RESTful services and automation for comfort-oriented smart devices

Master Thesis

David Wettstein

December 2017

Thesis supervisors:

Prof. Dr. Jacques Pasquier-Rocha
and
Arnaud Durand,
Software Engineering Group, Department of Informatics,
University of Fribourg (Switzerland)

Abstract

Internet of Things (IoT) services have a huge potential and there is already a growing market and need for them. However, the IoT is heavily fragmented, lacks interoperability across platforms and uses many different standards [21]. This is where the Web of Things (WoT) comes into play. By using and extending existing, standardized web technologies, it allows an easy integration of services and things with less development costs [21].

Having a smart device attached to a service or the internet is useless without the possibility to control or interact with it. Furthermore, to bring such devices into the WoT, we need to make them accessible via a RESTful web Application Programming Interface (API) [20]. Finally, as the name WoT suggests, we should be able to wire together or integrate multiple things into web applications or services.

In this thesis, we address these remarks by implementing a framework containing a RESTful web API for a real-life physical device collecting comfort-oriented sensors data. Additionally, by integrating an automation tool into the framework, we bring these devices into the WoT.

Keywords: Internet of Things, Web of Things, RESTful API, Smart Gateway, Smart Devices, Automation

Acknowledgements

I want to thank all the people in the Software Engineering Group and in the Human Centered Interaction (Human-IST) group for providing me with ideas and feedback about my project.

Especially, I want to thank Arnaud Durand for working together with me and supervising my project.

Last but not least, I want to thank my wife for all her patience during the project.

Table of Contents

List of Figures

List of Tables

Listings

1 Introduction

1.1. Motivation

In the ever-growing world of IoT and big data, more and more devices for collecting data arise. Often, these devices have only limited resources in terms of computation, memory, bandwidth or power available, and thus can only use light weight protocols for communication. As a consequence, it is not possible to implement a complete RESTful web API as proposed by Roy Fielding in his Ph.D. thesis [1] and there can be the need for a *Smart Gateway* application as proposed and defined in Dominique Guinard's Ph.D. thesis [2].

In this work we implement exactly such a *Smart Gateway* for a smart device, which is sending MQTT[1] messages but has no unified RESTful web API and no data storage.

For such a device, the *Smart Gateway* should provide the following parts [2]:

- Device Drivers: the API communicating with the devices
- Core Services: the REST application framework for the RESTful web API of the Smart Gateway
- Pluggable Services: additional services like a storage service or a search service

Since several technologies already exist for each part of the application, we can use them and combine them such that they fulfill our needs.

[1]MQTT on Wikipedia: `https://en.wikipedia.org/wiki/MQTT`

1.1.1. Goals

Derivated from the motivation, these are the goals of this thesis:

- Choose and deploy a suitable database system to store time-series data.
- Provide an unified RESTful web API for managing the smart devices and for querying data.
- Make use of a process/workflow engine to automate tasks or workflows.

1.2. Organization

- **Chapter 1: Introduction**
  The introduction contains the motivation and goals of this work, a short recapitulation of each chapter along with an overview of the formatting conventions.
- **Chapter 2: Smart Devices and the Web of Things (WoT)**
  This chapter introduces the concept of smart devices and explains what can be understand as the WoT. Furthermore, it explains what is meant by comfort.
- **Chapter 3: A WoT Framework for Comfort-Oriented Smart Devices**
  The third chapter describes the designed framework for achieving the goals defined in chapter 1.1.1. It contains an overview of the framework as well as it explains each component and why they were chosen.
- **Chapter 4: Implementation of the Motivating Example**
  In this chapter, we present the implementation of the motivating example along with possible use cases, which can be accomplished with our framework.
- **Chapter 5: Future Work**
  This chapter discusses how the framework could be improved or extended.
- **Chapter 6: Conclusion**
  Finally, the conclusion describes if the original goals were reached and what we achieved with this thesis.
- **Appendix**
  The appendix contains acronyms, the document license, information about the project source code and references used throughout this work.

1.3. Notations and Conventions

- Formatting conventions:
  - Abbreviations and acronyms are formatted as follows Web of Things (WoT) for the first usage and WoT for any further usage;
  - `https://localhost:3000/explorer` is used for web addresses;
  - Code is formatted as follows:

```
public double division(int _x, int _y) {
    double result = _x / _y;
    return result;
```

```
}
```

List. 1.1: A code example

- – Footnotes use a superscript number: W3C[2]
- – Keywords are formatted in a monospaced font: `Authorization`
- – Inline source code is formatted as: `public static void main(String[] args)`
- – Quotes are formatted as:

  > "Write Once, Run Anywhere"

- – Cites are formatted as: [1]
- – Web references are formatted as: [21]
- – A sinparaenum environment is formatted as: *(i)* the first; *(ii)* the second; *(iii)* the third;

- The work is divided into six chapters that are formatted in sections and subsections. Every section or subsection is organized into paragraphs, signalling logical breaks.
- Figure s, Table s and Listings s are numbered inside a chapter. For example, a reference to Figure j of Chapter i will be noted *Figure* $i.j$:

Fig. 1.1.: Logo of the Software Engineering Group

- As far as gender is concerned, I systematically select the masculine form due to simplicity. Both genders are meant equally.

[2]World Wide Web Consortium (W3C): `https://w3.org`

2

Smart Devices and the Web of Things (WoT)

2.1. Internet of Things (IoT)

According to the *Internet of Things Global Standards Initiative* the Internet of Things (IoT) has been defined as follows:

> "A global infrastructure for the information society, enabling advanced services by interconnecting (physical and virtual) things based on existing and evolving interoperable information and communication technologies." [9]

In their book *Building the Web of Things*, Dominique Guinard and Vlad Trifa described the IoT slightly different:

> "The Internet of Things is a system of physical objects that can be discovered, monitored, controlled, or interacted with by electronic devices that communicate over various networking interfaces and eventually can be connected to the wider internet." [4]

Both definitions have in common that the IoT is about multiple things or objects connected and communicating with each other. However, as mentioned in the abstract, the IoT is heavily fragmented, lacks interoperability across platforms and uses many different standards [21]. As a consequence, the term IoT should be rather treated as a general definition or description.

2.1.1. WoT vs IoT

The WoT is a specialization of the IoT. It is only concerned about the OSI[1] layer 7, the application layer, whereas the IoT usually focuses on the lower layers to reduce computation and power resources [4].

A big advantage of such a high level of abstraction is that it allows us to connect many devices regardless of their actual transport protocols [4]. Furthermore, by using and extending existing, standardized web technologies, the WoT allows an easy integration of services and things with less development costs [21].

D. Guinard and V. Trifa initially proposed an architecture for the WoT including embedding web servers on smart things and applying REST architectural style in two papers [3], [5].

Additionally, they stated the following goal of the WoT in their book:

> "The idea of maximizing existing and emerging tools and techniques used on the web and applying them to the development of Internet of Things scenarios is the ultimate goal of the Web of Things." [4]

2.2. Smart Devices

A smart device or smart thing is a physical object with one or several of sensors (e.g. temperature), actuators (e.g. display) or computing capacities, and that is generally connected by wired or wireless communication interfaces [4].

For using those devices within the WoT, they need to offer a (RESTful) web API, hosted either from the device itself or through a gateway or cloud service [20].

2.2.1. Comfort-Oriented Smart Devices

As comfort-oriented smart devices we classify smart devices containing one or several sensors regarding personal comfort or convenience, e.g. temperature or humidity sensors. Such devices gained a lot of attraction in the context of home automation. As we are only considering smart devices, they are able to send the measured values along with a timestamp (time series data[2]) over some communication interface.

The motivating device for the implementation of our proposed framework is exactly such a comfort-oriented smart device, called *ComfortBox*. It is described in chapter 4.1.

[1] OSI model on Wikipedia: `https://en.wikipedia.org/wiki/OSI_model`
[2] Time series on Wikipedia: `https://en.wikipedia.org/wiki/Time_series`

3

A WoT Framework for Comfort-Oriented Smart Devices

3.1. Description

As written in chapter 1.1 of this thesis, the main goal of the framework proposed in this chapter is to provide a *Smart Gateway* for a comfort-oriented smart device sending time series data.

The framework consists of several components of existing technologies to provide data storage, configuration and automation. However, the main part of the framework is the RESTful web API. This API allows the user to implement new WoT use cases without the need to take care of the basics like data storage. Furthermore, this API represents a unified entry point for the user to the core services of the proposed framework.

In the next sections of this chapter, you can first find an overview of the framework and subsequently a closer look at each component of it.

3.2. Big Picture

The framework includes several services. Each service adds more functionality and has its specific role. The figure 3.1 shows an overview of all components.

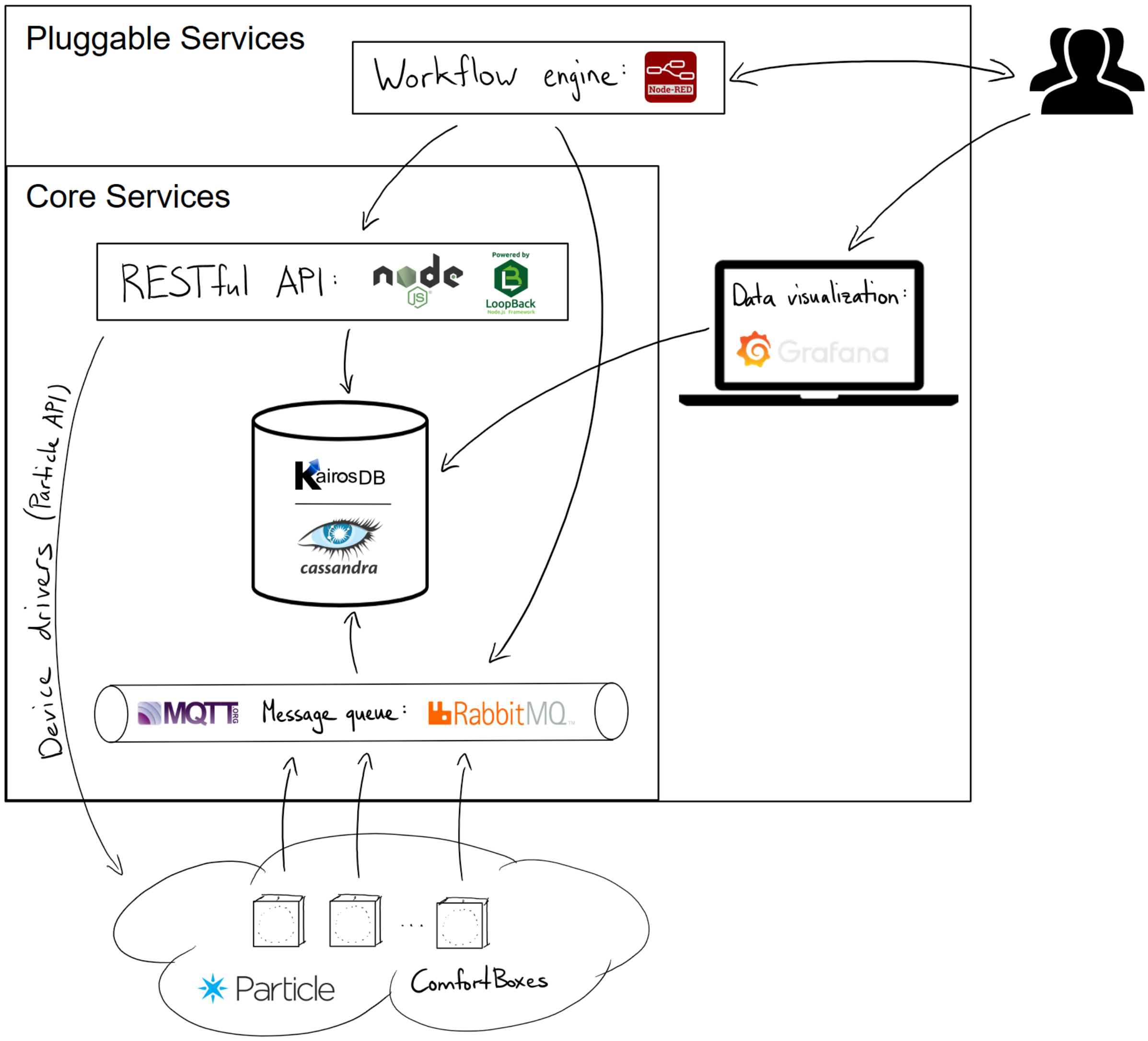

Fig. 3.1.: Big picture of framework

We already know that a *Smart Gateway* should consist of the following parts [2]:

- Device Drivers: the API communicating with the devices
- Core Services: the REST application framework for the RESTful web API of the Smart Gateway
- Pluggable Services: additional services like a storage service or a search service

In figure 3.1, the bottom part can be considered as the *Device Drivers*. Often, the API for communicating with the smart device is proprietary and does not use standard techniques potentially. Since, this part is depending on the used smart devices, it cannot be generalized easily. However, important is that the device API is integrated into the *Smart Gateway*. Additionally, it is an advantage if other devices with different APIs can be added to the *Smart Gateway* later to avoid a vendor lock-in[1].

[1] Vendor lock-in on Wikipedia: `https://en.wikipedia.org/wiki/Vendor_lock-in`

The middle and top part of figure 3.1 are the *Core Services* and *Pluggable Services*. The technologies and applications used in this framework are all open source and have a wide acceptance. However, to avoid a dependance on a specific product, every component could be easily changed later. Furthermore, the framework is designed to be flexible considering the installation of each component. They can be on the same server or also split up on to several servers. Hence, it is possible to distribute the computing load, which can help to get a more reliable system.

As you can see in figure 3.1, the smart devices are not sending their data directly to the database. Although some of the database applications could also handle the data directly, we are using a message broker in between the smart devices and the database. The reason for that is on the one hand to be more flexible and on the other hand to increase possible uses. With a message broker several interested clients could be informed asynchronously about a new message, whereas data in the database would have to be queried by those clients separately.

3.3. Components

In this section we describe each component and what the requirements for the selected application or product were. For doing that we structured every subsection with the following parts:

- Description and existing tools
- Requirements
- Selected product

3.3.1. Message Broker

The message broker component is the entry point for the data coming from the smart devices. Often, the message producers send their messages to a single endpoint or exchange along with a routing key or topic. It is then the responsibility of the message broker system to redirect them to the according queues or to distribute them to the interested clients.

There exist several messaging protocols, including, but not limited to, Advanced Message Queuing Protocol (AMQP), Simple (or Streaming) Text Oriented Message Protocol (STOMP) and Message Queue Telemetry Transport (MQTT).

- **AMQP** - an open application layer internet protocol for business messaging [7]
  - + standardized message format and encoding
  - + offers a wide range of features including queuing and routing
  - − consists of several layers and is thus rather complex
- **MQTT** - an open and light weight publish/subscribe messaging transport protocol [13]
  - + standardized and simple message format, designed so as to be easy to implement

  + small transport overhead with reduced network traffic makes it ideal for constrained environments such as IoT devices
  – doesn't offer queuing, despite the name

- **STOMP** - a simple interoperable, frame based protocol designed for asynchronous messaging [18]
  + standardized text based message format and encoding with so called frames modelled on HTTP
  + builds on simplicity and interoperability
  – doesn't offer a comprehensive messaging API and thus no queues or topics

All of these protocols are based on existing transport layer protocols such as Transmission Control Protocol (TCP). Furthermore, they are supported by some of the most popular message broker applications:

Message Broker	Website	AMQP	MQTT	STOMP
Apache ActiveMQ	`http://activemq.apache.org`	✓	✓	✓
Mosquitto	`https://mosquitto.org`		✓	
RabbitMQ	`https://rabbitmq.com`	✓	✓	✓

Tab. 3.1.: Overview of message brokers

Requirements

Since our framework is supposed to be used with smart devices collecting data, the message broker is an important part of it. The messaging protocol has to be supported by the smart device and it should not consume a lot of resources from it, as we want the sensors to work always properly. For compatibility reasons, it should also be possible to use different protocols within the framework, e.g. on the one hand for the communication between the smart device and the message broker, and on the other hand for the communication between the broker and the database (see also figure 3.1).

Message Format

As the data coming from the smart device is time series data, we need the messages to be in the following JSON format:

```json
{
    "timestamp": "1506317025000",
    "value": "23.910000"
}
```

List. 3.1: Message body

Additionally, the MQTT message topic or AMQP routing key is defined as follows, whereas the forward slash (/) is the topic level separator for MQTT [13] and the dot (.) is the common separator for AMQP version 0.9.1 [6].

```
MQTT: comfort/123456789012345678901234/temp
AMQP: comfort.123456789012345678901234.temp
```

List. 3.2: MQTT message topic or AMQP routing key

The first part defines the type of the device, the second part is the identifier of it and the last part specifies the sensor or event of the device.

Selected Product

For implementing the proposed framework, we used *RabbitMQ* as message broker. It is open source and widely used, also within enterprises. Although *RabbitMQ* is a message broker application for AMQP version 0.9.1 mainly, it supports also MQTT, STOMP and AMQP version 1.0 through additional plug-ins[2]. Furthermore, it has an easy to use management interface accessible from any web browser.

3.3.2. Database

Usually, a database just takes care of storing data. However, since we want to be able to query data over time, we use a database application that is optimized for time series data. These databases are called Time Series Database (TSDB).

As the amount of data can grow big in a relatively short time range, TSDBs offer built-in functionality to aggregate data for queries and to downsample data after a while. It is common to keep high precision data (e.g. at every second) only for a short period of time, whereas older data is automatically downsampled [19].

With a data aggregator it is possible to aggregate high precision data from a large query into summarized values to answer specific questions. As an example, it is possible to query the average temperature over the last month. Another example could also be, the maximum temperature per day over the last week.

Some of the most mature TSDB are:

- **InfluxDB** - `https://influxdata.com`
  - \+ High performance with SQL-like queries
  - \+ Data downsampling and retention
  - \+ Query data with a REST API
  - − Clustering only in enterprise editions
  - − *InfluxData Telegraf* with AMQP consumer plugin as additional components needed
- **KairosDB** - `https://kairosdb.github.io`
  - \+ Several built-in collectors
  - \+ Data retention

[2]Supported protocols by RabbitMQ: `https://www.rabbitmq.com/protocols.html`

  - + Query data with a REST API
  - + Extendable with plugins
  - – No data downsampling
  - – Additional plugin for AMQP needed
- **ThingSpeak** - `https://thingspeak.com`
  - + *MATLAB* for analytics
  - – Outdated source code, some parts seems to be proprietary
  - – Querying data needs expert knowledge
  - – Unknown data lifecyle management

Requirements

As already mentioned, we don't want to only store the data. The chosen TSDB should allow us to query and especially to aggregate data through a web API easily. Another important requirement is the possible collectors. They are used to collect the data (e.g. from a message queue) and to push it into the database. The more collectors and thus protocols like MQTT are supported, the better we can integrate the application. These collectors are used to connect the database application to the message broker from chapter 3.3.1.

Furthermore, having an extendable application would be a plus, but is only an optional requirement.

Selected Product

The open source TSDB *KairosDB* has fulfilled the requirements for our framework the best. Although, it offers a wide range of features, it is simple enough to be easy to use and maintain.

KairosDB has several built-in data collectors and is thus able to communicate through several protocols. However, as you can see in figure 3.1 above, we don't send the data directly to the database, but to the message broker. As a consequence, we need to use a plugin to receive the data from the queues of the message broker and push it into *KairosDB*. We are using an open source plugin originally developed by another person and updated by ourself[3].

A downside of *KairosDB* is that it can only be used either with the in-memory database *H2*[4], which is only useful for development, or with *Apache Cassandra*[5].

Finally, *KairosDB* has well-defined APIs, allows querying data easily and offers several data aggregators.

[3] KairosDB-RabbitMQ plugin: `https://github.com/dwettstein/kairosdb-rabbitmq`
[4] H2 database: `http://h2database.com`
[5] Apache Cassandra: `https://cassandra.apache.org`

3.3.3. API

The API of our framework is the main entry point for user requests and represents an unified interface for querying data from the database and for orchestrating the connected smart devices. Because the smart devices are sending sensitive data, it is mandatory that the API includes user authentication. Besides, the API should be well documented, e.g. using *Swagger*[6] or a similar tool, and it should be able to store metadata about the registered smart devices in a simple and dedicated database.

This is the most work-intensive component of the framework. However, for implementing a RESTful API we don't have to do everything from scratch, as there exist many frameworks in different programming languages which can boost the early development stage.

Requirements

To focus on implementing the important features of the API for our *Smart Gateway*, the chosen framework should include the following requirements out of the box:

- User authentication
- API documentation
- Database connectors
- Extendable and adjustable models
- Compatibility with other technologies

Framework	Webpage	Techn.	Auth.	Doc.	Ext.
ASP.NET	`https://asp.net/web-api`	C#	✓	ϟ	
Django	`http://django-rest-framework.org`	Python	✓	✓	✓
Express	`https://expressjs.com`	Node.js	ϟ	ϟ	✓
Flask	`http://flask.pocoo.org`	Python	ϟ	ϟ	✓
Jersey	`https://jersey.github.io`	Java	✓	ϟ	✓
LoopBack	`https://loopback.io`	Node.js	✓	✓	✓
Sinatra	`http://sinatrarb.com`	Ruby	✓	ϟ	✓
Spring	`https://spring.io`	Java	✓	✓	

Tab. 3.2.: Overview of API frameworks (features with a ϟ are offered by third-party plugins)

Finally, the used framework and technology should have a good and complete documentation about how to use and integrate it.

Selected Product

For the implementation of our API, we selected the technology *Node.js* and the framework *LoopBack*. *Node.js* is an open source, cross-platform runtime for the programming

[6] Swagger: `https://swagger.io`

language *JavaScript*, which is one of the most popular programming language over the last few years according to the analyst company RedMonk [17].

LoopBack builds upon the popular framework *Express* and has additional built-in features like an API explorer for the documentation or several database connectors. Moreover, it can raise the initial development speed with code generators and is able to manage permissions with an Access Control List (ACL) [11].

3.3.4. Workflow Engine

The workflow engine is part of the *Pluggable Services*. Although it is an optional and not necessarily needed component of our framework, it can be used to wire together multiple (web) services or devices and to implement automated workflows.

The user could execute his requests also directly via our API. However, a visual automation tool can simplify the usage for users with less programming knowledge. By using the API within workflows, one can automate its invocations and increase the number of possible use cases for the whole system significantly. Moreover, such engines can often also communicate with other interfaces or services.

Requirements

As we want to enhance our framework, the workflow engine should include the following features:

- A visual editor respectively a Graphical User Interface (GUI) with drag and drop mechanics
- User authentication
- Ability to communicate with any web API
- Extendable with plug-ins or similar
- Ability to consume messages from the message broker
- A user guide with example workflows

Since, all our previous components are open source, we want the workflow engine to be open source as well. Often, there exist a lot of useful work from other developers, if an application is open source.

Selected Product

We decided that the workflow engine *Node-RED* fits our requirements best. It provides a browser-based visual workflow editor and is extendable by implementing so called nodes. When loaded with the editor, a node can be dragged and dropped into your workflows. Although, our API can be called with the built-in nodes directly, we wanted to provide some custom plugin nodes. The idea is again to simplify the usage of our framework and also to increase its capabilities.

Because *Node-RED* builds on *Node.js*, we can use the same programming language for implementing the additional nodes as we used for the API implementation. However, we will not map every API function as a *Node-RED* node, but only the most important ones. Nevertheless, for using custom functions the user doesn't have to implement his own nodes as the editor allows implementing basic scripts written in *JavaScript* directly within the workflow and browser.

4

Implementation of the Motivating Example

4.1. ComfortBox: the Smart Device

The *ComfortBox* is a smart device consisting of several sensors used for collecting data regarding personal indoor comfort or cosiness. It is designed and developed by the Human-IST[1] research group and the supervisor of this thesis, Arnaud Durand, at the University

[1] Human-IST website: `http://human-ist.unifr.ch`

of Fribourg.

Fig. 4.1.: ComfortBox: a 10 x 10 x 10 cm cube

The core component of the *ComfortBox* is a *Particle Photon*[2] (formerly *Spark Core*) microcontroller. As an advantage versus other microcontrollers, the *Particle Photon* offers a built-in Wi-Fi chip. Beside the sensors, the *ComfortBox* has a 2.42 inch monochrome OLED screen for displaying ASCII encoded text and a ring with 24 LED lights for displaying colors around it. Additionally, the *ComfortBox* has an integrated battery.

Although the device has an integrated Wi-Fi, it doesn't have enough resources to run a complete RESTful API on it directly. Since there is no storage in the *ComfortBox* neither, it uses the light weight messaging protocol MQTT for sending the sensor values.

Finally, the *ComfortBox* provides two buttons and an accelerometer for interacting with it.

For any technical details, please have a look at the *GitHub* repository of the *ComfortBox* device (`https://github.com/DurandA/comfortbox`) or contact the developer Arnaud Durand directly.

4.1.1. Sensors

As already mentioned, the *ComfortBox* has several built-in sensors. The measured values of each sensor are regularly sent to the message broker as MQTT messages. As defined in chapter 3.3.1, each message includes a timestamp and a value. The unit of the value depends therefore directly on the sensor and should be documented by the smart device. Additionally, for routing the message to the corresponding queue, we need an abbreviation of the sensor name. This abbreviation is then used for defining the topic or routing

[2] Particle Photon Datasheet: `https://docs.particle.io/datasheets/photon-datasheet`

key.

The *ComfortBox* includes the following sensors:

Sensor	**Abbreviation**	**Unit**
Battery level	bat	%
CO2	co2	ppm
Pressure	hpa	hPa
Humidity	hum	%H
Illuminance	lux	lux
Sound level	sound	dB
Temperature	temp	°C
Wind	wind	km/h

Tab. 4.1.: Overview of ComfortBox sensors

4.1.2. Events

Besides the sensors related to comfort, the *ComfortBox* sends also some messages about events. These events can either origin from interactions or from the connection status.

Event	**Abbreviation**
Press button thumb up	event/button/0
Press button thumb down	event/button/1
Double-tap on device	event/dtap
Tap on device	event/tap
Device is offline	offline
Device is online	online

Tab. 4.2.: Overview of ComfortBox events

4.1.3. Particle Cloud

Although the microcontroller of the *ComfortBox* has an HTTP web API, it is limited in functionality and one has to call an external web service called *Particle Cloud*[3], where all the devices need to be registered. With this service, one can also flash a new firmware for the microcontroller or edit the code within the web Integrated Development Environment (IDE).

For configuring or interacting with a *Particle Photon* device, it is possible to define variables, functions or event handlers within the device firmware. However, there exists some limitations when doing that [15]:

- Variables:

  > "Up to 20 cloud variables may be registered and each variable name is limited to a maximum of 12 characters."

[3]Particle Cloud: `https://build.particle.io`

- Functions:

  "Up to 15 cloud functions may be registered and each function name is limited to a maximum of 12 characters."

- Events:

  "A device can register up to 4 event handlers. This means you can call Particle.subscribe() a maximum of 4 times; after that it will return false."

```
    // A device can register up to 4 event handlers. This means you can call Particle.subscribe() a maximum of 4 times;
    Particle.subscribe(myID+"/led", updateLedsHandler);
    Particle.subscribe(myID+"/display", updateDisplayHandler);
    // Up to 15 cloud functions may be registered and each function name is limited to a maximum of 12 characters.
    Particle.function("set_interval", setInterval);
    Particle.function("set_host", setHost);
    Particle.function("display_data", displayData);
    Particle.function("set_showdata", setShowDataRegularly);
    Particle.function("set_worktime", setWorktime);
```

Fig. 4.2.: ComfortBox: Particle function registrations

In order to send events or call functions of a *Particle Photon* device, you need an access token. Because this token is created and linked with an account on *Particle*, you need to register all devices you want to use within that account.

Fig. 4.3.: Particle device registrations

According to chapter 3.2, we can consider the *Particle Cloud* as the *Device Drivers* for our *Smart Gateway* application.

4.2. Message Broker and Database

As defined in chapter 1.1.1, receiving and storing the data sent by the *ComfortBox* devices is one of the main tasks of our framework.

4.2.1. Message Broker

As decided in chapter 3.3.1 we used *RabbitMQ* as message broker. Since *RabbitMQ* is mainly used with the AMQP protocol, the MQTT adapter has to be enabled first. How to do that and how the plugin works is described within the documentation of the *RabbitMQ* MQTT adapter. However, since MQTT uses slashes (/) for topic segment separators and AMQP 0-9-1 uses dots (.), the plugin has to translate those characters, for example, `comfort/*/temp` becomes `comfort.*.temp` and vice versa. Unfortunately, the consequence is that one cannot use dots in MQTT topics or slashes in AMQP routing keys [16].

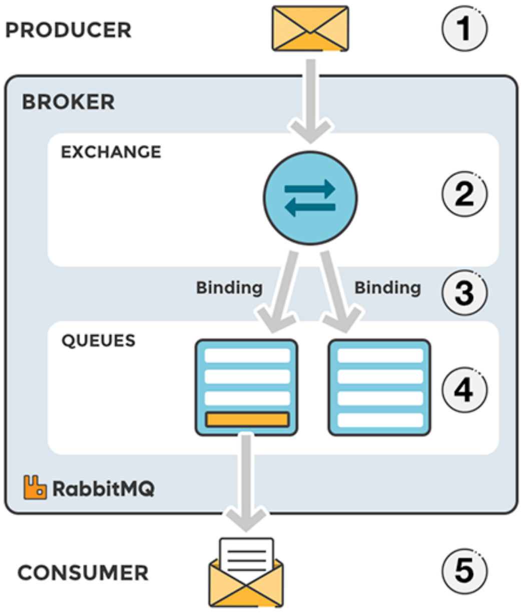

Fig. 4.4.: Message flow in RabbitMQ (figure from CloudAMQP)

We can see in figure 4.4 that a binding is used to route the various messages from the exchange (default `amq.topic`) to the right queue. For each sensor of a *ComfortBox* we create a dedicated queue, since we want to store its data separately. As a guideline, we name the queue equally to the routing keys used for the exchange binding and defined in chapter 3.3.1. This results in the following setup:

Overview			Messages			Message rates		
Name	Features	**State**	**Ready**	**Unacked**	**Total**	**incoming**	**deliver / get**	**ack**
comfort.*.bat	D	idle	0	0	0	0.00/s	0.00/s	0.00/s
comfort.*.online	D	idle	0	0	0	0.00/s	0.00/s	0.00/s
comfort.220037000f47343432313031.bat	D	idle	0	0	0	0.00/s	0.00/s	0.00/s
comfort.220037000f47343432313031.co2	D	idle	0	0	0	0.00/s	0.00/s	0.00/s
comfort.220037000f47343432313031.event.button.0	D	idle	0	0	0			
comfort.220037000f47343432313031.event.button.1	D	idle	0	0	0			
comfort.220037000f47343432313031.event.dtap	D	idle	0	0	0			
comfort.220037000f47343432313031.event.tap	D	idle	0	0	0	0.00/s	0.00/s	0.00/s
comfort.220037000f47343432313031.hpa	D	idle	0	0	0	0.00/s	0.00/s	0.00/s
comfort.220037000f47343432313031.hum	D	idle	0	0	0	0.00/s	0.00/s	0.00/s
comfort.220037000f47343432313031.lux	D	idle	0	0	0	0.00/s	0.00/s	0.00/s
comfort.220037000f47343432313031.offline	D	idle	0	0	0	0.00/s	0.00/s	0.00/s
comfort.220037000f47343432313031.online	D	idle	0	0	0	0.00/s	0.00/s	0.00/s
comfort.220037000f47343432313031.sound	D	idle	0	0	0	0.00/s	0.00/s	0.00/s
comfort.220037000f47343432313031.temp	D	idle	0	0	0	0.00/s	0.00/s	0.00/s
comfort.220037000f47343432313031.wind	D	idle	0	0	0	0.00/s	0.00/s	0.00/s

Fig. 4.5.: RabbitMQ queues

The consequence of having a dedicated queue per sensor is that we will end up with total 14 queues per *ComfortBox* device. However, we don't have to create and bind them manually. Using the special queue `comfort.*.online`, we implemented a process for automated device registration. As this feature is part of our API, it is described in section 4.3.7.

4.2.2. Database

Once the sensors data is ready in the corresponding *RabbitMQ* queues, it has to be consumed and stored into the database. These tasks are done by *KairosDB* with an additional plugin acting as a collector.

In chapter 3.3.2, we mentioned that *KairosDB* can be either used with the *H2* in-memory database or with *Apache Cassandra*. Nevertheless, changing the used datastore can be easily done by editing the `kairosdb.properties` file [10]. Additionally, we had to configure the plugin and set the connection parameters to RabbitMQ.

For subscribing to the relevant queues, the plugin has to know all queue names and the associated binding keys. This information is added to the file `bindings.json`.

```
{
  "bindings": [
    {
      "exchange": "amq.topic",
      "exchangeType": "topic",
      "exchangeDurable": "true",
      "exchangeAutoDelete": "false",
      "exchangeInternal": "false",
      "binds": [
        {
          "bindingkey": "comfort.123456789012345678901234.temp",
          "queueName": "comfort.123456789012345678901234.temp"
        },
        ...
      ]
    }
  ],
  "queues": [
    {
      "queueName": "comfort.123456789012345678901234.temp"
    },
    ...
  ]
}
```

List. 4.1: Snippet of bindings.json file

The binding keys or routing keys respectively are then used to create a data point in *KairosDB*. As we know from chapter 3.3.1, each message body contains a timestamp and a value formatted as JSON. The following table shows how the AMQP messages are mapped to *KairosDB* data points:

Message	=>	**Data Point**
Routing Key	=>	Metric Name
Message Timestamp in Unix time	=>	Data Point Timestamp
Message Value	=>	Data Point Value

Tab. 4.3.: Conversion of messages into data points

For more information about the *KairosDB* plugin, please have a look at the README within the *Git* repository: `https://github.com/dwettstein/kairosdb-rabbitmq`.

Database Queries

Querying and aggregating data were some main requirements described in chapter 3.3.2. With *KairosDB* both tasks can be easily achieved through its REST API.

For a data point query the following API endpoint is used:

```
POST http://[host]:[port]/api/v1/datapoints/query
```

The query parameters are added to the request body formatted as JSON:

```json
{
  "metrics": [
    {
      "tags": {},
      "name": "comfort.123456789012345678901234.temp",
      "aggregators": [
        {
          "name": "avg",
          "align_sampling": true,
          "sampling": {
            "value": "1",
            "unit": "days"
          },
          "align_start_time": false
        }
      ]
    }
  ],
  "cache_time": 0,
  "start_absolute": 1496268000000,
  "end_absolute": 1497045540000
}
```

List. 4.2: Body of a query request

In the `metrics` list, one or multiple metrics can be specified to query. For each metric, only the `name` parameter is mandatory. Optionally, one or multiple aggregators can be defined for a given metric. If multiple aggregators are given, they will be processed in the given order [10]. In the listing above, an average aggregator with a sampling of 1 value per day is used.

Beside the `metrics`, a start and end time for specifying the time range is also needed. Both times can be either absolute, as a Unix timestamp in milliseconds, or relative. The relative time will be calculated by subtracting the given time, e.g. 1 day, from the current date and time [10]. While one of the parameters `start_absolute` or `start_relative` is mandatory, the end time doesn't have to be specified. In this case, the current date and time will be assumed as end time [10].

For all time parameters, including the aggregator sampling, the following units can be used [10]: *(i)* `milliseconds` *(ii)* `seconds` *(iii)* `minutes` *(iv)* `hours` *(v)* `days` *(vi)* `weeks` *(vii)* `months` *(viii)* `years`

To aggregate data, *KairosDB* offers the following aggregators[4]: *(i)* `avg` *(ii)* `count` *(iii)* `dev` *(iv)* `diff` *(v)* `div` *(vi)* `first` *(vii)* `gaps` *(viii)* `last` *(ix)* `least_squares` *(x)* `max` *(xi)* `min` *(xii)* `percentile` *(xiii)* `rate` *(xiv)* `sampler` *(xv)* `save_as` *(xvi)* `scale` *(xvii)* `sum` *(xviii)* `trim`

When installing *KairosDB*, a web GUI is automatically installed for development purposes. Using this GUI, one can get familiar with executing queries as it allows us to set up queries conveniently and looking at the corresponding query JSON. In the following figure, you can see the same query as written above in listing 4.2.

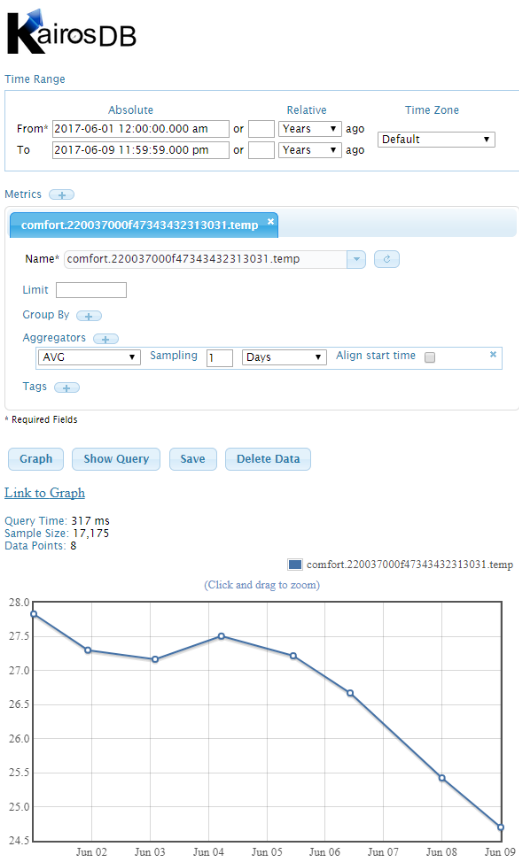

Fig. 4.6.: Query of a KairosDB metric with absolute time range and average aggregator

[4] KairosDB Aggregators Documentation: `https://kairosdb.github.io/docs/build/html/restapi/Aggregators.html`

4.3. RESTful API

4.3.1. API Framework

As decided in chapter 3.3.3 we used *LoopBack* for implementing the RESTful web API of our framework. *LoopBack* not only fulfills the requirements for our API, but also offers almost everything out of the box. Hence, we were able to focus on implementing the actual functionality of our API, which involved the following tasks:

- Add the data sources for the *Particle API*, the *KairosDB API* and the dedicated database for metadata
- Create the *ComfortBox* model
- Implement the needed API functions
- Enable authentication
- Implement automatic device registration by leveraging the message broker

4.3.2. Data sources

> "LoopBack data sources represent backend systems such as databases, external REST APIs, SOAP web services, and storage services." [12]

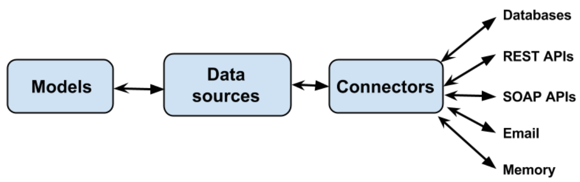

Fig. 4.7.: *LoopBack* data sources (figure from *LoopBack* documentation [12])

As shown in figure 4.7, *LoopBack* uses so called connectors built on the corresponding database driver or client API. The framework provides connectors for several relational and non-relational (NoSQL) databases as well as for backend systems like SOAP or REST APIs and even for push notifications. Furthermore, there exist also plenty of connectors developed by the community and it is also possible to implement your own connector [12].

For implementing our API, we only needed to use data sources for REST APIs and for databases. Adding those data sources can be easily achieved by editing the `datasources.json` file.

Each data source has a property `connector`, which defines the type of the data source entry. In listing 4.3 below the type is `rest`. Other types are, for example, `memory` or `postgresql` as used in section 4.3.2.

```
{
  "ParticleAPI": {
    "name": "ParticleAPI",
    "baseURL": "https://api.particle.io/",
    "crud": false,
    "connector": "rest",
    "options": {
      "headers": {
        "accept": "application/json",
        "content-type": "application/json",
        "Authorization": "Bearer 0000000_your_Particle_token_here_0000000"
      }
    },
    "operations": [
        ...
    ]
  }
}
```

List. 4.3: Example of a REST API data source

A REST data source can contain several operations. Each operation definition consists of the HTTP parameters within the `template` part and the functions of your API (not the external API) within the `functions` part.

```
{
  "template": {
    "method": "POST",
    "url": "https://api.particle.io/v1/devices/events",
    "form": {
      "name": "{particleId}/display",
      "data": "{text}"
    },
    "responsePath": "$"
  },
  "functions": {
    "displayText": ["particleId", "text"]
  }
}
```

List. 4.4: Example of a HTTP POST operation with a body encoded as `application/x-www-form-urlencoded`

When using REST APIs, the URI or request body often contains variables (e.g. ids). In a *LoopBack* data source operation, one can annotate these variables using curly brackets. By adding the variable names to the corresponding function in the `functions` part, the variable values can be set within the source code of the function.

```
curl "https://api.particle.io/v1/devices/events" -d "name={particleId}/display" -d "data={text}"
```

List. 4.5: Analog cURL command of listing 4.4

Database

In section 3.3.2 we already described the database for storing the data sent by the *ComfortBox* devices. Because we want to be able to serve more than one smart device, we need a dedicated database for our API too. In this database, we store metadata about the devices registered within the application. This includes not only the `name` and `particleId` of a *ComfortBox*, but also a timestamp of when the device was registered (property `created`) and optionally some `labels`, which allow us to describe or even group the smart devices.

During the development stage, using an in-memory database is often sufficient. The definition of this data source is also the easiest one, as shown in the following listing 4.6.

```
{
  "db": {
    "name": "db",
    "connector": "memory"
  }
}
```

List. 4.6: Example of a database data source

However, for the production environment an in-memory database is not what you want to have, since a system reboot due to maintenance or power failure would reset the whole database with the consequence of data loss.

In *LoopBack*, a defined data source can be overwritten in a production environment with the file `datasources.production.json`. For overwriting a existing data source, the property `name` needs to be equal. *LoopBack* first initializes all data sources defined in the file `datasources.json` and then overwrites equally named data sources with the definition from the file `datasources.production.json`.

As an example, with the following data source definition we can overwrite the data source with name `db` of type `memory`, which is used for development, with a data source of type `postgresql` for production.

```
{
  "db": {
    "name": "db",
    "host": "localhost",
    "port": 5432,
    "database": "comfortboxapi",
    "user": "comfortboxapi",
    "password": "comfortboxapi",
    "connector": "postgresql"
  }
}
```

List. 4.7: Example of a production database data source

4.3.3. Models

According to the documentation of *LoopBack*, the model definition JSON file declaratively defines a *LoopBack* model. This file (`modelName.json`) is in either the `server` or the `common` project sub-directory, depending on whether the model is server-only or defined for both server and client [12]. In this model file, not only all the options, properties, permissions, methods, etc. are defined, it is also used for generating the API explorer, an interactive documentation to get started with the available API operations. For our API, we had to declare only one JSON file respectively model, the *ComfortBox*.

```
{
  "name": "ComfortBox",
  "plural": "ComfortBoxes",
  "base": "PersistedModel",
  "idInjection": true,
  "options": {
    "validateUpsert": true
  },
  "properties": {
    "name": {
      "type": "string"
    },
    "particleId": {
      "type": "string",
      "required": true
    },
    "created": {
      "type": "date"
    },
    "labels": {
      "type": ["string"]
    }
  },
  "validations": [],
  "relations": {},
  "acls": [
    ...
  ],
  "methods": {
    ...
  }
}
```

List. 4.8: Definition of the *ComfortBox* model in *LoopBack*

As we can see in listing 4.8, a *ComfortBox* object has the following properties:

- **name** - An optional name for the ComfortBox. This name will only be stored within the API database and won't be sent to *Particle Cloud*.
- **particleId** - The id of the device from *Particle Cloud*. This property is mandatory.
- **created** - An optional date and time formatted in ISO 8601 standard: `yyyy-mm-ddThh:mm:ss.sssZ` (e.g. `2017-12-13T14:26:59.993Z`).
- **labels** - An optional list of labels, which can be used to tag or group devices.

Furthermore, each *ComfortBox* device will get a unique `id` while registration. Apart from the properties, the model file also defines the ACL permissions. However, these are kept as simple as possible: all authenticated users are allowed to execute operations, whereas all unauthenticated ones will be denied.

The `methods` list in the model file specifies additional model methods, which are not provided by *LoopBack* itself. These methods are called remote methods [12].

4.3.4. Remote Methods

A remote method can be specified by adding the method name as a key and its options as a value under the methods list in the model definition file.

```
{
  ...
  "methods": {
    "prototype.displayText": {
      "accepts": [
        {
          "arg": "text",
          "type": "string",
          "required": true,
          "description": "Text to display on the ComfortBox"
        }
      ],
      "returns": {
        "arg": "response",
        "type": "string",
        "root": true,
        "description": "Response from Particle API"
      },
      "description": "Display a message on a ComfortBox",
      "http": [
        {
          "verb": "post"
        }
      ]
    },
    ...
  }
}
```

List. 4.9: Example of model remote method specification

By using the `prototype` preposition, one can declare a method as an instance method, which means the method can only be executed on a given instance of the model (e.g. `POST /ComfortBoxes/{id}/displayText`, where `{id}` is the id of the *ComfortBox* instance).

The options object is given as the value of the remote method declaration. It contains mainly the following properties:

- **accepts** - Defines all arguments needed by the remote method.

- **returns** - Defines the return value of the remote method.
- **description** - Describes the remote method. This description is shown in the API explorer.
- **http** - Specifies information about the HTTP route (e.g. verb, path)

Principally, it is not mandatory to specify any options property. However, if the remote method requires arguments, the `accepts` option has to be set. The same applies to `returns` [12].

Finally, the actual code or functionality of a remote method is implemented in the model extension file (`modelName.js`).

```
module.exports = function(ComfortBox) {
  ...
  ComfortBox.prototype.displayText = function(text, callback) {
    var response = 'Called function displayText with param text: ' + text;
    callback(null, response);
  };
  ...
};
```

List. 4.10: Example of a remote method in the model extension file of the *ComfortBox* model

To return a value to the caller at the end of a remote method, *LoopBack* automatically provides the argument `callback`. This argument is actually a reference to the method `callback(error, response)`. While the `error` argument is assumed by *LoopBack*, the remaining arguments correspond to the arguments defined in the `returns` option of the remote method.

4.3.5. API Explorer and Overview of Operations

A usable API needs a good documentation and overview of all available operations and how they have to be requested. *LoopBack* uses the power of *Swagger UI*[5] to automatically generate such a documentation based on the model definition files. This documentation can be viewed in any browser and it is even possible to try out operations immediately.

Besides the built-in basic CRUD operations, our API offers the following operations for interacting with the registered *ComfortBoxes*:

[5] Swagger UI: `https://swagger.io/swagger-ui/`

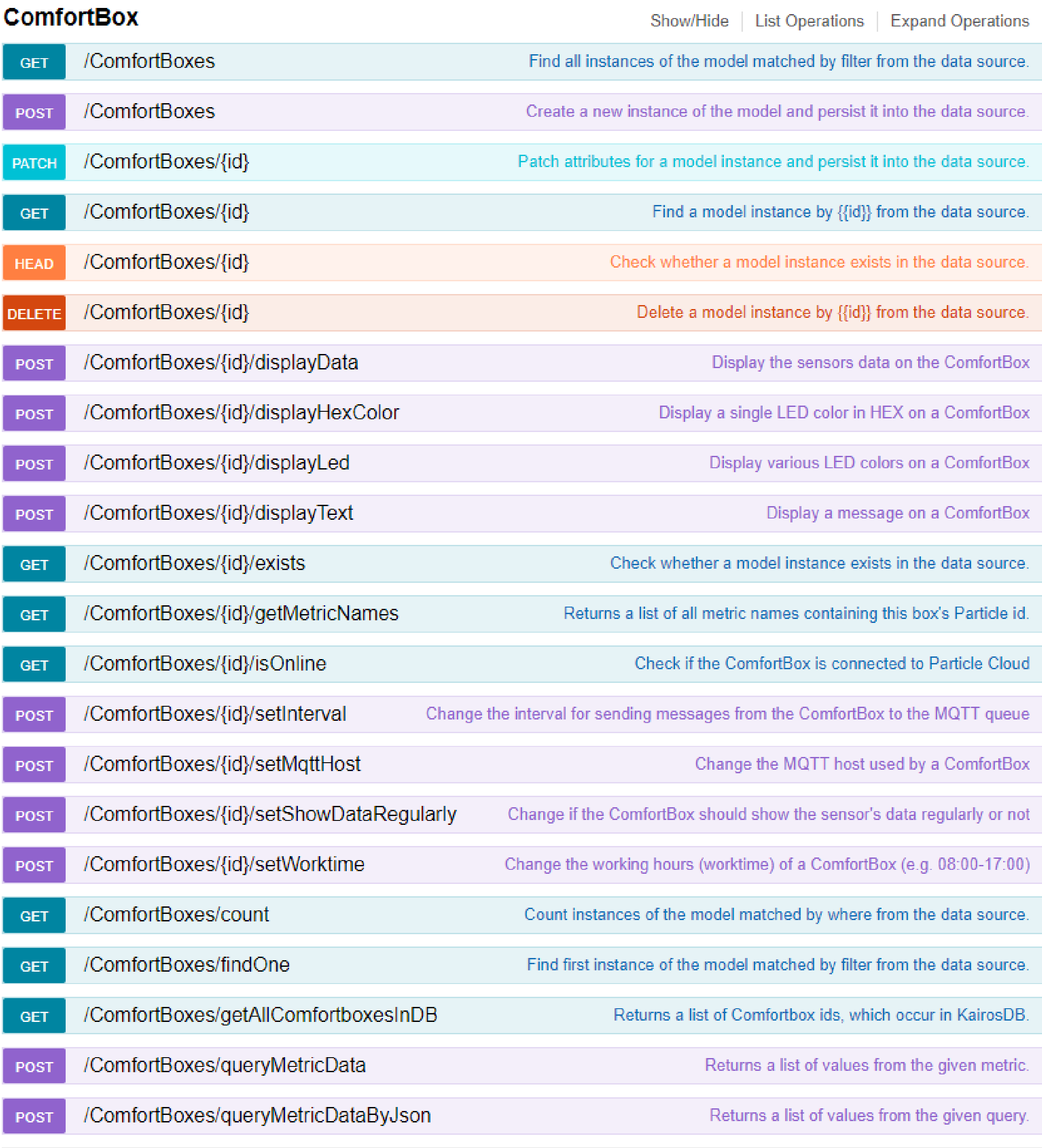

Fig. 4.8.: Screenshot of API explorer showing all *ComfortBox* operations

4.3.6. Authentication

In general, authentication is an important part of every API. Thankfully, *LoopBack* offers a `User` model and ACL mechanisms out of the box. Principally, the authentication system of *LoopBack* is based on a few main concepts [12]:

- **Principal** - An entity (e.g. a user, an application, a role) that can be identified or authenticated.
- **Role** - A group of principals with the same permissions.
- **RoleMapping** - Assignments of principals to roles.

- **ACL** - Access control list: Controls if a principal can perform a certain operation against a model.

By leveraging the full authentication system, we would be able to define very granular permissions. Nevertheless, those models are not published on the API as we want to keep authentication as simple as possible. As a consequence, we don't need the full feature set of the `User` model and we just take advantage of the following `User` operations:

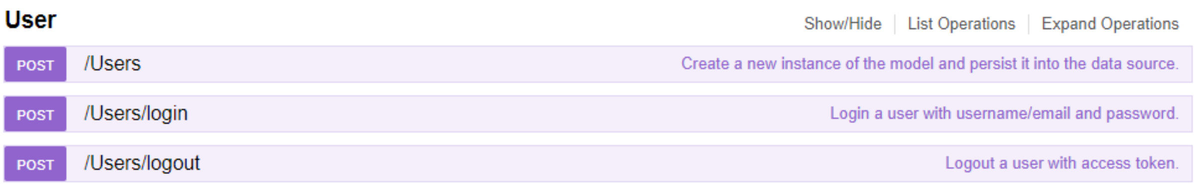

Fig. 4.9.: Screenshot of API explorer showing all *User* operations

Since to create new users one has to be authenticated already, our API uses a default user or master account, which can be configured in the API options. Beside the username and password of this account, you can even set a default access token optionally. Considering that the password and the access token are stored in plain text, it is your responsibility to keep them hidden and save on the underlying operating system.

When logging in with an existing user, one has to use the following format for the request body: `{"username": "defaultUser", "password": "defaultPassword"}`. If the provided credentials were correct, the response contains an access token, which can be used for succeeding requests during a certain time range. The default Time To Live (TTL) for a token is 14 days.

```
{
  "id": "64abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789",
  "ttl": 1209600,
  "created": "2017-12-17T20:45:19.354Z",
  "userId": 1
}
```

List. 4.11: Response of a user login

While authenticated with the initial default user, the request `POST /Users` allows us to create new users, e.g. service accounts. In the end, *LoopBack* would offer full CRUD operations for the `User` and `AccessToken` models, but they are not essential for a straightforward authentication process.

4.3.7. Automatic Device Registration

Our *Smart Gateway* is designed to run even when each component is installed on different servers, as shown in the big picture figure 3.1. If you don't have to separate all components, you can take advantage of the automatic device registration feature when running at least the API service and *KairosDB* on the same server. To enable this feature, the

API service must have access to *RabbitMQ* and to the *KairosDB* bindings file.

When running, the API service listens on a dedicated queue named `comfort.*.online`. Whenever a *ComfortBox* device sends a `online` message, the API service checks in its own database if this device was already registered or not. If the `particleId`, which is the number between `comfort.` and `.online` as defined in section 3.3.1, is unknown, the API service will automatically register or create the device in its dedicated database.

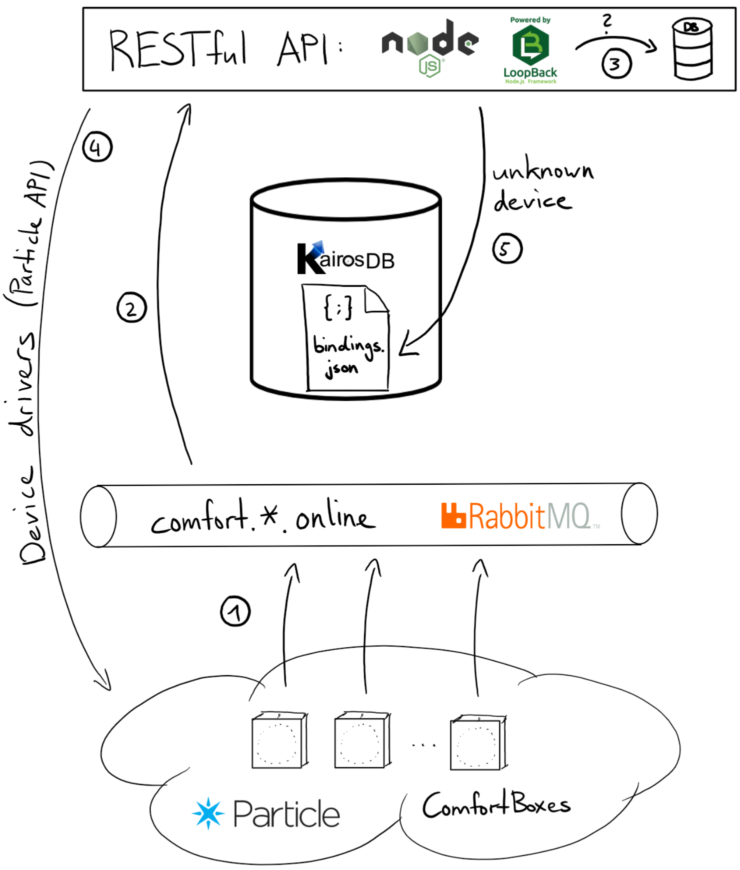

Fig. 4.10.: Automatic device registration process

To create the device instance, the API service will query *Particle* to find out the device name. In cases where the device is not in your own *Particle* account as explained in section 4.1.3 and thus can't be requested nor configured through the *ParticleAPI*, the name `unknown` will be used.

Together with creating the device in the dedicated database, the API service will create the according queues and bindings in the *KairosDB* bindings file (see listing 4.1). This step is needed such that the data coming from the device messages will be stored in *KairosDB* and that we will be able to query this data finally.

4.4. Workflow Automation

4.4.1. Workflow Engine

In chapter 3.3.4 we decided to use *Node-RED* as workflow engine and automation tool. The installation of *Node-RED* is straightforward, as it can be done with *npm*, the default package manager of *Node.js*. After the installation, it can be instantly run with *Node.js* and viewed in your favorite browser.

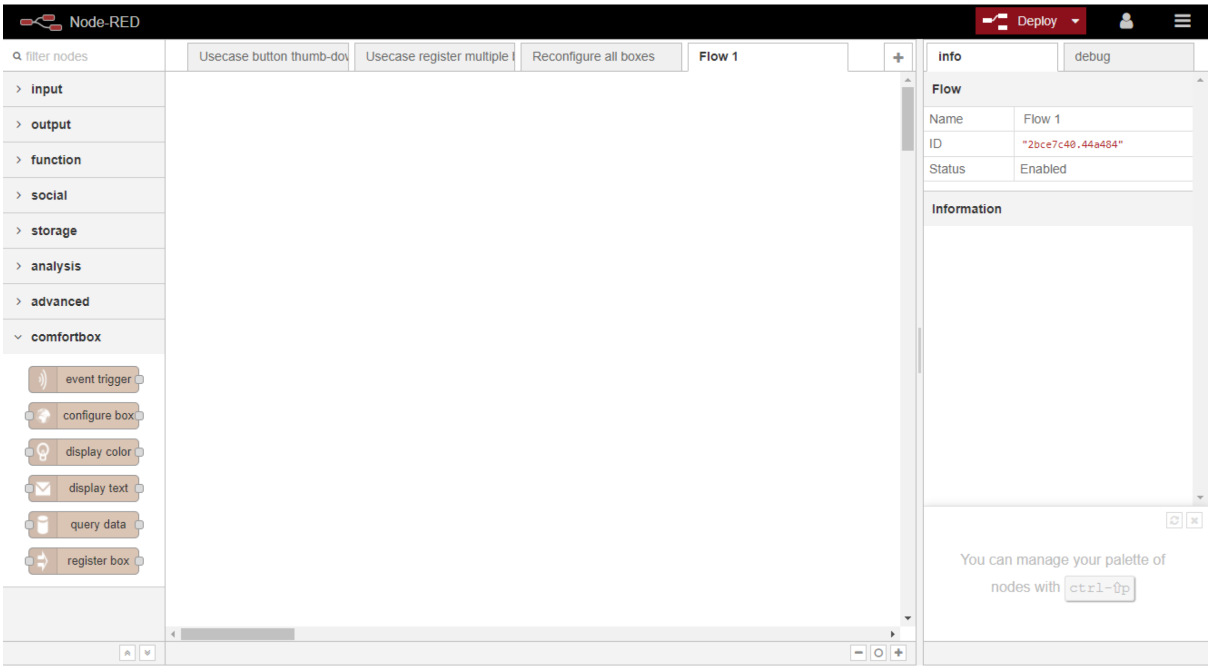

Fig. 4.11.: An empty flow in Node-RED

In figure 4.11 we can see an empty flow in the middle pane, some information on the right pane and all available nodes on the left pane. These nodes can be drag and dropped to the flow and then wired together.

Since our *Smart Gateway* offers an HTTP API, one could already start automating workflows or managing *ComfortBox* devices. Nevertheless, with *Node-RED* it is possible to implement your own customized nodes. Such nodes can simplify using the API for example when querying data, or take over repetitive tasks such as showing a list of all devices registered within our API. For our API, we implemented several *Node-RED* nodes combined in a package[6]. The following section lists and briefly explains those nodes, which can be installed either by cloning the repository into the *Node-RED* user data directory `$HOME/.node-red/nodes` or, if packaged and published, with *npm* [14].

4.4.2. Custom Nodes

Overall we implemented 6 custom nodes and 2 additional configurations, which are used within the other nodes. Each of those nodes is shown below.

[6]Node-RED nodes package: `https://github.com/dwettstein/node-red-contrib-comfortbox`

Configurations

- comfortbox-amqp-server
- comfortbox-api-server

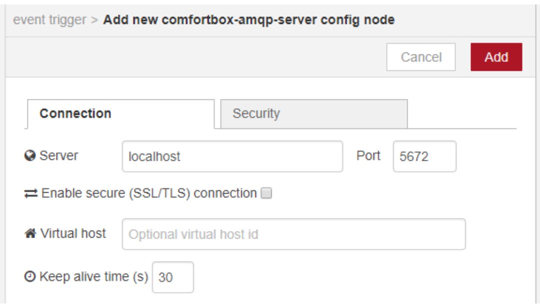

Fig. 4.12.: A configuration node to set an AMQP endpoint for the event trigger

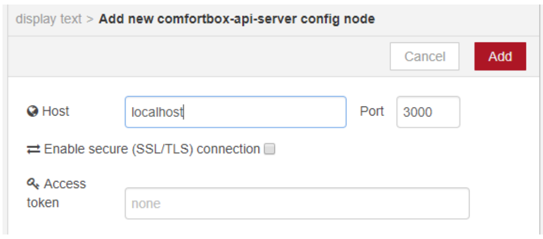

Fig. 4.13.: A configuration node to set the API services endpoint

When there is a successfully configured API server, all nodes will query and list the registered *ComfortBox* devices when editing them. Those devices can then be selected and their `boxId` and `particleId` will be filled automatically.

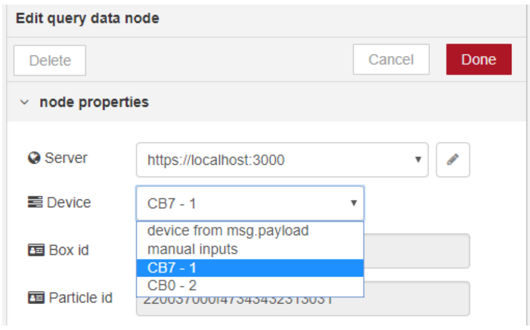

Fig. 4.14.: Selecting a registered device from the API server `https://localhost:3000`

Nodes

- configure box
- display color
- display text
- event trigger
- query data
- register box

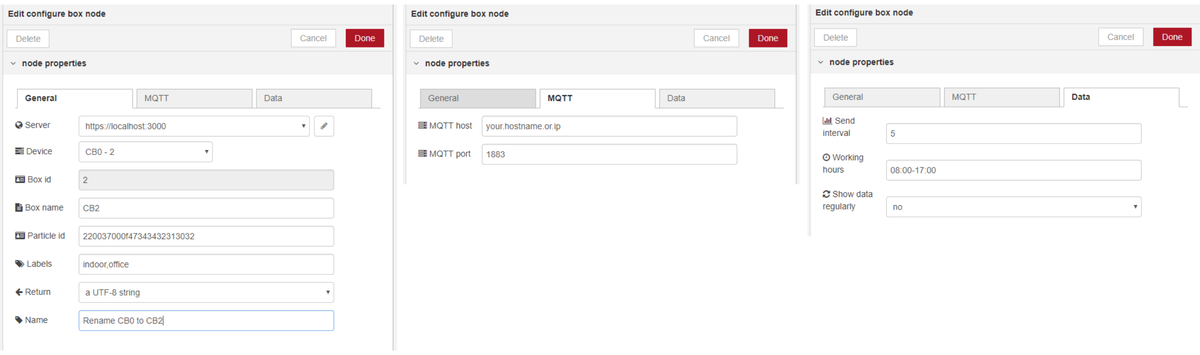

Fig. 4.15.: A node to configure a registered ComfortBox device (e.g. set the MQTT host)

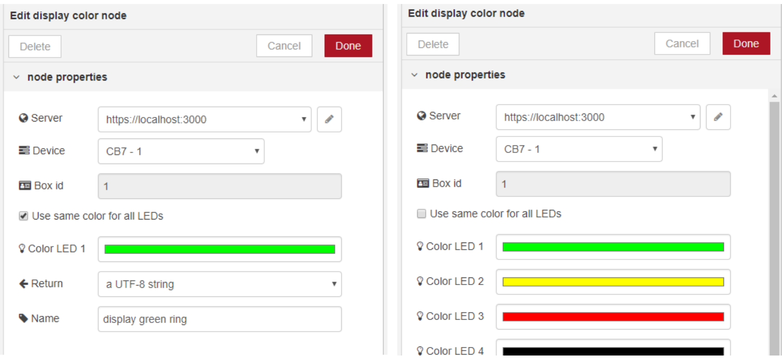

Fig. 4.16.: A node to display one or multiple colors on a ComfortBox device

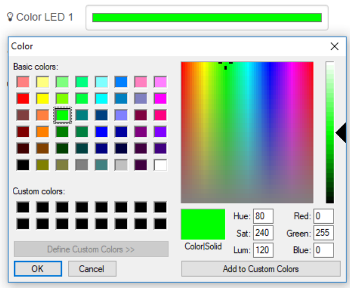

Fig. 4.17.: Color selector

As you can see in figure 4.16, you can either choose to use the same color for all 24 LEDs or to select each color individually. To simplify the selection of a color, a color selector will pop up (figure 4.17) when clicking on a color field. If you select the color black, it won't be seen on the box as its background is black.

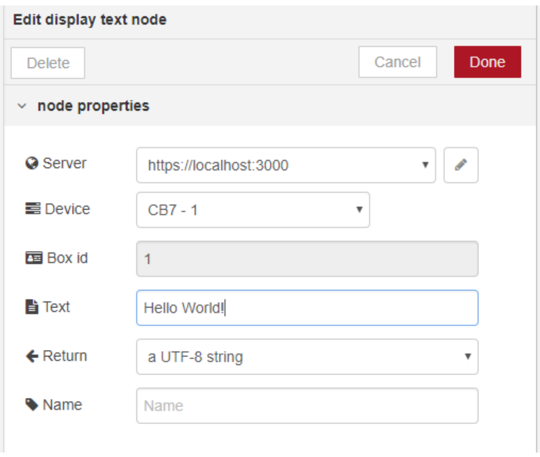

Fig. 4.18.: A node to display ASCII text on a ComfortBox device

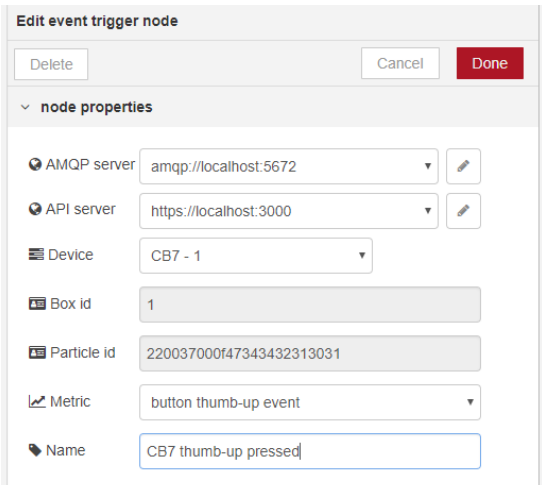

Fig. 4.19.: A node to trigger a flow from an AMQP event

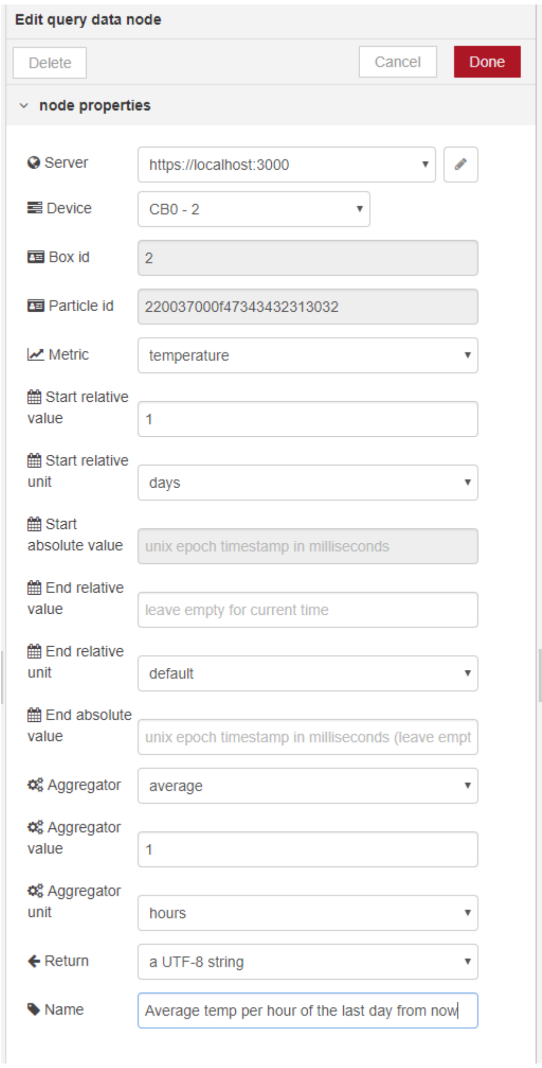

Fig. 4.20.: A node to query data of a ComfortBox device

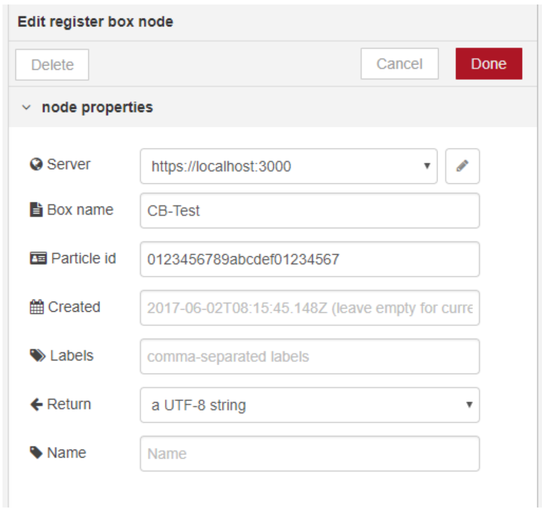

Fig. 4.21.: A node to register a new ComfortBox device within the API services

4.4.3. Use Cases

To demonstrate the possibilities of using a workflow engine, we prepared three demo use cases. The following subsections show and explain each of those use cases.

Use Case 1: Register multiple devices

When using comfort-oriented smart devices, such as the *ComfortBox*, you have usually more than one device. As a consequence, registering every device within our API service one by one would take a lot of time. With a workflow you can script this process such that you just have to provide a list of device ids, which have to be registered.

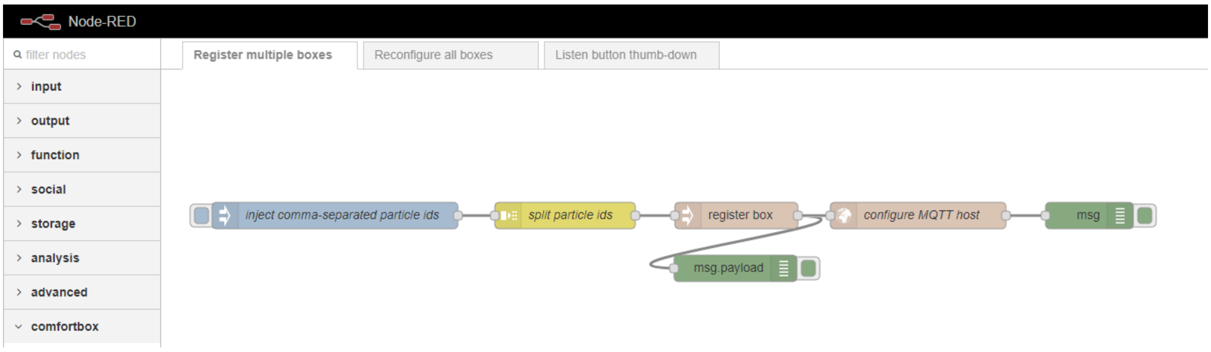

Fig. 4.22.: Workflow to register multiple devices

Use Case 2: Reconfigure all devices

As we know that the devices are sending their data to an external message broker, it can happen that the connection to this broker change and you thus need to reconfigure all registered devices. This can be achieved with a workflow as well.

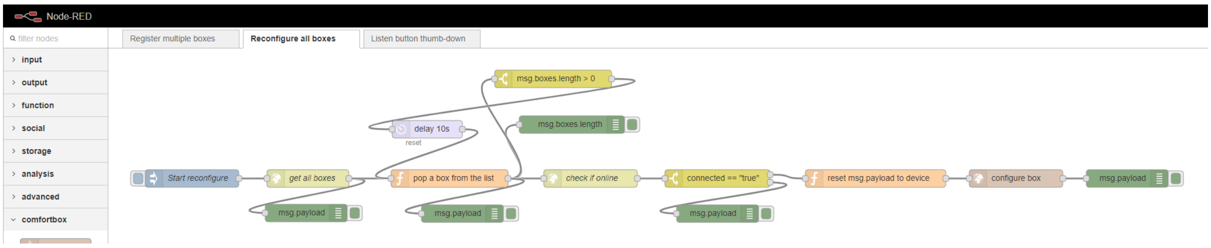

Fig. 4.23.: Workflow to reconfigure all registered devices

Use Case 3: Listen to button thumb-down

The third use case shows a more advanced use case of the workflow engine. By configuring the message broker within the workflow engine, it is possible to directly listen on certain events sent by a device. In our example, we listen to thumb-down button events.

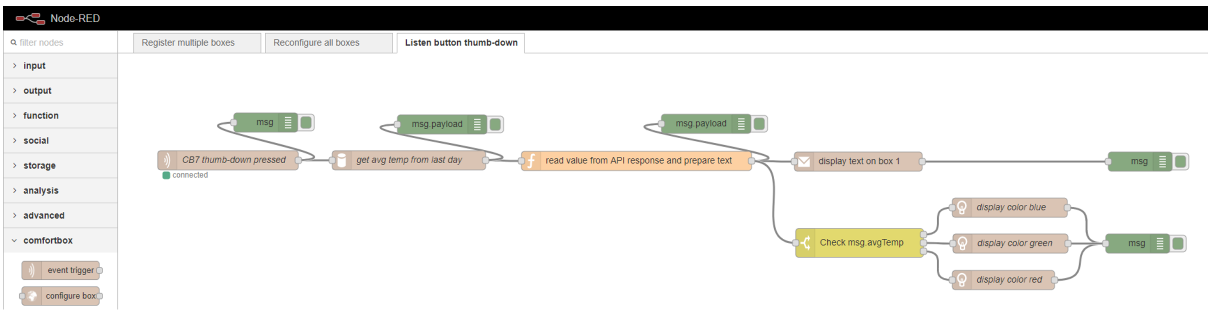

Fig. 4.24.: Workflow to listen to a specific event

Every time the button is pressed and thus sends an event, we execute a query for getting the average temperature during the last 24 hours. This value is then sent back to the device along with some text. Additionally, we display a certain color depending on the value.

This use case shows how to assign interaction possibilities to a device. Furthermore, it demonstrates how to react and to execute subsequent actions when receiving a certain event.

4.5. Data Visualization

As a supplementary component, we installed *Grafana*[7] on the same server. Since we are not focusing on data visualization in this thesis, we choosed this software just because of personal preferences and compatibility reasons. A possible alternative would be *freeboard*[8]. Nevertheless, *Grafana* is very popular and the leading open source software for time series analytics, according to their website. Furthermore, *Grafana* offers a data source plugin for *KairosDB*, which we are using as our database.

Fig. 4.25.: An example dashboard in *Grafana* for a *ComfortBox* device

[7]Grafana: `https://grafana.com/`
[8]freeboard: `https://freeboard.io/`

5 Future Work

5.1. Limited to *ComfortBox* devices

Currently, our implemented framework only works with *ComfortBox* devices. With the growing market of smart devices, including thermostats or hygrometers, it would be useful to extend the framework such that various devices could be connected and used. A popular device would be the *Raspberry Pi* and its *Sense HAT*[1], which offers similar sensors compared to the *ComfortBox*.

5.2. Proper runtime handling and monitoring

By now, our API services and *Node-RED* instance have to be run with the `nohup`[2] command to ignore the terminal logout signal. This is a quick and dirty solution and should be improved, e.g. with a `systemd` service unit.

Furthermore, the different components of our framework should be monitored properly, but this is only possible by having a decent runtime handling first.

5.3. Automated framework installation

Installing all components of our framework takes some time and requires some operating system knowledge currently. By putting everything into a container application, e.g.

[1] Sense HAT for Raspberry Pi: `https://www.raspberrypi.org/products/sense-hat/`
[2] nohup on Wikipedia: `https://en.wikipedia.org/wiki/Nohup`

like *Docker*[3], or a virtual machine appliance, e.g. using the Open Virtualization Format (OVF)[4] standard, the installation could be reasonably simplified and automated.

5.4. Other ideas

Beside the above main points, the following ideas could also help to improve our work:

- Extend user model for more granular access roles

[3] Docker: `https://www.docker.com/`
[4] OVF standard: `http://www.dmtf.org/standards/ovf`

6 Conclusion

IoT or smart devices are getting more and more popular. Nevertheless, these devices are often too constrained in terms of computing capacities and power to run a whole RESTful web API on themselves. This is where a *Smart Gateway* application, as proposed by Dominique Guinard [2], comes into play.

In this thesis we implemented such a *Smart Gateway* application for a real-life physical device collecting comfort-oriented sensors data. We defined not only the data storage, but implemented primarily an unified RESTful web API including authorization for managing and communicating with the devices, and querying data from the data storage.

Finally, the orchestration and maintenance of those devices can be automated by using a workflow engine or similar tool, just as we did in our implemented framework. This tool can further be used to create mashups of several existing services or devices.

With our framework, we showed that we can bring physical devices into the WoT and extend their capabilities or possible uses. Additionally, we proved that this can be achieved with existing and open source components.

A
Common Acronyms

ACL	Access Control List
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
CRUD	Create, Read, Update, Delete
DBMS	Database-Management System
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IDE	Integrated Development Environment
IoT	Internet of Things
JSON	JavaScript Object Notation
LED	Light-Emitting Diode
MOM	Message-Oriented Middleware
MQTT	Message Queue Telemetry Transport
OLED	Organic Light-Emitting Diode
OSI	Open Systems Interconnection
OVF	Open Virtualization Format
RFID	Radio Frequency Identification
REST	Representational State Transfer
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
STOMP	Simple (or Streaming) Text Oriented Message Protocol
TCP	Transmission Control Protocol
TSDB	Time Series Database
TTL	Time To Live
URI	Unified Resource Identifier
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
WoT	Web of Things
XML	eXtensible Markup Language

B
License of the Documentation

A full copy of the license text can be read from [8].

C
Project Repositories

The following source code repositories are released under the MIT license. You can find the according license files in the root folder of each repository within a file named `LICENSE`.

C.1. Framework

The code repository for the framework can be found on the following website:
`https://github.com/dwettstein/comfortbox-api-services`

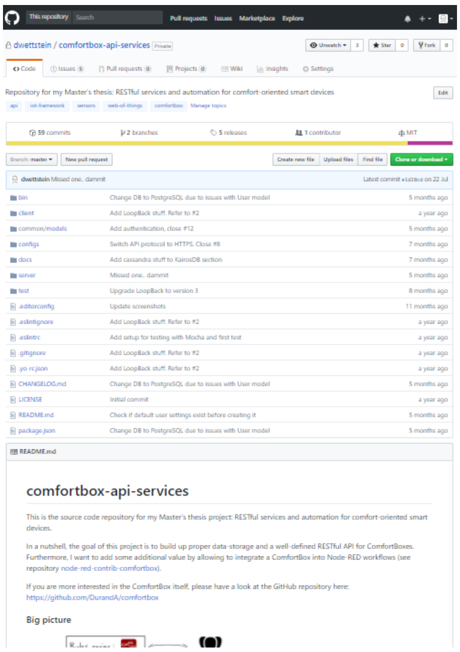

Fig. C.1.: Screenshot of the framework repository

On this site you will find:

- The source code of the API service
- The documentation for preparing, installing and configuring the API
- The documentation for installing each framework component

- Example configurations for each framework component
- A *Postman*[1] collection with the *ParticleAPI* operations of a *ComfortBox*

C.2. Node-RED plugin nodes

The code repository for the *Node-RED* plugin nodes can be found on the following website:
`https://github.com/dwettstein/node-red-contrib-comfortbox`

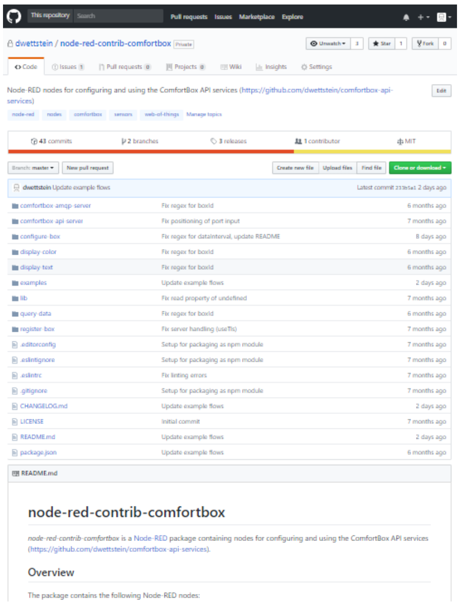

Fig. C.2.: Screenshot of the *Node-RED* plugin repository

On this site you will find:

- The source code of the *Node-RED* plugin nodes
- The documentation for the plugin
- The example flows, which can be imported into any *Node-RED* instance

[1] Postman: `https://www.getpostman.com/`

References

[1] R. Fielding. *Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, University of California, Irvine, United States, 2000.

[2] D. Guinard. *A Web of Things Application Architecture - Integrating the Real-World into the Web.* PhD thesis, ETH Zürich, Switzerland, 2011.

[3] D. Guinard and V. Trifa. Towards the web of things: Web mashups for embedded devices. Technical report, ETH Zürich, Switzerland, 2009.

[4] D. Guinard and V. Trifa. *Building the Web of Things.* Manning Publications Co., 20 Baldwin Road, PO Box 761, Shelter Island NY 11964, United States, 2016.

[5] D. Guinard, V. Trifa, and E. Wilde. Architecting a mashable open world wide web of things. Technical report, ETH Zürich, Switzerland, 2010.

Referenced Web Resources

[6] Advanced message queuing protocol (amqp) specification, version 0.9.1. https://www.rabbitmq.com/resources/specs/amqp0-9-1.pdf (accessed October 27, 2017).

[7] Oasis advanced message queuing protocol (amqp) specification, version 1.0. http://docs.oasis-open.org/amqp/core/v1.0/os/amqp-core-overview-v1.0-os.html (accessed October 13, 2017).

[8] Free documentation licence (gnu fdl). http://www.gnu.org/licenses/fdl.txt (accessed July 01, 2017).

[9] Internet of things global standards initiative. https://www.itu.int/en/ITU-T/gsi/iot/Pages/default.aspx (accessed December 31, 2017).

[10] Kairosdb documentation. https://kairosdb.github.io/docs/build/html/GettingStarted.html (accessed December 06, 2017).

[11] Loopback compare. http://loopback.io/resources/#compare (accessed October 16, 2017).

[12] Loopback documentation. http://loopback.io/doc/en/lb3/index.html (accessed November 02, 2017).

[13] Oasis mqtt specification, version 3.1.1. http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html (accessed October 13, 2017).

[14] Node-red documentation. https://nodered.org/docs/ (accessed December 20, 2017).

[15] Particle cloud reference. https://docs.particle.io/reference/ (accessed October 28, 2017).

[16] Rabbitmq mqtt adapter documentation. https://www.rabbitmq.com/mqtt.html (accessed December 03, 2017).

[17] Redmonk programming languages. http://redmonk.com/sogrady/category/programming-languages/ (accessed October 16, 2017).

[18] Stomp protocol specification, version 1.2. https://stomp.github.io/stomp-specification-1.2.html (accessed October 13, 2017).

[19] Time series database (tsdb) explained. https://www.influxdata.com/time-series-database/ (accessed December 31, 2017).

[20] Web thing model. https://www.w3.org/Submission/wot-model/#dfn-web-thing (accessed July 26, 2017).

[21] W3c white paper for the web of things. https://w3c.github.io/wot/charters/wot-white-paper-2016.html (accessed July 26, 2017).

Index