, err
	}

	minute := t.Hour()*60 + t.Minute()

	if start <= end {
		return minute >= start && minute < end, nil
	} else {
		return minute >= start || minute < end, nil
	}
}

func (c *TimeCondition) GetType() ConditionType {
	return ConditionType(service.Condition_TIME)
}

func (c *TimeCondition) ToService() (*service.Condition, error) {
	start, err := c.IntMinuteStart.ToService()
	if err != nil {
		return nil, err
	}
	end, err := c.IntMinuteEnd.ToService()
	if err != nil {
		return nil, err
	}

	return &service.Condition{
		ConditionType: service.Condition_ConditionType(c.GetType()),
		Condition: &service.Condition_TimeCondition{
			TimeCondition: &service.TimeCondition{
				IntMinuteStart: start,
				IntMinuteEnd:   end,
			},
		},
	}, nil
}

func TimeConditionFromService(c *service.Condition) (*TimeCondition, error) {
	start, err := CtxVarFromService(c.GetTimeCondition().GetIntMinuteStart())
	if err != nil {
		return nil, err
	}
	end, err := CtxVarFromService(c.GetTimeCondition().GetIntMinuteEnd())
	if err != nil {
		return nil, err
	}
	return &TimeCondition{
		Condition:      Condition{},
		IntMinuteStart: start,
		IntMinuteEnd:   end,
	}, nil
}
```

A.9 condition.go

```go
package rules

import (
	"github.com/CAS-ual-TY/honua_local_backend/service"
	"time"
)

type ConditionType service.Condition_ConditionType

func (t ConditionType) ToService() service.Condition_ConditionType {
	return service.Condition_ConditionType(t)
}

type ICondition interface {
	GetType() ConditionType
	CheckCondition(ctx IContext, t time.Time) (bool, error)
	ToService() (*service.Condition, error)
}

type Condition struct{}

type TypeBasedCondition struct {
	Condition
	Type ConditionType
}

func (c *TypeBasedCondition) GetType() ConditionType {
	return c.Type
}

func ConditionFromService(c *service.Condition) (ICondition, error) {
	switch c.GetConditionType() {
	case service.Condition_IMMEDIATE:
		return ImmediateConditionFromService(c)
	case service.Condition_NOT:
		return NotConditionFromService(c)
	case service.Condition_OR:
		return AggregateConditionFromService(c)
	case service.Condition_AND:
		return AggregateConditionFromService(c)
	case service.Condition_NUMERIC_SENSOR:
		return NumericSensorConditionFromService(c)
	case service.Condition_SENSOR:
		return SensorConditionFromService(c)
	case service.Condition_TIME:
		return TimeConditionFromService(c)
	case service.Condition_CRON:
		return CronConditionFromService(c)
	default:
		break
	}

	return nil, ErrUnknownType
}
```

A.10 context.go

```go
package rules

import (
	"github.com/CAS-ual-TY/honua_local_backend/database"
	"github.com/CAS-ual-TY/honua_local_backend/homeassistant_api"
	"github.com/CAS-ual-TY/honua_local_backend/server"
)

type IContext interface {
	GetDB() *database.HonuaDB
	GetAPI() *homeassistant_api.HomeAssistantAPI
	EnableRule(ruleID uint32) error
	DisableRule(ruleID uint32) error
	GetReference(id string, dataTypeType DataTypeType) (any, error) // generic
↪  methods not possible yet in go: https://github.com/golang/go/issues/49085
}

type Context struct {
	server   *server.HonuaServer
	template *TemplateInstance
}

func (ctx *Context) GetDB() *database.HonuaDB {
	return ctx.server.GetDB()
}

func (ctx *Context) GetAPI() *homeassistant_api.HomeAssistantAPI {
	return ctx.server.GetAPI()
}

func (ctx *Context) EnableRule(ruleID uint32) error {
	return ctx.template.EnableRule(ruleID)
}

func (ctx *Context) DisableRule(ruleID uint32) error {
	return ctx.template.DisableRule(ruleID)
}

func (ctx *Context) GetReference(id string, dataTypeType DataTypeType) (any, error) {
	return ctx.template.GetReference(id, dataTypeType)
}
```

A.11 ctxvar.go

```go
package rules

import (
	"errors"
	"github.com/CAS-ual-TY/honua_local_backend/service"
)

var ErrWrongCtxVarType error = errors.New("ctx var is wrong data type")
var ErrUnknownDataType error = errors.New("unknown data type")

type ICtxVar interface {
	GetVarID() string
	GetDataType() IDataType
	GetValue(ctx IContext) (any, error)
	TypeAssert(immediate func(*ImmediateCtxVar), referenced func(ctxVar
↪	*ReferencedCtxVar))
	ToService() (*service.CtxVar, error)
}

type CtxVar struct {
	VarName  string
	VarID    string
	DataType IDataType
}

func CtxVarFromService(v0 *service.CtxVar) (ICtxVar, error) {
	dtt := DataTypeType(v0.Type)
	dt := GetDataType(dtt)
	if dt == nil {
		return nil, ErrUnknownDataType
	}

	v := CtxVar{
		VarName:  v0.Name,
		VarID:    v0.Id,
		DataType: dt,
	}

	if v0.GetReference() != nil {
		return ReferencedCtxVarFromService(v, v0)
	} else if v0.GetImmediate() != nil {
		return ImmediateCtxVarFromService(v, v0)
	}

	return nil, errors.New("can not deserialize ctx var")
}

func (v *CtxVar) GetVarID() string {
	return v.VarID
}

func (v *CtxVar) GetDataType() IDataType {
	return v.DataType
}

type ImmediateCtxVar struct {
	CtxVar
	Value any
}

func (v *ImmediateCtxVar) GetValue(ctx IContext) (any, error) {
	return v.Value, nil
}

func (v *ImmediateCtxVar) TypeAssert(immediate func(*ImmediateCtxVar), referenced
↪  func(ctxVar *ReferencedCtxVar)) {
	immediate(v)
}

func (v *ImmediateCtxVar) ToService() (*service.CtxVar, error) {
	bs, err := v.DataType.Serialize(v.Value)

	if err != nil {
		return nil, err
	}

	return &service.CtxVar{
		Name: v.VarName,
		Id:   v.VarID,
		Type: service.DataTypeType(v.DataType.GetType()),
		Var: &service.CtxVar_Immediate{
			Immediate: &service.Immediate{
				Value: bs,
			},
		},
	}, nil
}

func ImmediateCtxVarFromService(v CtxVar, v0 *service.CtxVar) (*ImmediateCtxVar,
↪  error) {
	val, err := v.DataType.Deserialize(v0.GetImmediate().Value)
	if err != nil {
		return nil, ErrWrongType
	}
	return &ImmediateCtxVar{
		CtxVar: v,
		Value:  val,
	}, nil
}

type ReferencedCtxVar struct {
	CtxVar
	ReferenceID string
}

func (v *ReferencedCtxVar) GetValue(ctx IContext) (any, error) {
	t, err := ctx.GetReference(v.ReferenceID, v.GetDataType().GetType())
	if err != nil {
		return nil, err
	}
	return t, nil
}

func (v *ReferencedCtxVar) TypeAssert(immediate func(*ImmediateCtxVar), referenced
↪  func(ctxVar *ReferencedCtxVar)) {
	referenced(v)
}

```

```go
func (v *ReferencedCtxVar) ToService() (*service.CtxVar, error) {
	return &service.CtxVar{
		Name: v.VarName,
		Id:   v.VarID,
		Type: service.DataTypeType(v.DataType.GetType()),
		Var: &service.CtxVar_Reference{
			Reference: &service.Reference{
				Id: v.VarID,
			},
		},
	}, nil
}

func ReferencedCtxVarFromService(v CtxVar, v0 *service.CtxVar) (*ReferencedCtxVar,
↪  error) {
	return &ReferencedCtxVar{
		CtxVar:      v,
		ReferenceID: v0.GetReference().GetId(),
	}, nil
}

type ReferenceMap struct {
	dataTypes map[string]DataTypeType
}

type SettingsMap struct {
	dataTypes map[string]DataTypeType
	data      map[string]any
}

func MakeReferenceMap() *ReferenceMap {
	return &ReferenceMap{
		dataTypes: make(map[string]DataTypeType),
	}
}

func MakeSettingsMap() *SettingsMap {
	return &SettingsMap{
		dataTypes: make(map[string]DataTypeType),
		data:      make(map[string]any),
	}
}

func MakeReferenceMapFrom(settingsMap *ReferenceMap) *ReferenceMap {
	m := MakeReferenceMap()
	m.AddAll(settingsMap)
	return m
}

func (m *ReferenceMap) GetDataTypeType(id string) (DataTypeType, error) {
	dtt, ok := m.dataTypes[id]
	if !ok {
		return 0, errors.New("reference type not found")
	}
	return dtt, nil
}

func (m *SettingsMap) GetDataTypeType(id string) (DataTypeType, error) {
	dtt, ok := m.dataTypes[id]
	if !ok {
```

```go
                return 0, errors.New("reference type not found")
        }
        return dtt, nil
}

func (m *SettingsMap) Get(id string, dataTypeType DataTypeType) (any, error) {
        dtt, ok := m.dataTypes[id]
        if !ok {
                return nil, errors.New("reference type not found")
        }
        if dtt != dataTypeType {
                return nil, ErrWrongType
        }
        v, ok := m.data[id]
        if !ok {
                return nil, errors.New("reference not found")
        }
        return v, nil
}

func (m *ReferenceMap) Put(id string, dataTypeType DataTypeType) {
        m.dataTypes[id] = dataTypeType
}

func (m *SettingsMap) Put(id string, dataTypeType DataTypeType, value any) {
        m.dataTypes[id] = dataTypeType
        m.data[id] = value
}

func (m *SettingsMap) Delete(id string) {
        delete(m.dataTypes, id)
        delete(m.data, id)
}

func (m *ReferenceMap) Delete(id string) {
        delete(m.dataTypes, id)
}

func (m *ReferenceMap) AddAll(other *ReferenceMap) {
        for k, v := range other.dataTypes {
                m.dataTypes[k] = v
        }
}

func (m *SettingsMap) AddAll(other *SettingsMap) {
        for k, v := range other.dataTypes {
                m.dataTypes[k] = v
        }
        for k, v := range other.data {
                m.data[k] = v
        }
}

func (m *ReferenceMap) ToService() ([]*service.ReferenceMapEntry, error) {
        refs := make([]*service.ReferenceMapEntry, len(m.dataTypes))
        for k, v := range m.dataTypes {
                refs = append(refs, &service.ReferenceMapEntry{
                        Ref: &service.Reference{
                                Id: k,
                        },
				DataTypeType: v.ToService(),
			})
			return refs, nil
		}
		return refs, nil
}

func (m *SettingsMap) ToService() ([]*service.SettingsMapEntry, error) {
		refs := make([]*service.SettingsMapEntry, len(m.data))
		for k, v := range m.data {
			dtt, ok := m.dataTypes[k]
			if !ok {
				return nil, ErrUnknownDataType
			}
			dt := GetDataType(dtt)
			bs, err := dt.Serialize(v)
			if err != nil {
				return nil, err
			}
			refs = append(refs, &service.SettingsMapEntry{
				Ref: &service.Reference{
					Id: k,
				},
				Value: &service.Immediate{
					Value: bs,
				},
			})
			return refs, nil
		}
		return refs, nil
}

func ReferenceMapFromService(ms []*service.ReferenceMapEntry) (*ReferenceMap, error) {
		m := MakeReferenceMap()
		for _, e := range ms {
			m.Put(e.GetRef().GetId(), DataTypeType(e.GetDataTypeType()))
		}
		return m, nil
}

func SettingsMapFromService(t *Template, ms []*service.SettingsMapEntry)
 ↪  (*SettingsMap, error) {
		m := MakeSettingsMap()
		for _, e := range ms {
			dt := t.GetReferenceDataType(e.GetRef().GetId())
			if dt == nil {
				return nil, ErrUnknownDataType
			}
			v, err := dt.Deserialize(e.GetValue().GetValue())
			if err != nil {
				return nil, err
			}
			m.Put(e.GetRef().GetId(), dt.GetType(), v)
		}
		return m, nil
}

type ICtxVarHolder interface {
		GetCtxVars() []ICtxVar
}
```

A.12 datatype_bool.go

```go
package rules

import (
	"github.com/CAS-ual-TY/honua_local_backend/service"
)

var BoolDataType = &DataTypeBool{
	TypeBasedDataType{
		Type: DataTypeType(service.DataTypeType_BOOL),
	},
}

type DataTypeBool struct {
	TypeBasedDataType
}

func (dt DataTypeBool) Serialize(v any) ([]byte, error) {
	if v, ok := v.(bool); ok {
		if v {
			return []byte{1}, nil
		} else {
			return []byte{0}, nil
		}
	}

	return nil, ErrWrongType
}

func (dt DataTypeBool) Deserialize(b []byte) (any, error) {
	if len(b) != 1 {
		return nil, ErrWrongType
	}

	if b[0] == 0 {
		return false, nil
	} else if b[0] == 1 {
		return true, nil
	}

	return nil, ErrWrongType
}

func (dt DataTypeBool) GetValue(v ICtxVar, ctx IContext) (bool, error) {
	v1, err := v.GetValue(ctx)

	if err != nil {
		return false, err
	}

	if v2, ok := v1.(bool); ok {
		return v2, nil
	} else {
		return false, ErrWrongCtxVarType
	}
}
```

A.13 datatype_cron.go

```go
package rules

import (
	"errors"
	"github.com/CAS-ual-TY/honua_local_backend/service"
	"github.com/CAS-ual-TY/simple_go_cron"
)

var CronDataType = &DataTypeCron{
	TypeBasedDataType{
		Type: DataTypeType(service.DataTypeType_CRON),
	},
}

type DataTypeCron struct {
	TypeBasedDataType
}

func (dt DataTypeCron) Serialize(v any) ([]byte, error) {
	if v, ok := v.(*simple_go_cron.Cron); ok {
		return []byte(v.String()), nil
	}

	return nil, ErrWrongType
}

func (dt DataTypeCron) Deserialize(b []byte) (any, error) {
	cron, err := simple_go_cron.ParseCron(string(b))

	if err != nil {
		return nil, errors.Join(ErrWrongType, err)
	}

	return cron, nil
}

func (dt DataTypeCron) GetValue(v ICtxVar, ctx IContext) (*simple_go_cron.Cron, error)
↪ {
	v1, err := v.GetValue(ctx)

	if err != nil {
		return nil, err
	}

	if v2, ok := v1.(*simple_go_cron.Cron); ok {
		return v2, nil
	} else {
		return nil, ErrWrongCtxVarType
	}
}
```

A.14 datatype_entity.go

```go
package rules

import (
	"github.com/CAS-ual-TY/honua_local_backend/service"
)

var DeviceDataType = &DataTypeDevice{
	TypeBasedDataType{
		Type: DataTypeType(service.DataTypeType_ENTITY_DEVICE),
	},
}

type DataTypeDevice struct {
	TypeBasedDataType
}

func (dt DataTypeDevice) Serialize(v any) ([]byte, error) {
	if v, ok := v.(string); ok {
		return []byte(v), nil
	}

	return nil, ErrWrongType
}

func (dt DataTypeDevice) Deserialize(b []byte) (any, error) {
	return string(b), nil
}

func (dt DataTypeDevice) GetValue(v ICtxVar, ctx IContext) (string, error) {
	v1, err := v.GetValue(ctx)

	if err != nil {
		return "", err
	}

	if v2, ok := v1.(string); ok {
		return v2, nil
	} else {
		return "", ErrWrongCtxVarType
	}
}
```

A.15 datatype_int.go

```go
package rules

import (
	"errors"
	"github.com/CAS-ual-TY/honua_local_backend/service"
	"strconv"
)

var IntDataType = &DataTypeInt{
	TypeBasedDataType{
		Type: DataTypeType(service.DataTypeType_INT),
	},
}

type DataTypeInt struct {
	TypeBasedDataType
}

func (dt DataTypeInt) FromString(s string) (int, error) {
	i, err := strconv.Atoi(s)
	if err != nil {
		return 0, err
	}
	return i, nil
}

func (dt DataTypeInt) ToString(v int) string {
	return strconv.Itoa(v)
}

func (dt DataTypeInt) Serialize(v any) ([]byte, error) {
	if v, ok := v.(int); ok {
		return []byte(dt.ToString(v)), nil
	}

	return nil, ErrWrongType
}

func (dt DataTypeInt) Deserialize(b []byte) (any, error) {
	i, err := dt.FromString(string(b))

	if err != nil {
		return nil, errors.Join(ErrWrongType, err)
	}

	return i, nil
}

func (dt DataTypeInt) GetValue(v ICtxVar, ctx IContext) (int, error) {
	v1, err := v.GetValue(ctx)

	if err != nil {
		return 0, err
	}

	if v2, ok := v1.(int); ok {
		return v2, nil
	} else {
```

```
                return 0, ErrWrongCtxVarType
        }
}
```

A.16 datatype_numeric_sensor.go

```go
package rules

import (
        "errors"
        "github.com/CAS-ual-TY/honua_local_backend/service"
        "strconv"
)

var NumericSensorDataType = &DataTypeNumericSensor{
        DataTypeInt{
                TypeBasedDataType{
                        Type: DataTypeType(service.DataTypeType_NUMERIC_SENSOR),
                },
        },
}

type DataTypeNumericSensor struct {
        DataTypeInt
}

func (dt DataTypeNumericSensor) Serialize(v any) ([]byte, error) {
        if v, ok := v.(int); ok {
                return []byte(dt.ToString(v)), nil
        }

        return nil, ErrWrongType
}

func (dt DataTypeNumericSensor) Deserialize(b []byte) (any, error) {
        i, err := dt.FromString(string(b))

        if err != nil {
                return nil, errors.Join(ErrWrongType, err)
        }

        return i, nil
}

func (dt DataTypeNumericSensor) GetValue(v ICtxVar, ctx IContext) (float64, error) {
        v1, err := v.GetValue(ctx)

        if err != nil {
                return 0, err
        }

        if v2, ok := v1.(int); ok {
                state, err := ctx.GetDB().GetLatestState(v2)
                if err != nil {
                        return 0, err
                }

                current, err := strconv.ParseFloat(state.State, 64)
                if err != nil {
                        return 0, err
                }

                return current, nil
        } else {
```

```
                    return 0, ErrWrongCtxVarType
            }
    }
```

A.17 datatype_sensor.go

```go
package rules

import (
	"errors"
	"github.com/CAS-ual-TY/honua_local_backend/service"
	"strings"
)

var SensorDataType = &DataTypeSensor{
	DataTypeInt{
		TypeBasedDataType{
			Type: DataTypeType(service.DataTypeType_SENSOR),
		},
	},
}

type DataTypeSensor struct {
	DataTypeInt
}

func (dt DataTypeSensor) Serialize(v any) ([]byte, error) {
	if v, ok := v.(int); ok {
		return []byte(dt.ToString(v)), nil
	}

	return nil, ErrWrongType
}

func (dt DataTypeSensor) Deserialize(b []byte) (any, error) {
	i, err := dt.FromString(string(b))

	if err != nil {
		return nil, errors.Join(ErrWrongType, err)
	}

	return i, nil
}

func (dt DataTypeSensor) GetValue(v ICtxVar, ctx IContext) (bool, error) {
	v1, err := v.GetValue(ctx)

	if err != nil {
		return false, err
	}

	if v2, ok := v1.(int); ok {
		state, err := ctx.GetDB().GetLatestState(v2)
		if err != nil {
			return false, err
		}

		current := strings.EqualFold(state.State, "on")

		return current, nil
	} else {
		return false, ErrWrongCtxVarType
	}
}
```

A.18 datatype.go

```go
package rules

import (
	"errors"
	"github.com/CAS-ual-TY/honua_local_backend/service"
)

var ErrWrongType = errors.New("mistype")

type DataTypeType service.DataTypeType

func (t DataTypeType) ToService() service.DataTypeType {
	return service.DataTypeType(t)
}

func GetDataType(dtt DataTypeType) IDataType {
	switch dtt.ToService() {
	case service.DataTypeType_BOOL:
		return BoolDataType
	case service.DataTypeType_INT:
		return IntDataType
	case service.DataTypeType_CRON:
		return CronDataType
	case service.DataTypeType_NUMERIC_SENSOR:
		return NumericSensorDataType
	case service.DataTypeType_SENSOR:
		return SensorDataType
	case service.DataTypeType_ENTITY_DEVICE:
		return DeviceDataType
	}
	return nil
}

type IDataType interface {
	GetType() DataTypeType
	Serialize(v any) ([]byte, error)
	Deserialize(b []byte) (any, error)
}

type IImmediateDataType interface {
	GetValues(ctx IContext) []any
}

type DataType struct{}

type TypeBasedDataType struct {
	DataType
	Type DataTypeType
}

func (t TypeBasedDataType) GetType() DataTypeType {
	return t.Type
}
```

A.19 rule.go

```go
package rules

import (
	"github.com/CAS-ual-TY/honua_local_backend/service"
	"time"
)

type Rule struct {
	RuleName      string
	Condition     ICondition
	Actions       []IAction
	InitialActive bool
}

type RuleInstance struct {
	*Rule
	RuleID int
	active bool
}

func (r *Rule) MakeInstance(RuleID int) *RuleInstance {
	return &RuleInstance{
		Rule:   r,
		RuleID: RuleID,
		active: r.InitialActive,
	}
}

func (r *Rule) ToService() (*service.Rule, error) {
	actions := make([]*service.Action, len(r.Actions), len(r.Actions))
	for i, a := range r.Actions {
		a1, err := a.ToService()
		if err != nil {
			return nil, err
		}
		actions[i] = a1
	}
	c, err := r.Condition.ToService()
	if err != nil {
		return nil, err
	}
	return &service.Rule{
		Condition:     c,
		Actions:       actions,
		InitialActive: r.InitialActive,
	}, nil
}

func RuleFromService(r *service.Rule) (*Rule, error) {
	condition, err := ConditionFromService(r.Condition)
	if err != nil {
		return nil, err
	}
	actions := make([]IAction, len(r.GetActions()), len(r.GetActions()))
	for i, a := range r.GetActions() {
		a1, err := ActionFromService(a)
		if err != nil {
			return nil, err
```

```
                }
                actions[i] = a1
        }
        return &Rule{
                RuleName:      r.GetRuleName(),
                Condition:     condition,
                Actions:       actions,
                InitialActive: r.GetInitialActive(),
        }, nil
}

func (r *RuleInstance) IsActive() bool {
        return r.active
}

func (r *RuleInstance) SetActive(active bool) {
        r.active = active
}

func (r *RuleInstance) Execute(ctx IContext, t time.Time) error {
        if !r.IsActive() {
                return nil
        }

        cond, err := r.Condition.CheckCondition(ctx, t)

        if err != nil {
                return err
        }

        if !cond {
                return nil
        }

        for _, action := range r.Actions {
                err := action.Execute(ctx, t)
                if err != nil {
                        return err
                }
        }

        return nil
}
```

A.20 state.go

```go
package rules

import "errors"

type IState interface {
	GetStringState() string
	GetBoolState() bool
}

type State bool

const (
	ON  = State(true)
	OFF = State(false)
)

func (s State) GetStringState() string {
	if s {
		return "on"
	} else {
		return "off"
	}
}

func (s State) GetBoolState() bool {
	return bool(s)
}

func FromStringState(state string) (State, error) {
	if state == ON.GetStringState() {
		return ON, nil
	} else if state == OFF.GetStringState() {
		return OFF, nil
	}
	return OFF, errors.New("Invalid state")
}
```

A.21 template_manager.go

```go
package rules

import (
	"context"
	"errors"
	"github.com/CAS-ual-TY/honua_local_backend/database"
	"github.com/CAS-ual-TY/honua_local_backend/server"
	"github.com/CAS-ual-TY/honua_local_backend/service"
	"sync"
	"time"
)

var ErrUnknownTemplate = errors.New("unknown template")
var ErrUnknownTemplateInstance = errors.New("unknown template instance")

type TemplateManager struct {
	Templates       []*Template
	ActiveInstances []*TemplateInstance
	lastTime        time.Time

	mt sync.RWMutex
}

func MakeTemplateManager() *TemplateManager {
	return &TemplateManager{
		Templates:       make([]*Template, 0),
		ActiveInstances: make([]*TemplateInstance, 0),
		lastTime:        time.Now().Truncate(time.Second),
		mt:              sync.RWMutex{},
	}
}

func (tpm *TemplateManager) AddTemplate(t *Template) int {
	tpm.mt.Lock()
	defer tpm.mt.Unlock()

	id := len(tpm.Templates)
	tpm.Templates = append(tpm.Templates, t)
	t.TemplateID = uint32(id)
	return id
}

func (tpm *TemplateManager) GetTemplate(id0 uint32) (*Template, error) {
	tpm.mt.RLock()
	defer tpm.mt.RUnlock()

	id := int(id0)
	if id >= len(tpm.Templates) || id < 0 {
		return nil, ErrUnknownTemplate
	}
	t := tpm.Templates[id]
	if t == nil {
		return nil, ErrUnknownTemplate
	}
	return t, nil
}

func (tpm *TemplateManager) DeleteTemplate(id0 uint32) (*Template, error) {
	tpm.mt.Lock()
	defer tpm.mt.Unlock()

	id := int(id0)
	if id >= len(tpm.Templates) || id < 0 {
		return nil, ErrUnknownTemplate
	}
	t := tpm.Templates[id]

	for _, ti := range tpm.ActiveInstances {
		if ti.Template == t {
			return nil, errors.New("template still in use")
		}
	}

	if t == nil {
		return nil, ErrUnknownTemplate
	}
	tpm.Templates[id] = nil
	return t, nil
}

func (tpm *TemplateManager) OverrideTemplate(t *Template) (*Template, error) {
	tpm.mt.RLock()
	defer tpm.mt.RUnlock()

	id := int(t.TemplateID)
	if id >= len(tpm.Templates) || id < 0 {
		return nil, ErrUnknownTemplate
	}
	prev := tpm.Templates[id]
	tpm.Templates[id] = t
	return prev, nil
}

func (tpm *TemplateManager) AddTemplateInstance(t *TemplateInstance) int {
	tpm.mt.Lock()
	defer tpm.mt.Unlock()

	id := len(tpm.Templates)
	tpm.ActiveInstances = append(tpm.ActiveInstances, t)
	t.InstanceID = uint32(id)
	return id
}

func (tpm *TemplateManager) GetTemplateInstance(id0 uint32) (*TemplateInstance, error)
↪  {
	tpm.mt.RLock()
	defer tpm.mt.RUnlock()

	id := int(id0)
	if id >= len(tpm.ActiveInstances) || id < 0 {
		return nil, ErrUnknownTemplateInstance
	}
	t := tpm.ActiveInstances[id]
	if t == nil {
		return nil, ErrUnknownTemplateInstance
	}
	return t, nil
}
```

```go

func (tpm *TemplateManager) DeleteTemplateInstance(id0 uint32) (*TemplateInstance,
↪  error) {
	tpm.mt.Lock()
	defer tpm.mt.Unlock()

	id := int(id0)
	if id >= len(tpm.ActiveInstances) || id < 0 {
		return nil, ErrUnknownTemplateInstance
	}
	t := tpm.ActiveInstances[id]
	if t == nil {
		return nil, ErrUnknownTemplateInstance
	}
	tpm.ActiveInstances[id] = nil
	return t, nil
}

func (tpm *TemplateManager) OverrideTemplateInstance(t *TemplateInstance)
↪  (*TemplateInstance, error) {
	tpm.mt.RLock()
	defer tpm.mt.RUnlock()

	id := int(t.InstanceID)
	if id >= len(tpm.ActiveInstances) || id < 0 {
		return nil, ErrUnknownTemplateInstance
	}
	prev := tpm.ActiveInstances[id]
	tpm.ActiveInstances[id] = t
	return prev, nil
}

func (tpm *TemplateManager) TemplateInstanceFromService(t *service.TemplateInstance)
↪  (*TemplateInstance, error) {
	tpm.mt.RLock()
	defer tpm.mt.RUnlock()

	template, err := tpm.GetTemplate(t.TemplateId)
	if err != nil {
		return nil, err
	}
	settingsMap, err := SettingsMapFromService(template, t.Settings)
	if err != nil {
		return nil, err
	}
	ti := template.MakeInstance(t.TemplateInstanceName, settingsMap)
	return ti, nil
}

func (tpm *TemplateManager) Tick(server *server.HonuaServer) error {
	tpm.mt.Lock()
	defer tpm.mt.Unlock()

	t := time.Now().Truncate(time.Second)

	for ; tpm.lastTime.Before(t); tpm.lastTime.Add(time.Second) {
		for _, ti := range tpm.ActiveInstances {
			if ti.active {
				err := ti.Execute(t, server)
				if err != nil {
```

```go
                                        return err
                                }
                        }
                }
        }

        return nil
}

func (tpm *TemplateManager) GetAllTemplates() []*Template {
        tpm.mt.RLock()
        defer tpm.mt.RUnlock()

        templates := make([]*Template, 0)
        for _, t := range tpm.Templates {
                if t != nil {
                        templates = append(templates, t)
                }
        }
        return templates
}

func (tpm *TemplateManager) GetAllTemplateInstances() []*TemplateInstance {
        tpm.mt.RLock()
        defer tpm.mt.RUnlock()

        tis := make([]*TemplateInstance, 0)
        for _, t := range tpm.ActiveInstances {
                if t != nil {
                        tis = append(tis, t)
                }
        }
        return tis
}

func (tpm *TemplateManager) SaveTemplateInstances(ctx context.Context, db
↪  *database.HonuaDB) error {
        tpm.mt.RLock()
        defer tpm.mt.RUnlock()

        instances := make([]*service.TemplateInstance, len(tpm.ActiveInstances))
        for i, t := range tpm.ActiveInstances {
                t1, err := t.ToService()
                if err != nil {
                        return err
                }
                instances[i] = t1
        }
        return db.SaveTemplateInstances(ctx, instances)
}

func (tpm *TemplateManager) CleanIDs() {
        tpm.mt.Lock()
        defer tpm.mt.Unlock()

        ts := make([]*Template, len(tpm.Templates))
        i := uint32(0)
        for _, t := range tpm.Templates {
                if t != nil {
                        ts[i] = t
```

```go
				t.TemplateID = i
				i++
			}
		}
		tpm.Templates = ts[:i]

		tis := make([]*TemplateInstance, len(tpm.ActiveInstances))
		i = uint32(0)
		for _, t := range tpm.ActiveInstances {
			if t != nil {
				tis[i] = t
				t.InstanceID = i
				i++
			}
		}
		tpm.ActiveInstances = tis[:i]
}

func (tpm *TemplateManager) SaveTemplates(ctx context.Context, db *database.HonuaDB)
↪  error {
		instances := make([]*service.TemplateInstance, len(tpm.ActiveInstances))
		for i, t := range tpm.ActiveInstances {
			t1, err := t.ToService()
			if err != nil {
				return err
			}
			instances[i] = t1
		}
		return db.SaveTemplateInstances(ctx, instances)
}

func (tpm *TemplateManager) SaveTime(ctx context.Context, db *database.HonuaDB) error
↪  {
		return db.SaveTemplatesTime(ctx, tpm.lastTime)
}

func (tpm *TemplateManager) SaveAll(ctx context.Context, db *database.HonuaDB) error {
		err := tpm.SaveTemplates(ctx, db)
		if err != nil {
			return err
		}
		err = tpm.SaveTemplateInstances(ctx, db)
		if err != nil {
			return err
		}
		return tpm.SaveTime(ctx, db)
}

func (tpm *TemplateManager) LoadAll(ctx context.Context, db *database.HonuaDB) error {
		ts0, err := db.LoadTemplates(ctx)
		if err != nil {
			return err
		}
		ts := make([]*Template, len(ts0))
		for i, t0 := range ts0 {
			t, err := TemplateFromService(t0)
			if err != nil {
				return err
			}
			ts[i] = t
```

```
	}

	tis0, err := db.LoadTemplateInstances(ctx)
	if err != nil {
		return err
	}
	tis := make([]*TemplateInstance, len(tis0))
	for i, t0 := range tis0 {
		t, err := tpm.TemplateInstanceFromService(t0)
		if err != nil {
			return err
		}
		tis[i] = t
	}

	tim, err := db.LoadTemplatesTime(ctx)

	tpm.Templates = ts
	tpm.ActiveInstances = tis
	tpm.lastTime = tim

	return nil
}
```

A.22 template.go

```go
package rules

import (
	"errors"
	"github.com/CAS-ual-TY/honua_local_backend/server"
	"github.com/CAS-ual-TY/honua_local_backend/service"
	"time"
)

type Template struct {
	TemplateID   uint32
	TemplateName string
	Rules        []*Rule
	references   *ReferenceMap
}

type TemplateInstance struct {
	Template     *Template
	InstanceID   uint32
	InstanceName string
	Rules        []*RuleInstance
	Settings     *SettingsMap
	active       bool
}

func (t *Template) ToService() (*service.Template, error) {
	rules := make([]*service.Rule, len(t.Rules))
	for i, r := range t.Rules {
		r1, err := r.ToService()
		if err != nil {
			return nil, err
		}
		rules[i] = r1
	}
	return &service.Template{
		TemplateName: t.TemplateName,
		Rules:        rules,
	}, nil
}

func (t *TemplateInstance) ToService() (*service.TemplateInstance, error) {
	settings, err := t.Settings.ToService()
	if err != nil {
		return nil, err
	}
	return &service.TemplateInstance{
		TemplateId:           t.Template.TemplateID,
		TemplateInstanceId:   t.InstanceID,
		TemplateInstanceName: t.InstanceName,
		Settings:             settings,
		Active:               t.active,
	}, nil
}

func TemplateFromService(t *service.Template) (*Template, error) {
	rules := make([]*Rule, len(t.Rules))
	for i, r := range t.Rules {
		r1, err := RuleFromService(r)
```

```go
			if err != nil {
				return nil, err
			}
			rules[i] = r1
		}
		refMap, err := ReferenceMapFromService(t.GetReferences())
		if err != nil {
			return nil, err
		}
		return &Template{
			TemplateID:   t.TemplateId,
			TemplateName: "",
			Rules:        rules,
			references:   refMap,
		}, nil
}

func (t *Template) MakeInstance(instanceName string, settings *SettingsMap)
↪ *TemplateInstance {
		rules := make([]*RuleInstance, len(t.Rules))
		for i, rule := range t.Rules {
			rules[i] = rule.MakeInstance(i)
		}
		return &TemplateInstance{
			Template:     t,
			InstanceName: instanceName,
			Rules:        rules,
			Settings:     settings,
			active:       true,
		}
}

func (t *Template) GetReferenceDataType(id string) IDataType {
		dtt, err := t.references.GetDataTypeType(id)
		if err != nil {
			return nil
		}
		return GetDataType(dtt)
}

func (tp *TemplateInstance) GetRuleInstance(ruleID uint32) (*RuleInstance, error) {
		if ruleID >= uint32(len(tp.Rules)) {
			return nil, errors.New("invalid rule id")
		}
		rule := tp.Rules[ruleID]
		if rule == nil {
			return nil, errors.New("unknown rule id")
		}
		return rule, nil
}

func (tp *TemplateInstance) EnableRule(ruleID uint32) error {
		rule, err := tp.GetRuleInstance(ruleID)
		if err != nil {
			return err
		}
		rule.SetActive(true)
		return nil
}

```

```go
func (tp *TemplateInstance) DisableRule(ruleID uint32) error {
	rule, err := tp.GetRuleInstance(ruleID)
	if err != nil {
		return err
	}
	rule.SetActive(false)
	return nil
}

func (tp *TemplateInstance) GetReference(id string, dataTypeType DataTypeType) (any,
↪  error) {
	return tp.Settings.Get(id, dataTypeType)
}

func (tp *TemplateInstance) Execute(t time.Time, server *server.HonuaServer) error {
	if !tp.IsActive() {
		return nil
	}

	ctx := &Context{template: tp, server: server}

	// prevent side effects. Example:
	// Rule[i] is active,  Rule[i] is inactive
	// Rule[i] activates Rule[i+1]
	// Rule[i+1] is now executed but should not
	activeRules := make([]bool, len(tp.Rules))
	for i, rule := range tp.Rules {
		activeRules[i] = rule.IsActive()
	}

	for i, rule := range tp.Rules {
		if activeRules[i] {
			err := rule.Execute(ctx, t)
			if err != nil {
				return err
			}
		}
	}
	return nil
}

func (tp *TemplateInstance) IsActive() bool {
	return tp.active
}

func (tp *TemplateInstance) SetActive(active bool) {
	tp.active = active
}
```

A.23 util.go

```go
package rules

import (
	"errors"
	"github.com/CAS-ual-TY/honua_local_backend/service"
)

var ErrUnknownType = errors.New("unknown type")

type ComparisonType int

const (
	EQ ComparisonType = iota // Equals
	NQ                       // Not Equals
	GT                       // Greater Than
	GE                       // Greater or Equal
	LT                       // Less Than
	LE                       // Less or Equal
)

func (c ComparisonType) CompareInt(x, y int) bool {
	switch c {
	case EQ:
		return x == y
	case NQ:
		return x != y
	case GT:
		return x > y
	case GE:
		return x >= y
	case LT:
		return x < y
	case LE:
		return x <= y
	}

	return false
}

func (c ComparisonType) CompareFloat(x, y float64) bool {
	switch c {
	case EQ:
		return x == y
	case NQ:
		return x != y
	case GT:
		return x > y
	case GE:
		return x >= y
	case LT:
		return x < y
	case LE:
		return x <= y
	}

	return false
}

```

```go
type Optional[T any] struct {
	Value *T
}

func (o Optional[T]) IsPresent() bool {
	return o.Value != nil
}

func (o Optional[T]) Get() T {
	return *o.Value
}

func (o Optional[T]) OrElse(def T) T {
	if o.Value == nil {
		return def
	}

	return *o.Value
}

func (o Optional[T]) IfPresent(f func(T)) {
	if o.Value != nil {
		f(*o.Value)
	}
}

func (o Optional[T]) IfPresentOrElseThrow(f func(T)) error {
	if o.Value != nil {
		f(*o.Value)
		return nil
	}

	return errors.New("value is not present")
}

type OptionalFloat struct {
	Optional[float64]
}

func (o *OptionalFloat) ToService() *service.OptionalFloat {
	if o.IsPresent() {
		return &service.OptionalFloat{
			Valid: true,
			Value: float32(*o.Value),
		}
	} else {
		return &service.OptionalFloat{
			Valid: false,
			Value: 0,
		}
	}
}

func OptionalFloatFromService(o *service.OptionalFloat) OptionalFloat {
	if o.Valid {
		v := float64(o.Value)
		return OptionalFloat{
			Optional[float64]{
				Value: &v,
			},
                }
        } else {
                return OptionalFloat{
                        Optional[float64]{
                                Value: nil,
                        },
                }
        }
}
```