Universal Explorer for the Web of Things

Master Thesis

Linus Schwab

Faculty of Science
University of Bern

3. September 2018

Prof. Dr. Jacques Pasquier
Arnaud Durand

Software Engineering Group
Department of Informatics
University of Fribourg

Acknowledgements

First I would like to thank *Prof. Dr. Jacques Pasquier* for giving me the opportunity to write my master thesis at the Software Engineering Group of the University of Fribourg. I would also like to thank *Arnaud Durand* for his ongoing support and advice during this project. Additionally, I would like to thank *Andreas Hohler* and *Pascal Giehl* for their support and understanding throughout the project. This made it possible for me to write my thesis while working almost full-time at our company devedis. Finally, thanks a lot to *Lucie Stöcklin*, who took the time to proof-read this thesis and provide me with her feedback.

Abstract

The W3C Web of Things Interest Group collects concepts and technologies to enable discovery and interoperability of Internet of Things devices and services on a worldwide basis. To counter the current IoT fragmentation, the Web of Things uses standard complementing building blocks. Among others, the WoT building blocks include the WoT Thing Description that contains the semantic metadata of a Thing as well as a functional description of its WoT Interface and the WoT Protocol Binding Templates that include mappings to support multiple protocols used in the IoT.
Besides the W3C, Mozilla is also taking part in the ongoing standardization efforts around the Web of Things. Those include a simple Web Thing Description format based on JSON and a Things Framework that provides a collection of re-usable software components for building Web Things.
This thesis introduces the Universal Explorer for the Web of Things, a web-based application that acts as a gateway and supports both the W3C and the Mozilla Thing Description formats to communicate with things. The Universal Explorer implements a simple Thing Description parser and encoder and includes a JavaScript API, a RESTful HTTP API with OpenAPI documentation and a WebSocket API for real-time notifications.

Keywords: Web of Things, REST, WebSockets, TypeScript, Node.js

Table of Contents

List of Figures

List of Tables

Listings

1 Introduction

For a few years now smart devices that are connected to the internet and communicate with each other are no longer a fantasy only existing in science finction movies. Nowadays there are various devices with sensors and actuators available from different manufacturers that can measure and interact with their surroundings and can be used in home automation, smart cities and many other fields. These gadgets include many items such as smart light bulbs, temperature or air quality sensors, wireless buttons or smart power switches.

With connected devices like these it is effortlessly possible to control all the lights in a house with an app, presence sensors or smart switches or to receive a notification on your phone if for example the CO_2 concentration in a room reaches a certain treshold - as long as all interacting devices are from the same manufacturer. Connecting devices from different manufacturers to interact together in automation scenarios or to use them in other applications is difficult and takes a lot of time and resources.

1.1. The Problem

Nowadays manufacturers often use different, proprietary communication protocols and platforms which are not interoperable. They form multiple small islands at the application layer [4]. This fragmentation makes it difficult to integrate devices from different platforms and use them together in one large system [1][5].

A good example for this is the current situation with home automation devices. A user study conducted by Brush *et al.* [18] with owners of home automation systems identified problems that are barriers for a broad adoption of such devices. Besides the high cost of

ownership in either money or time (and sometimes both), proprietary systems are inflexible. As different platforms are not compatible, the user must choose between either ease of integration by only using systems used by one manufacturer or flexibility. However, choosing multiple systems by different manufacturers results in a high effort to integrate the devices. Another problem identified in the study was the poor manageability, as it is necessary to use the proprietary software provided by the vendor of the devices for automation. The rule-based automation each manufacturer has to develop separately did not work reliably and resulted in unexpected behaviour. Aditionally, most user interfaces of the proprietary applications were complex and unintuitive.

Due to these proprietary systems, smart things remain hard to integrate into composite applications [4]. Without an unified protocol to interconnect the various devices, a large amount of time, expert knowledge and resources are required to build applications that communicate with different things due to the overly complicated process [5][8]. This time and resources would be much better spent in developing the actual application instead of wasting it on the integration of proprietary protocols and systems. Of course, there are currently proprietary Internet of Things plattforms available like IFTTT[1] that make it easy even for regular consumers to integrate devices from different manufacturers and use them together [6]. However, time and resources are still required to integrate the devices by the manufacturers, if they even choose to do so.

To solve these issues, an open standard is necessary to connect the many already existing Internet of Things plattforms. The solution is to enable worldwide discovery and interoperability by exposing things trough the Web of Things [3]. The web is an outstanding candidate for a universal integration platform due to the ubiquitous availability on almost any device and the development of composite applications [4]. By making smart things an integral part of the web, they become easier to build upon. This allows the use of existing popular web technologies such as HTML, JavaScript, PHP or Node.js to build applications involving smart things [2][7].

1.2. Motivation and Goals

At the time of writing, the W3C Web of Things Interest Group[2] is working on the architecture for the Web of Things, intended to enable interoperability across IoT Platforms. Its main goal is to allow Internet of Things devices to communicate with each other, independent of their underlying implementation and across multiple networking protocols, in a standardized way [19].

To achieve this, the W3C defines building blocks that build the architecture for the Web of Things. Those include a Thing abstraction model that represents a physical device or a virtual entity to be used in Internet of Things applications, a Thing Description that provides stuctured metadata about a Thing as the primary building block, protocol Binding Templates to support the multiple Internet of Things protocols and a Scripting

[1] https://ifttt.com
[2] https://www.w3.org/2016/12/wot-wg-2016.html

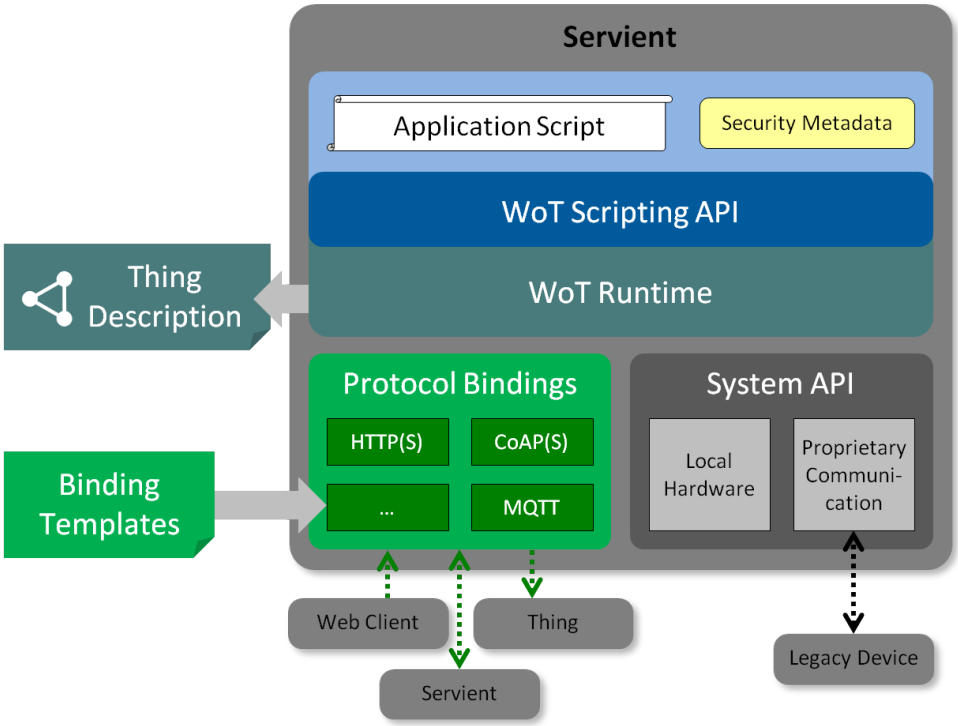

Fig. 1.1.: Architecture of a Web of Things Servient

API to ease Internet of Things application development.

These building blocks are planned to be implemented in Servients. A Servient is a software stack that can perform either as the server or as client by hosting and exposing Things and/or consuming Things. As visible in Figure 1.1, the architecture of a Servient includes Protocol Bindings to communicate with other Servients, Things or Web Clients to build the Web of Things. Trough the System API, Servients can even implement proprietary communication protocols to include legacy devices in the Web of Things. Another possible usage scenario is to implement a Servient as part of a new smart device and then directly accessing the local hardware to connect the device to the Web of Things. On top of that, the WoT Runtime generates the Thing Description providing the metadata and the Scripting API.

Besides the W3C, Mozilla[3] is also taking part in the ongoing standardization efforts and working on their vision of the Web of Things. They provide their own version of a Thing Description and a Things Framework to connect smart devices that use other protocols than web-based ones to the Web of Things.

This thesis introduces the Universal Explorer for the Web of Things, a gateway that can interact with things providing a W3C or a Mozilla Thing Description. The Universal Explorer intends to provide a JavaScript, REST and WebSocket API and can translate and proxy between the two Thing Description formats.

[3] `https://iot.mozilla.org`

1.3. Outline

Chapter 1: Introduction
This introduction chapter provides an overview of the current situation of the Web of Things and the Internet of Things. Additionally, a brief overview of the general content and structure of the thesis is given.

Chapter 2: WoT Thing Description
The WoT Thing Description chapter describes the important concepts of the Thing Description, as this is one of the key parts for the new Web of Things architecture and an integral part for the work of this thesis.

Chapter 3: Related Work
The third chapter shows related work in this area, focussing on the current progress made in the Web of Things based on the new possible standards.

Chapter 4: The Universal Explorer
The Universal Explorer chapter introduces the web application developed as part of this thesis. The main focus here is put on the features that the Universal Explorer provides and the ways that it can be used by developers to facilitate the integration of smart things into other applications.

Chapter 5: Technical Implementation
This chapter involves the description of the implementation and software architecture. In addition it explains why and how the application was developed in the chosen way.

Chapter 6: Evaluation
In this chapter the Universal Explorer is assesed in a small demo scenario and performance tests are executed to measure the delay that it introduces to requests, as it acts as a gateway.

Chapter 7: Conclusion and Future Work
Finally, the last chapter concludes with the results of this thesis and presents an outlook for possible future research.

Appendix
The appendix includes a manual on how to install and use the Universal Explorer, a description and link of the repository of the project, commonly used acronyms and references used in this thesis.

1.4. Conventions

- The word Thing written with a capital letter refers to the Web of Things model of a real device. If the word is written with lower case letters, thing refers to the real device itself.
- In code examples and UML class diagrams, optional parameters are marked with a question mark (?), as used by the TypeScript[4] language.
- Abbreviations and acronyms are written in full before the first usage with the abbreviation in brackets, for example Web of Things (WoT). Additionally, Appendix C provides an overview of commonly used acronyms.

Apart from these conventions, the notation follows the regular documentation guidelines of the Software Engineering Group.

[4] `https://www.typescriptlang.org`

2 WoT Thing Description

With the Web of Things (WoT) Thing Description, the W3C is currently working on a possible new standard to describe the metadata and interfaces of Things in a machine-understandable way [20]. This makes it possible to easily integrate diverse devices from multiple manufacturers such as smart light bulbs or connected power switches and therefore allows diverse applications to interoperate.

The Thing Description can be considered as the entry point of a Thing that contains semantic metadata about the Thing itself and its WoT interface that describes the available interactions and how to interact with them. Interactions are divided into Properties, Actions and Events, as described in the following Section 2.1.

As the Thing Description is currently a W3C Working Draft, it is constantly evolving. This chapter provides an overview of the Thing Description at its current state.

2.1. Interaction Resources

2.1.1. Properties

Properties represent readable and possible writable data attributes of a Thing. An example for a Property would be the current power state or light color of a smart lamp. The

metadata of a Property consists of its name, the data schema, communication metadata and boolean flags that indicate if it is writable and observable.

2.1.2. Actions

Actions are processes or changes that can be invoked on a Thing. Contrary to a Property write, Actions in general need some time to complete and cannot be executed instantaneously. Example Actions could be to toggle the power state of a lamp or to open an automatic door. The Action metadata includes the name of the Action, separate schemas for the input and output data and the communication metadata.

2.1.3. Events

Events are mechanisms to be notified by a Thing if a certain condition is met. This could be for example an overheating message if a Thing reaches a certain temperature treshold. The metadata of an Event contains the name of the Event, the data schema and the communication metadata.

2.2. Serialization

To make the Thing Description machine-readable, the model is serialized in the JSON-LD format, an extension of JSON with additional semantic information. The following Listing 2.1 shows an example of a Lamp Thing that provides a status Property, a toggle Action and an overheating Event. Each interaction contains various meta information as described in Section 2.1 and provides the link to interact with it, here with the COAP protocol binding.

Since the start of this thesis, the W3C working group have already been working on the next version of the Thing Description serialization format. As visible in Listing 2.2, which represents the exact same Lamp Thing, it has been slightly simplified. Properties, Actions and Events are separate in the JSON-LD 1.1 format instead of one common interaction array with "@type" annotations.

The JSON-LD 1.1 format is part of the Editor's Draft at the time of writing, which means that it is still evolving. The Universal Explorer, which will be introduced in Chapter 4, supports the JSON-LD 1.0 format, as the 1.1 format could still change and did not yet exist at the time of development.

In addition to the new JSON-LD version, the Editor's Draft also includes a plain JSON serialization that excludes the additional semantic information that JSON-LD provides.

```
{
  "@context": ["https://w3c.github.io/wot/w3c-wot-td-context.jsonld"],
  "@type": ["Thing"],
  "name": "MyLampThing",
  "interaction": [
    {
      "@type": ["Property"],
      "name": "status",
      "schema": {"type": "string"},
      "writable": false,
      "observable": true,
      "form": [{
        "href": "coaps://mylamp.example.com:5683/status",
        "mediaType": "application/json"
      }]
    },
    {
      "@type": ["Action"],
      "name": "toggle",
      "form": [{
        "href": "coaps://mylamp.example.com:5683/toggle",
        "mediaType": "application/json"
      }]
    },
    {
      "@type": ["Event"],
      "name": "overheating",
      "schema": {"type": "string"},
      "form": [{
        "href": "coaps://mylamp.example.com:5683/oh",
        "mediaType": "application/json"
      }]
    }
  ]
}
```

List. 2.1: Example Thing Description serialization in JSON-LD 1.0 format

```
{
  "@context": "https://w3c.github.io/wot-thing-description/context/td-context.jsonld",
  "id": "urn:dev:wot:com:example:servient:lamp",
  "name": "MyLampThing",
  "properties": {
    "status": {
      "writable": false,
      "observable": false,
      "type": "string",
      "forms": [{
        "href": "coaps://mylamp.example.com:5683/status",
        "mediaType": "application/json"
      }]
    }
  },
  "actions": {
    "toggle": {
      "forms": [{
        "href": "coaps://mylamp.example.com:5683/toggle",
        "mediaType": "application/json"
      }]
    }
  },
  "events": {
    "overheating": {
      "type": "string",
      "forms": [{
        "href": "coaps://mylamp.example.com:5683/oh",
        "mediaType": "application/json"
      }]
    }
  }
}
```

List. 2.2: Example Thing Description serialization in JSON-LD 1.1 format

2.3. Type System

For the data schema definitions of the interaction types, the Thing Description includes a type system with semantic annotations. This describes the input and output data formats that are required to interact with an interaction. The type system includes simple data types like boolean, integer, number or string and more complex, nested object data types including arrays. For more precise data type definitions, the Thing Description additionally allows to restrict the values with the usage of restriction terms from the JSON schema validation[1]. The following example schema shown in Listing 2.3 defines that the value has to be an integer between 0 and 255.

```
"schema": {
    "type": "integer",
    "minimum": 0,
    "maximum": 255
}
```

List. 2.3: Data schema example with restriction

2.4. Protocol Bindings

The Thing Description supports various protocols to communicate with the thing that it describes. Protocol binding templates [21] allow the adaption to multiple protocol types. This is achieved by specifying the URL, protocol methods/options and the media type as the form object and by using a data schema model that is supported by the specific protocol.

Supporting multiple protocols offers the advantage that almost all devices can be supported, even battery-powered devices with limited computing power that need a protocol optimized for this type of device [9]. Besides HTTP, the Thing Description plans to support multiple alternative protocols like for instance CoAP and MQTT, which could play an essential part in the Web of Things.

CoAP, the Constrained Application Protocol, is a transfer protocol for constrained nodes and networks, specifically developed for the Web of Things that uses the REST architectural style [16][17]. MQTT is a publish/subscribe protocol that can be run on low-end and battery-operated devices and can operate over bandwidth-constraint networks [15].

2.5. Mozilla Web Thing Description

Besides the W3C, Mozilla is also working on supporting a Thing Description format. The Mozilla Web Thing Description is an unofficial draft that proposes a plain JSON serialization and concrete HTTP and WebSocket protocol bindings. Their proposal is intended to complement the W3C Thing Description [23].

[1] https://tools.ietf.org/html/draft-handrews-json-schema-validation-00

The JSON serialization format is very similar to the serialization that the W3C is currently working on, although it is a little simplified by not having to include multiple protocol bindings for the interactions itself. The following Listing 2.4 is an example of the serialization of a Mozilla Thing Description for the same Lamp Thing used for the W3C Thing Description examples. As visible, the links for Actions and Events are not included in the interaction itself in Mozilla Thing Descriptions, a separate links JSON property is used instead.

```
{
  "name":"MyLampThing",
  "description": "A WoT-connected Lamp Thing",
  "properties": {
    "status": {
      "label": "Status",
      "type": "string",
      "description": "Current status of the lamp",
      "href": "/things/lamp/properties/status"
    }
  },
  "actions": {
    "toggle": {
      "label": "Toggle",
      "description": "Toggle the lamp status"
    }
  },
  "events": {
    "overheating": {
      "description": "High temperature alert"
    }
  },
  "links": [
    {
      "rel": "properties",
      "href": "/things/lamp/properties"
    },
    {
      "rel": "actions",
      "href": "/things/lamp/actions"
    },
    {
      "rel": "events",
      "href": "/things/lamp/events"
    },
    {
      "rel": "alternate",
      "href": "wss://mylamp.example.com/things/lamp"
    }
  ]
}
```

List. 2.4: Example Mozilla Thing Description serialization in JSON format

3
Related Work

Together with the work on the Thing Description, there are various projects that implement the possible new standard by the W3C and Mozilla. This chapter presents the most important related projects. These include Node-WoT, the reference implementation of the WoT Scripting API, the Mozilla Things Gateway that implements the Mozilla Thing Description and the Mozilla Things Framework that allows to script Things according to the Mozilla Thing Description.

3.1. Node-WoT

The W3C is currently working on many proposals to enable the Web of Things. Besides the Thing Description and the general WoT Architecture [19], they are also proposing the WoT Scripting API. The WoT Scripting API [22] is an optional building block to easily write scripts to expose Things that do not support the new WoT Architecture. Besides exposing, it also allows to discover and consume things. It is built on top of the Thing abstraction model and the Thing Description.

The WoT Scripting API specification consists of three main parts:

- The WoT object, which is the entry point to discover, consume, and expose Things
- The ConsumedThing interface that allows clients to consume Things
- The ExposedThing interface that allows servers to expose Things over the network

The Node-WoT[1] project is the official reference implementation by the W3C of the WoT Scripting API and the Servient architecture.

[1] `https://github.com/eclipse/thingweb.node-wot`

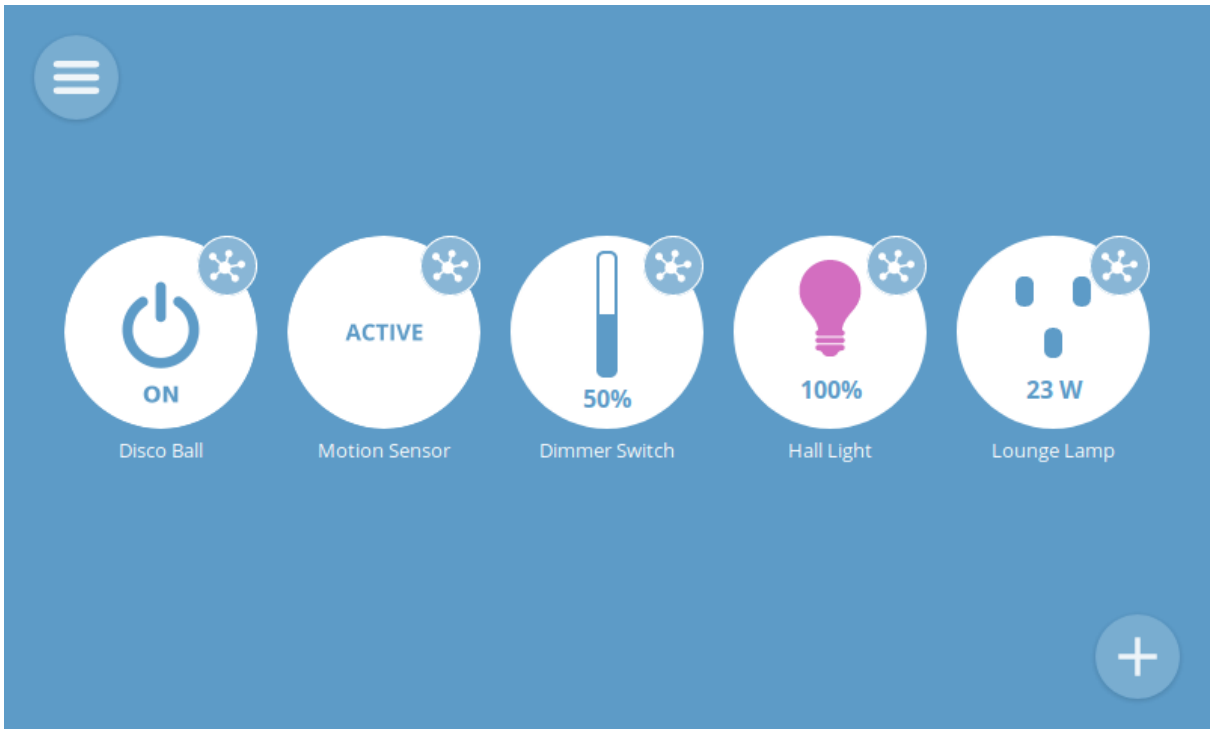

Fig. 3.1.: Screenshot of the Mozilla Gateway user interface (UI)

3.2. Mozilla Things Gateway

With the Things Gateway[2], Mozilla is developing a gateway that implements the Mozilla Thing Description [23] for various home automation devices that use protocols like ZigBee. It is directed towards consumers and provides a web-based user interface to control the connected devices, which is shown in Figure 3.1. It offers various other features like a rules engine to create simple automations or a smart assistant to control devices via voice or text commands.

3.3. Mozilla Things Framework

Similar to the WoT Scripting API, the Mozilla Things Framework[3] allows to script Things that provide a server according to the Mozilla Web Thing specification [23] and a Mozilla Thing Description. The Things Framework is a collection of re-usable software components that help developers to build their own Web Things. It is available in various programming languages including Node.js, Java, Python and Rust.

[2] `https://iot.mozilla.org/gateway`
[3] `https://iot.mozilla.org/things`

4 The Universal Explorer

The Universal Explorer is a gateway that can interact with things that provide a W3C or Mozilla Thing Description. By parsing the Thing Description, it can recognize the interactions that the thing provides and then model each real thing as a virtual JavaScript Thing that can control the corresponding real thing. The Universal Explorer then provides a RESTful HTTP API with a machine-readable OpenAPI[1] description and a WebSocket API for real-time, event-based updates that can be used by other applications. Additionally, it can generate W3C and Mozilla Thing Descriptions from the Virtual Things and therefore translate between the two Thing Description types.
Figure 4.1 visualizes the features of the Universal Explorer and shows how they interact with each other.

This chapter provides an overview of the features that are included in the Universal Explorer. In the next sections, those features are presented in more detail. The technical details of the implementation will be shown in Chapter 5.

4.1. Virtual Things (JavaScript API)

The JavaScript API allows to use the Virtual Things to interact with the real things from any JavaScript (or TypeScript) application. It provides methods to read and write

[1] `https://www.openapis.org`

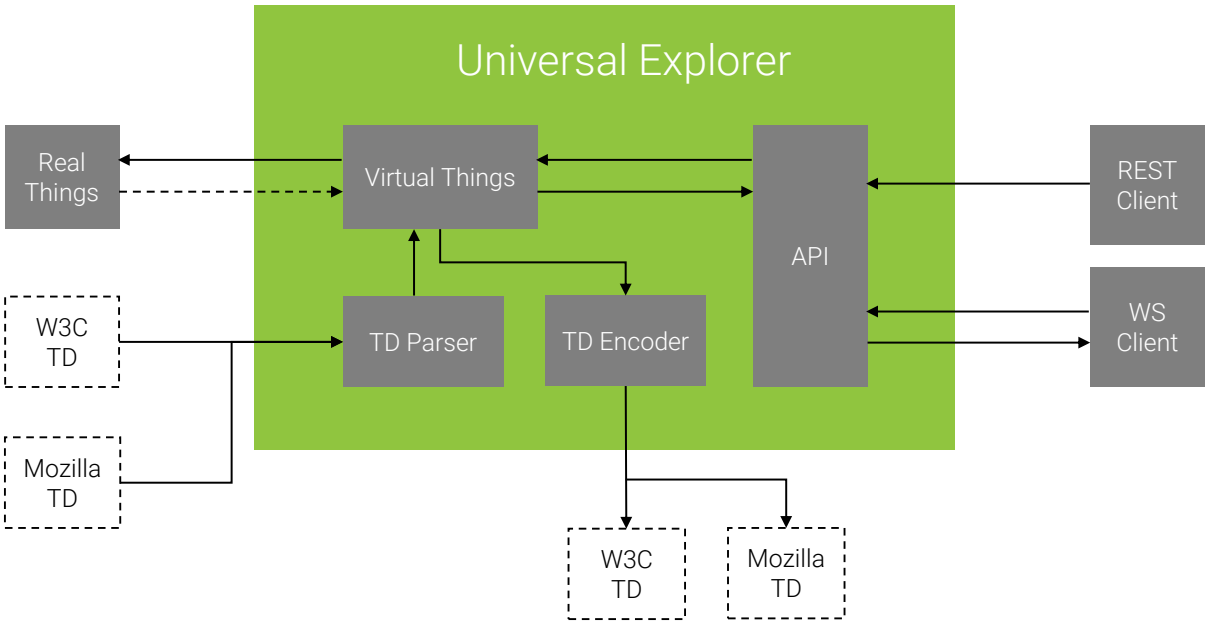

Fig. 4.1.: Overview of the Universal Explorer

Properties, invoke Actions and get past Event data. Moreover it is possible to subscribe to (and unsubscribe from) each interaction by providing a subscriber callback method that is called instantly each time an update occurs.

4.1.1. Methods

After the Virtual Thing is generated from a Thing Description, it provides the methods described in the following subsections. For the subscribe and unsubscribe methods, a subscriber callback function according to the ISubscriber interface must be provided. The method needs to accept the interaction and interaction data objects, as shown in Listing 4.1:

```
export interface ISubscriber {
    (interaction: InteractionPattern, data: InteractionData) :void;
}
```

List. 4.1: ISubscriber interface, used for subscriber callback functions

4.1.1.1. Properties

readProperty(name)

The readProperty method returns the value of the Property specified by the name parameter. It is async and therefore returns a Promise, as the real thing needs time to respond. In case the real thing does not respond, the method throws a TimeoutError.

writeProperty(name, data)

To update the value of a Property, the async writeProperty function is used. It requires the name of the Property and the new value as parameters. If the real thing does not respond, a TimeoutError is thrown. An incorrect data schema results in a RequestError.

subscribeToProperty(name, subscriber)

The subscribeToProperty method allows to subscribe to a Property by specifying the name of the Property and a subscriber callback function. The callback function is then called each time the value of the observed Property changes. It throws an InteractionError if the Property is not observable.

unsubscribeFromProperty(name, subscriber)

Similar to the subscribe method, the unsubscribeFromProperty method allows to unsubscribe from Property status updates by specifying the name and the reference to the subscriber callback function that was used.

4.1.1.2. Actions

invokeAction(name, data?)

To invoke an Action, the async method invokeAction can be used, with the name of the Action as a parameter. If the Action needs any additional information, it has to be specified as data parameter. Otherwise or if the data schema is not correct, a RequestError is thrown. In case the real thing does not respond, the method throws a TimeoutError.

subscribeToAction(name, subscriber)

The subscribeToAction method is used to subscribe to an Action specified by the name as a parameter and the subscriber callback function. The callback function is called each time the Action is executed.

unsubscribeFromAction(name, subscriber)

To unsubscribe from a previously subscribed Action, the unsubscribeFromAction function is used with the name and subscriber callback function used.

4.1.1.3. Events

getEventData(name, newerThan?, limit?)

The getEventData function allows to get data on Events that have happened in the past. It requires the name of the Event as a parameter. Optionally, the amount of data can be limited by specifying the number using the limit parameter. Aditionally, with the newerThan parameter it is also possible to only get Event data newer than a timestamp.

subscribeToEvent(name, subscriber)

The subscribeToEvent method allows to receive instant notifications as soon as an Event occurs. To subscribe, the function has to be called with the name of the Event and a subscriber callback function.

unsubscribeFromEvent(name, subscriber)

To unsubscribe, the unsubscribeFromEvent function has to be called with the name of the Event and the subscriber callback function used.

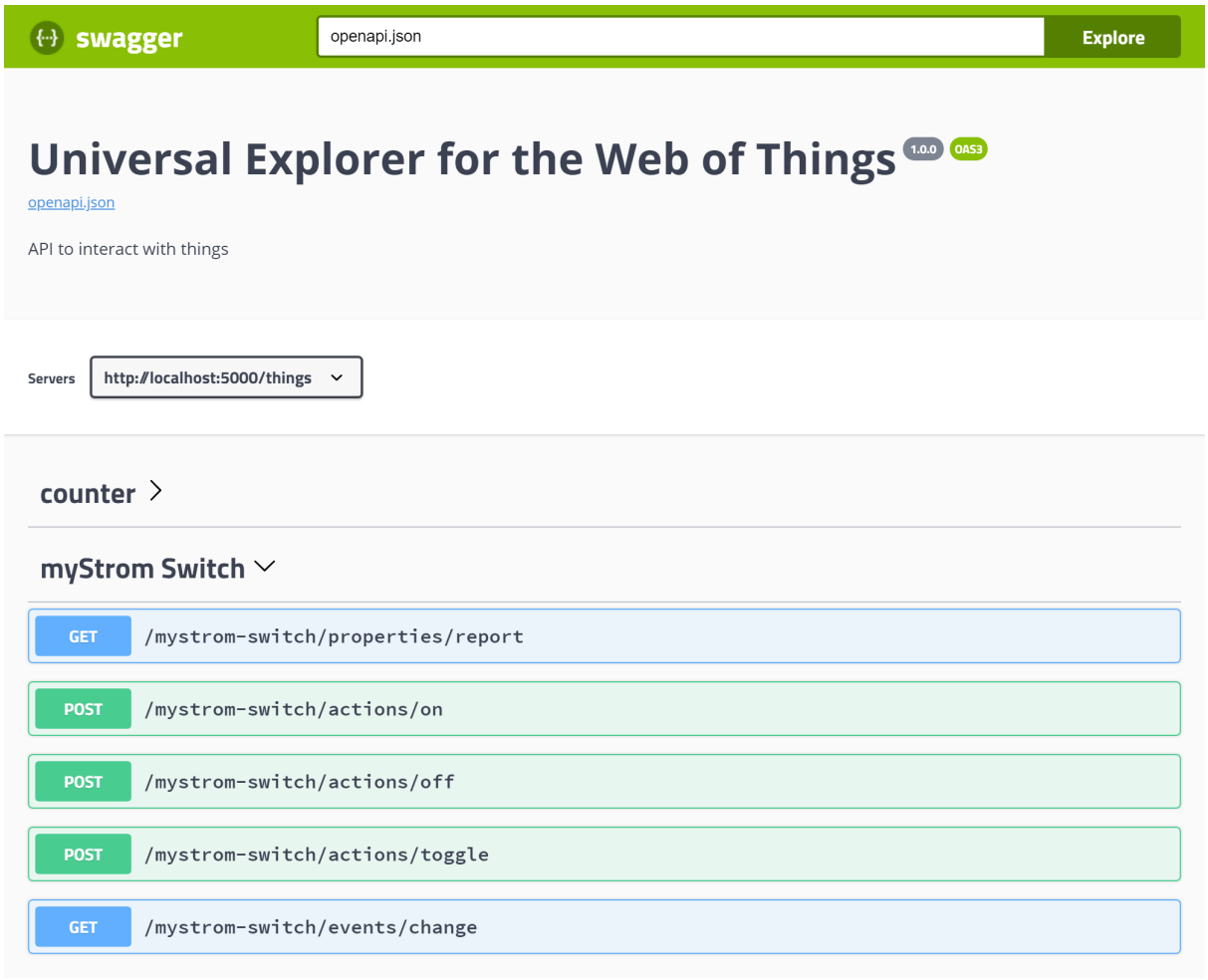

Fig. 4.2.: Swagger UI interface with example Things

4.2. REST API

The REST API [10] enables other applications to control all the things connected to the Universal Explorer over HTTP requests. It is documented by an automatically generated, machine-readable OpenAPI documentation that includes models for the input and output data of each interaction.

To support Event notifications, the API provides a GET Event resource, that allows to poll a list of the latest events that have taken place. This approach was chosen because the REST architecture does not include a suitable way to directly notify a client, as requests need to be initiated by the client [11]. WebHooks (also known as callbacks[2]) would have been a possibility, however those are for server to server communication. Real-time notifications have therefore been implemented by using WebSockets, as described in Section 4.3.

Using Swagger UI[3], which is served by the Universal Explorer, the available Things and

[2] https://swagger.io/docs/specification/callbacks
[3] https://swagger.io/tools/swagger-ui

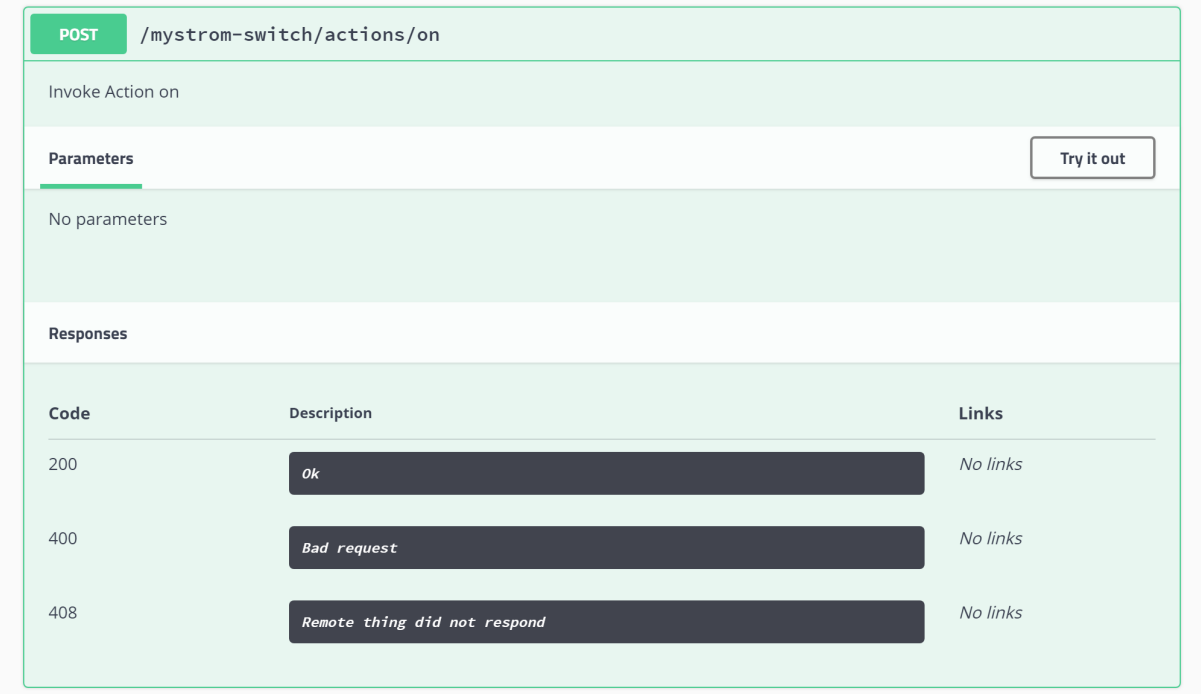

Fig. 4.3.: Example interaction expanded in Swagger UI

interactions can then be explored and consumed directly in the browser. Figure 4.2 demonstrates Swagger UI with some example Things and interactions. The interactions are grouped by the Thing that they belong to, and can then be expanded by clicking on them. The expanded view provides additional details about the interaction and allows to directly interact with the corresponding API endpoint by using the "Try it out" button, as shown in Figure 4.3.

4.2.1. Structure

The endpoints of the REST API are structured by Thing, interaction type and name of the interaction. For example, the URI of the Property "count" of a "Counter" thing would be "/counter/properties/count/". The interaction types are the same as in the W3C Thing Description working draft, Properties, Actions and Events as described in Chapter 2.

4.2.1.1. Properties

To get the value of a Property, a GET request has to be made to the corresponding Property endpoint. Besides the name of the Property, no additional parameters are necessary.

Updating the value of a Property is possible by sending a PUT request with the new value as body parameter according to the data schema.

The API then either responds with a HTTP status 200 and the value of the Property, a 400 error code if the name of the Property is wrong or a 408 error code in case the Universal Explorer can not reach the corresponding real thing.

4.2.1.2. Actions

Actions can be invoked by making a POST request to the corresponding Action endpoint. An Action can have additional parameters beside the name of the Action. If this is the case, the data model of the payload is specified in the OpenAPI description.

If the Action request was successfully passed to the real thing, the API responds with a 200 success status code. Otherwise, either a 400 error message with the rejection message of the real thing or a 408 (not reachable) error message is returned.

4.2.1.3. Events

To make it possible to get Event data using REST, the Universal Explorer provides a GET Event resource. The Universal Explorer monitors the Events of a Thing, logs and stores the Event notifications that occur and allows clients to poll the Event data using the GET Event resource. There are two optional parameters that the client can provide: a "timestamp" parameter to only get Event data newer than the specified timestamp, and a "limit" parameter to limit the maximum amount of Event notifications that are returned for the specific Event.

If the request was correct, the API responds with a 200 status code. Otherwise, a 400 bad request status code is returned.

4.2.2. Thing Description

Besides the Things endpoints, the REST API additionally provides endpoints to add new Things from a Thing Description and to generate a Thing Description from a Thing.

To add a new Thing from a Thing Description, a POST request has to be sent to `/td` or to `/td/moz` for Mozilla Thing Descriptions with the Thing Description serialization as payload. The new Thing is then automatically added to the Universal Explorer and the OpenAPI description is regenerated.

To get a generated Thing Description from a Thing added to the Universal Explorer, a GET request can be sent to `/td/name` to receive a serialized W3C Thing Description or to `/td/moz/name` to receive the Mozilla Thing Description, where the name part of the URL is the name of the Thing. Those generated Thing Descriptions can then be used by other applications, the Universal Explorer acts as a proxy for the requests.

4.3. WebSocket API

In addition to the REST API, the Universal Explorer provides a WebSocket API. The main advantage that WebSockets provide over REST is two-way communication. This makes it possible to subscribe to interactions of a thing and get instantly notified as soon as an interaction happens or an interaction value changes, without having to poll manually.

4.3.1. Protocol

The message protocol used for the WebSocket API is based on simple JSON messages. Each message consists of a "messageType" key that indicates the type of the message and a "data" key for the payload of the message. The protocol used is based on the "webthing" protocol proposed by Mozilla [23], extended with some additional features. Those extensions include a more advanced subscription management and the ability to request Property values over the WebSocket API.

To open a WebSocket connection, the default JavaScript WebSocket API[4] can be used, which takes care of the connection setup. The WebSocket API provides a separate endpoint for each thing in order to keep the protocol as simple as possible.

After the connection is establised, messages can be sent by the client to either subscribe to an interaction or for interaction requests, which are followed by a response either immediately or as soon as an interaction update happens.

The following section provides an overview over the possible message types.

4.3.2. Message Types

4.3.2.1. Subscription Management

subscribe Message

The subscribe message as shown in Listing 4.2 subscribes the client to all interactions of the corresponding Thing. By default, there are no active subscriptions for a new client.

```
{
    "messageType": "subscribe",
    "data": ""
}
```

List. 4.2: subscribe message

unsubscribe Message

The unsubscribe message shown in Listing 4.3 allows to unsubscribe from all previously subscribed interactions of the Thing at once.

[4] https://www.w3.org/TR/websockets

```
{
    "messageType": "unsubscribe",
    "data": ""
}
```

List. 4.3: unsubscribe message

addSubscription Message

In addition to the global subscribe and unscubscribe messages, the addSubscription message allows to subscribe to specific interactions only. The client can subscribe to one or multiple interactions at once by specifying the interaction type as the key and then either the name of a single interaction as a string or multiple interaction names as an array of strings as value. An example is shown in listing Listing 4.4.

```
{
  "messageType": "addSubscription",
  "data": {
    "property": "propertyName",
    "action": ["actionName", "actionName2"]
  }
}
```

List. 4.4: addSubscription message

removeSubscription Message

To unsubscribe from specific interactions, the removeSubscription message is used. The syntax used to specify the interactions to unsubscribe from is exactly the same as for the addSubscription message. Listing 4.5 shows an example message.

```
{
  "messageType": "removeSubscription",
  "data": {
    "property": ["propertyName", "propertyName2"],
    "event": "eventName"
  }
}
```

List. 4.5: removeSubscription message

4.3.2.2. Interaction Requests

getProperty Message

The getProperty message allows to instantly request one or multiple Property updates. The name of the Property to obtain is set as the key, the value has to be empty. An example is shown in Listing 4.6.

```
{
  "messageType": "getProperty",
  "data": {
    "propertyName": "",
    "propertyName2": ""
  }
}
```

List. 4.6: getProperty message

setProperty Message

To update the value of one or multiple properties, the setProperty message is used. Similar to the getProperty message, the name of the Property is set as the key, with the new Property value as the JSON value. See Listing 4.7 for an example.

```
{
  "messageType": "setProperty",
  "data": {
    "propertyName": 10,
    "propertyName2": true
  }
}
```

List. 4.7: setProperty message

requestAction Message

With the requestAction message it is possible to execute one or multiple actions simultaneously. As shown in Listing 4.8, the Action name is set as the key. If the Action requires any data as a parameter, it can be set as the value. Otherwise the value can be an empty string.

```
{
  "messageType": "requestAction",
  "data": {
    "actionName": "",
    "actionName2": "data"
  }
}
```

List. 4.8: requestAction message

4.3.2.3. Interaction Responses

propertyStatus Message

The propertyStatus message is sent to all subscribers as soon as the value of the Property changes. The message body contains the name of the Property as the key and the new value as the JSON value. Listing 4.9 shows an example message for a Property value update.

```
{
  "messageType": "propertyStatus",
  "data": {
    "propertyName": 10
  }
}
```

List. 4.9: propertyStatus message

action Message

Each time an Action is executed, the action message is sent so all subscribers. It contains the name of the Action that was executed and the Action result data if one is available for the Action in question, this is shown in Listing 4.10.

```
{
  "messageType": "action",
  "data": {
    "actionName": true
  }
}
```

List. 4.10: action message

event Message

The event message is sent to all subscribers as soon as an Event happens. The message data contains the name of the Event and its value, as shown in Listing 4.11.

```
{
  "messageType": "event",
  "data": {
    "eventName": true
  }
}
```

List. 4.11: event message

4.3.2.4. Status Messages

After the client sends a message, the server responds with a short status message to either confirm or reject the message, or to report errors that are not the fault of the client. These messages contain a HTTP status code and a short error or success message. An example for a 400 error is shown in Listing 4.12.

```
{
    "messageType": "error",
    "data": {
        "status": 400,
        "message": "error message"
    }
}
```

List. 4.12: status messages

5 Technical Implementation

The Universal Explorer is a server-side Node.js[1] web application. It is written in TypeScript[2], a superset of JavaScript with optional static typing.

Node.js was chosen as it allows to write high-performance applications that can handle many connections concurrently, using asynchronous I/O with an event-driven programming model [12]. As JavaScript is a dynamically typed language, only a limited amount of code completion is possible. Due to static typing, TypeScript facilitates improved code completion and can detect 15% of the bugs automatically, according to a study by Gao *et al.* [13].

The source code of the Universal Explorer is licensed under the MIT License[3] and can be found on GitHub: `https://github.com/linusschwab/wot-universal-explorer`

5.1. Architecture

To make it possible to easily extend and adapt the application and to allow to reuse components, the architecture of the Universal Explorer is based on the MVC pattern [14].

[1] `https://nodejs.org`
[2] `https://www.typescriptlang.org`
[3] `https://opensource.org/licenses/MIT`

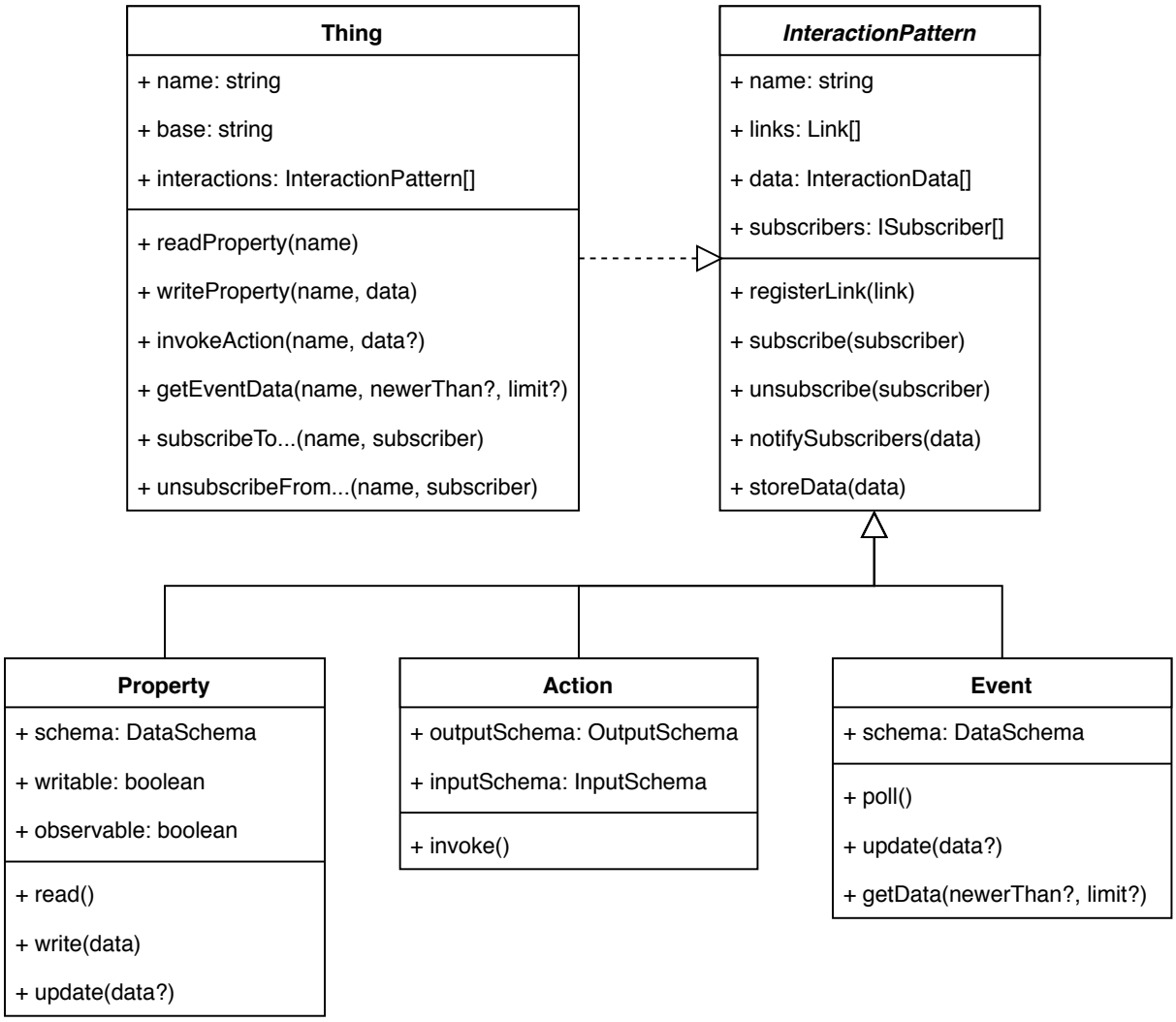

Fig. 5.1.: UML of the Object Model

The application is divided into the Object Model that represents the Virtual Things (JavaScript API), controllers to provide the REST and WebSocket API and parser and encoder classes for the W3C and Mozilla Thing Descriptions. The following subsections will explain the different application parts in more detail.

5.1.1. Object Model

The Object Model is the central part of the Universal Explorer. It represents the real things as Virtual Things and provides the JavaScript API, which is also used by the controllers to provide the REST and WebSocket API. The structure of the Object Model closely represents the structure of the W3C Thing Description working draft that was introduced in Chapter 2.

As shown in Figure 5.1, the Object Model is represented by the Thing class. Besides the methods provided for the JavaScript API, it consists of some basic attributes such as the name and base URI of the real thing and contains the available interactions. To keep the UML readable, some class attributes and methods have been left out. Optional method

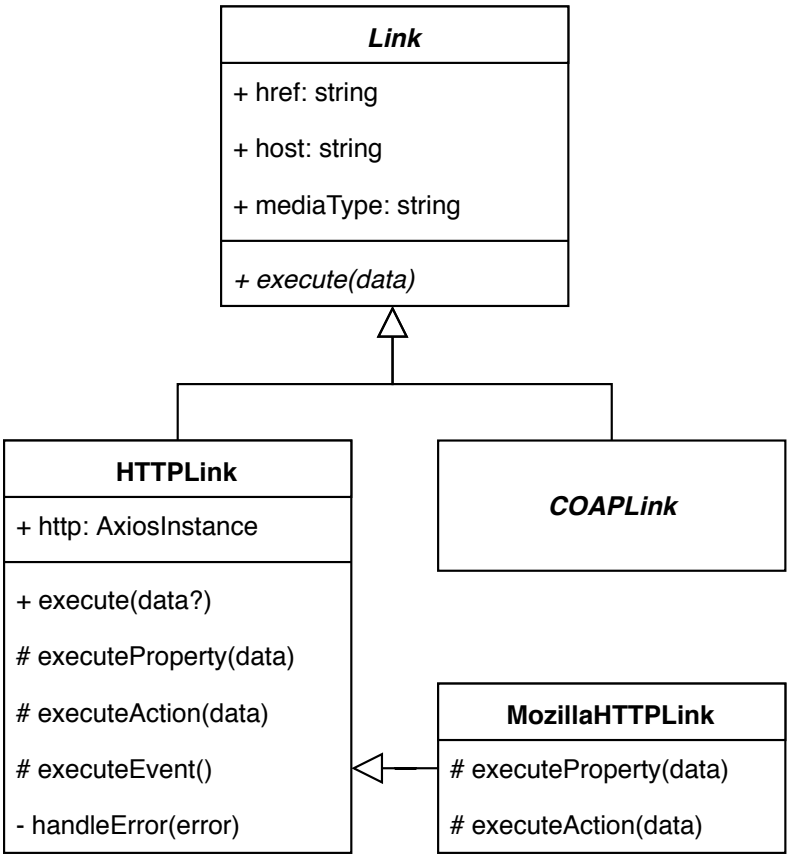

Fig. 5.2.: UML of the link model

parameters are marked by a question mark.

The interactions of the real thing are modelled by the Property, Action and Event classes, that inherit from an abstract InteractionPattern class. They contain the input and output data schemas, implement the Observer pattern to notify subscribers and manage the links that allow to communicate with the interactions of the real thing.

Even though the Universal Explorer only supports the HTTP protocol for links in the current version, the HTTPLink class extends an abstract Link class, as visible in Figure 5.2. The reason for this is to make it as easy as possible to add additional protocol bindings like COAP or MQTT later.

Each HTTPLink class instance handles the communication with a REST API endpoint provided by the real thing. To achieve this, the Promise-based (and therefore supporting async/await) axios[4] library was used. Additionally, the class provides an error handling method that catches and interprets the generic errors thrown by the network communication. Instead it throws new custom, meaningful error classes that can easily be handled in other parts of the application. This is shown in Listing 5.1.

[4] `https://github.com/axios/axios`

```
private handleError(e: any) {
    if (e.code === 'ECONNABORTED'
     || e.code === 'EHOSTUNREACH'
     || e.code === 'ENETUNREACH') {
        throw new TimeoutError('Remote thing did not respond');
    }else if (e.response && e.response.status === 400) {
        throw new RequestError('Request data schema not correct');
    }else {
        throw e;
    }
}
```

List. 5.1: HTTP error handling

The MozillaHTTPLink subclass is used for minor adaptions in the communication with Mozilla Things. This is necessary because according to the Mozilla Thing Description, primitives like booleans or integers have to be wrapped in an object, while the W3C Thing Description specifies that it must not be wrapped. To solve this, the owerwritten methods in the MozillaHTTPLink class automatically wrap unwrapped primitives before forwarding them to the real thing and unwrap the response.

To communicate with the WebSocket that the Mozilla Thing Description includes, a MozillaThing subclass of the regular Thing class is used. This class also uses the ws (client) library and connects to the WebSocket of the real thing. It further tries to reconnect automatically if the connection is lost.

5.1.2. From the TD to the Object Model and Back

5.1.2.1. Parsing

To add a new thing to the Universal Explorer, the W3C or Mozilla Thing Description is parsed by the application. The JSON and JSON-LD files are initially transformed into a JavaScript object using the `JSON.parse()` method. From this plain object, the Object Model is then iteratively created by manually parsing the different parts and instantiating the corresponding Object Model class by the TDParser and MozillaTDParser classes.

The Universal Explorer automatically reads Thing Descriptions that are placed in the correct folders (`public/td` or `public/td-moz`) at the start of the application. Alternatively it is also possible to add new things using the REST API during runtime, as explained in Section 4.2.2.

Figure 5.3 shows the parsing process over the REST API as a sequence diagram. The API request is handled by the TDController, which forwards the Thing Description to the TDParser that parses it into a Virtual Thing. Finally, the Thing is then stored in the ThingsManager class that is responsible for storing and managing the Thing instances of the Universal Explorer.

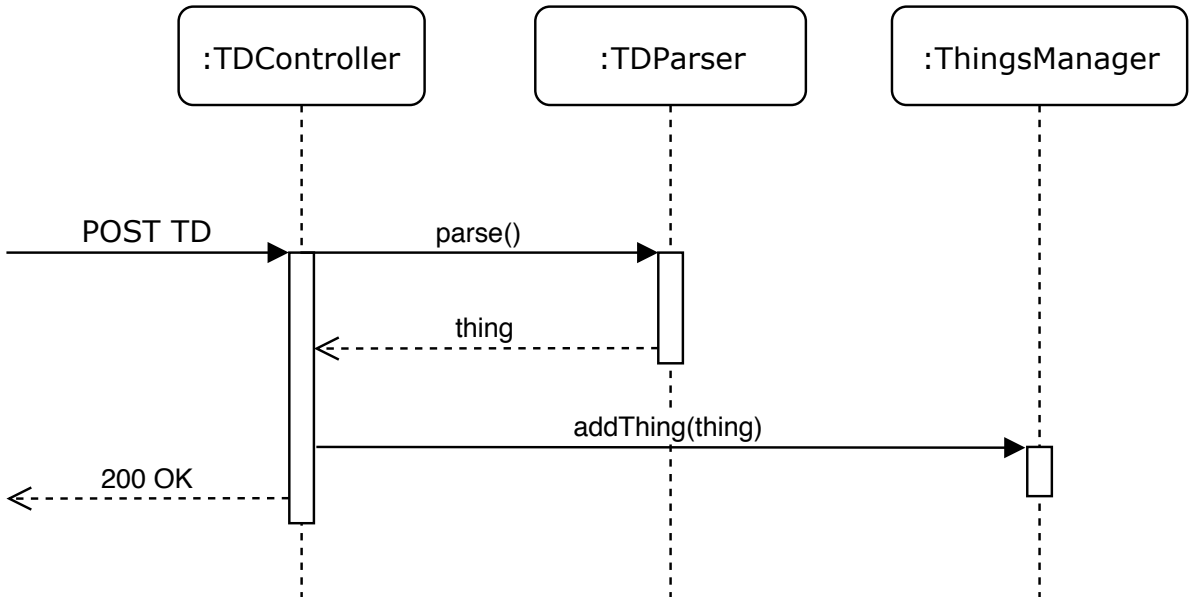

Fig. 5.3.: Sequence diagram of the Thing Description parsing

5.1.2.2. Encoding

Virtual Things are encoded back into W3C or Mozilla Thing Descriptions using the `JSON.stringify()` method. For each class of the Object Model, the TDEncoder and MozillaTDEncoder classes contain a custom replacer function that transforms the JavaScript class instance into the correct JSON or JSON-LD representation.

To get the encoded Thing Descriptions, the REST API can be used. The Universal Explorer acts as a proxy for these generated Thing Descriptions, as the requests formats are different for the W3C and Mozilla Thing Descriptions and the real thing might not support both types.

Figure 5.4 explains the encoding process as a sequence diagram. The Thing Description request from the client is handled by the TDController that retrieves the Thing from the ThingsManager with the Thing name provided by the client. The Thing is then forwarded to the TDEncoder that handles the actual encoding and returns the generated Thing Description as a string. The TDController then returns the Thing Description to the client.

5.1.3. OpenAPI Documentation Generation

The OpenAPI documentation is directly generated from the Object Model. Similar to the Thing Description encoding, the OpenAPIEncoder class generates the JSON document using the `JSON.stringify()` method with custom replacer functions for each class instance of the Object Model.

As the structure of the OpenAPI 3.0 standard is not as similar as the Thing Description is to the Object Model, various helper functions were necessary. For example, the

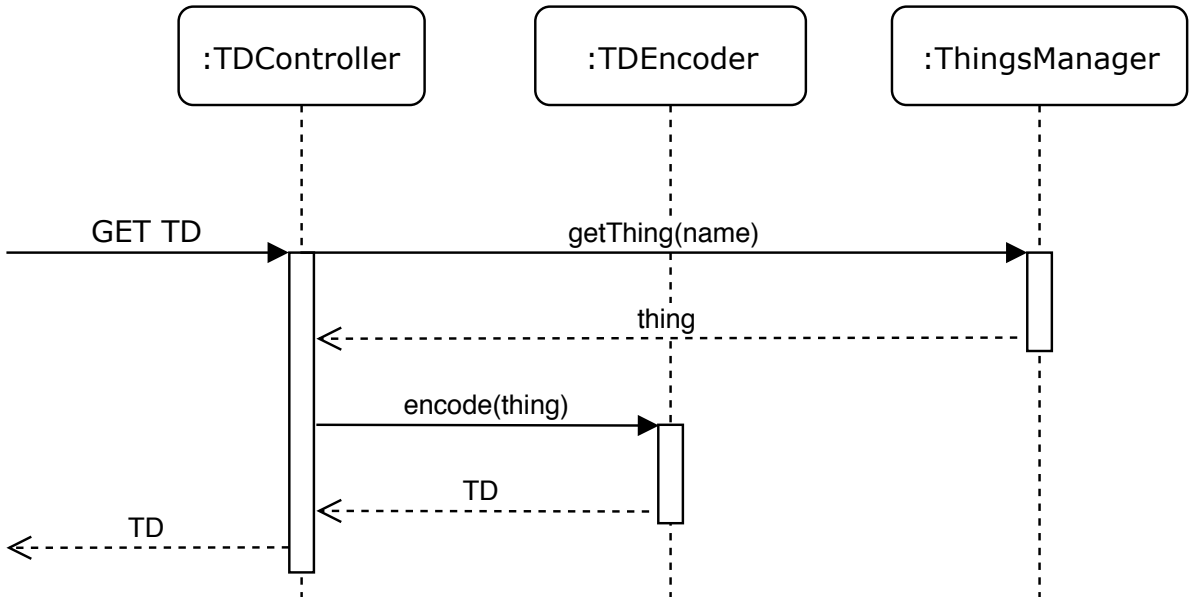

Fig. 5.4.: Sequence diagram of the Thing Description encoding

Object Model interactions also include Operation classes, that specify the HTTP REST operations that are required to interact with this interaction. This made the OpenAPI document generation much easier, as the interactions need to be represented as path objects with operations at the top level.

5.1.4. Controllers

For the REST and WebSocket APIs, the Universal Explorers uses controllers to handle requests. The IndexController is responsible for serving Swagger UI. The TDController provides endpoints to add new W3C and Mozilla Thing Descriptions and to generate both Thing Description types from the Virtual Things Object Model. The ThingsController handles the actual thing requests by both REST and WebSocket clients and forwards them to the Virtual Things. All controllers extend from an abstract BaseController class that provides the basic structure and helper methods.

To provide the REST API, Koa was used. Koa[5] is a modern web framework that aims to provide a lightweight, robust foundation for API development and is fully based on async/await, without the use of callbacks. For the WebSocket API, the ws[6] library was used, a simple and fast WebSocket client and server implementation.

Figure 5.5 shows an UML of the controllers. The `ctx` parameter used by almost all methods is the Koa Context[7] object that contains the request and response objects used by Node. The `ws` parameter is a WebSocket connection. As visible in the UML, the ThingsController is supported by the WebSocketManager class to handle WebSocket requests. The WebSocketManager is responsible for the connection setup, message parsing

[5] `https://koajs.com`
[6] `https://github.com/websockets/ws`
[7] `https://koajs.com/#context`

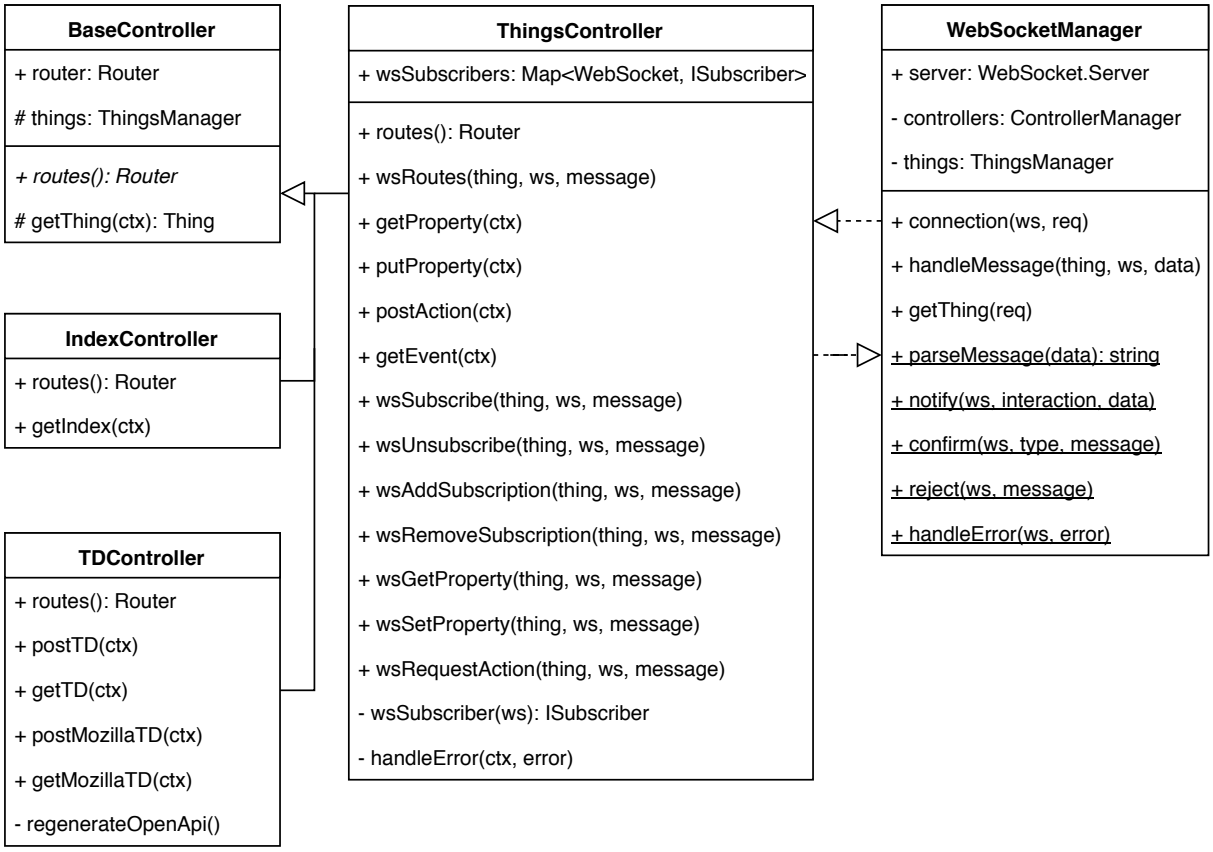

Fig. 5.5.: UML of the controllers

and validating. Furthermore it provides static helper methods to send responses. Valid messages are forwarded and routed to the correct ThingsController method.

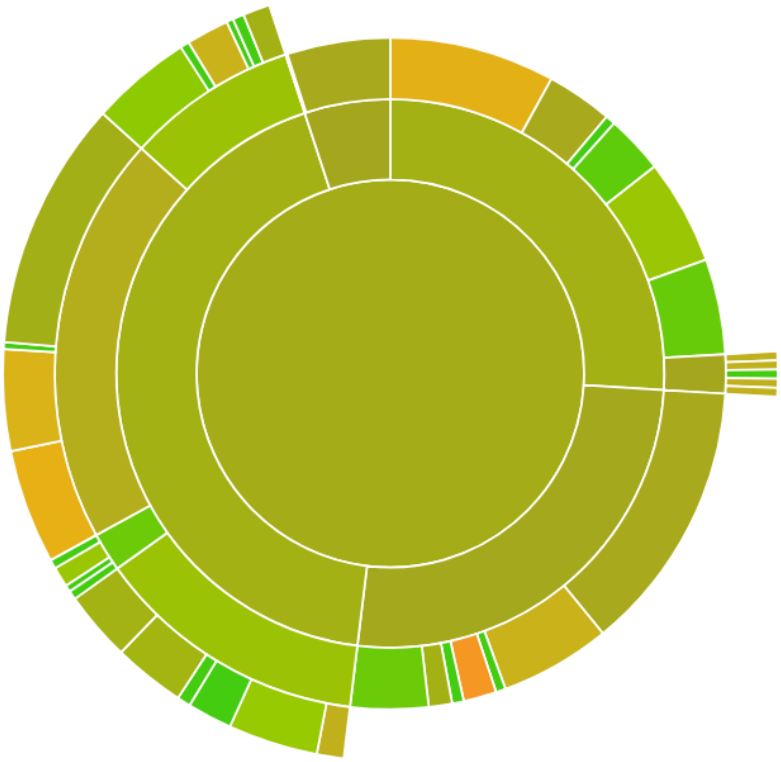

Fig. 5.6.: Visualization of the test coverage of the code

5.2. Testing

To increase the robustness of the code and to minimize the amount of bugs, Jest[8] was used to write unit and integration tests. Jest is a testing platform developed by Facebook that runs tests fast and sandboxed and provides instant feedback. It introduces snapshot testing to easily test serializable values like for example JSON messages used for the WebSocket API.

In the end, a total of 97 tests were written, which resulted in a test coverage (line coverage) of 87.26% according to Jest, or 82.29% according to Codecov. Figure 5.6 provides an overview of the test coverage of the different classes of the whole application. The colors indicate the coverage percentage: green means a high test coverage (up to 100%), yellow a medium coverage, and red a low coverage (down to 50% or lower). This coverage sunburst was generated by Codecov[9]. An interactive version which allows to inspect the coverage of the specific classes by hovering and clicking on them can be found here:
`https://codecov.io/gh/linusschwab/wot-universal-explorer`

[8] `https://jestjs.io`
[9] `https://codecov.io`

6 Evaluation

To evaluate the Universal Explorer, a small demo scenario was used with multiple devices interacting with each other in a real-world environment. Additionally, performance tests were executed to measure the delay that it introduces, as it acts as a gateway.

6.1. Demo Scenario

6.1.1. Device Setup

For the scenario, a total of four smart devices were used. Those include a myStrom Switch, a Philips Hue lamp, a Philips Hue motion sensor and a Nordic Thingy:52. Figure 6.1 provides an overview of the device setup, which will be explained in detail in the following subsections.

The Universal Explorer run on a laptop connected to a WiFi router, to which the myStrom Switch was directly connected. The Mozilla Gateway and the Mozilla Things Framework were both run on a Raspberry Pi[1] that was connected to the same router.

6.1.1.1. myStrom Switch

The myStrom WiFi Switch[2] is a power switch with a relay, that provides a simple REST API. This is one of the devices that was used to test the Universal Explorer in general

[1] `https://www.raspberrypi.org`
[2] `https://mystrom.ch/wifi-switch-ch`

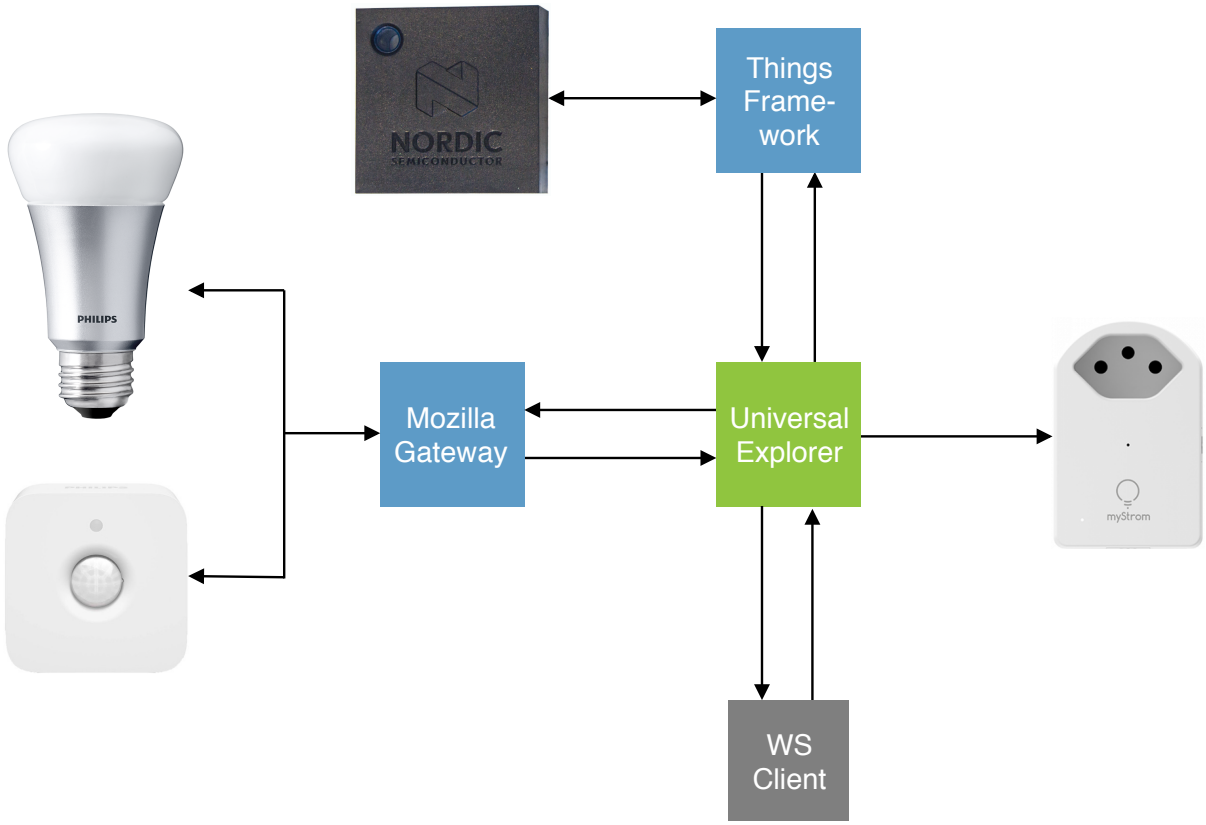

Fig. 6.1.: Device setup of the demo scenario

during development. As it does not provide a Thing Description, a W3C one was written for its API. Listing 6.1 shows the Thing Description, including the Property Report that shows the current power usage and the Action Toggle that switches the relay state. Additionally the myStrom Switch includes On and Off Actions to set the relay to the specific state, those were left out to keep the example readable.

For the scenario, the myStrom Switch was used with a regular fan attached that could be turned on and off over the REST API. It was directly connected to the Universal Explorer using the written W3C Thing Description.

6.1.1.2. Philips Hue Lamp and Motion Sensor

With the Hue[3] system, Philips has an ecosystem of smart lamps and accessories that communicate over ZigBee and are connected to a Hue Bridge WiFi gateway that provides a REST API. In the scenario, a Hue color lamp and a Hue motion sensor were used. Both of them were connected to the Mozilla Gateway, which already supported the Hue Bridge. The Mozilla Gateway generated a Mozilla Thing Description, that could be parsed by the Universal Explorer and used to communicate with the REST and WebSocket API of the Mozilla Gateway.

[3] `https://www.meethue.com`

```
{
  "@context": ["https://w3c.github.io/wot/w3c-wot-td-context.jsonld"],
  "@type": ["Switch"],
  "name": "myStrom Switch",
  "base": "http://192.168.0.3/",
  "interaction": [
    {
      "@type": ["Property"],
      "name": "report",
      "schema": {
        "type" :"object",
        "field": [
            {
                "name": "power",
                "schema": {"type": "integer"}
            },
            {
                "name": "relay",
                "schema": {"type": "boolean"}
            }
        ]
      },
      "writable": false,
      "observable": false,
      "form": [
        {
           "href": "report",
           "mediaType": "application/json"
        }
      ]
    },
    {
      "@type": ["Action"],
      "name": "toggle",
      "inputSchema": null,
      "outputSchema": {
        "type" :"object",
        "field": [
          {
              "name": "relay",
              "schema": {"type": "boolean"}
          }
        ]
      },
      "form": [
        {
           "href":"toggle",
           "mediaType":"application/json"
        }
      ]
    },
    ...
  ]
}
```

List. 6.1: Thing Description written for the myStrom Switch

Fig. 6.2.: Simple hardcoded visualization of the demo devices

6.1.1.3. Nordic Thingy:52

The Nordic Thingy:52[4] is a small battery-powered device with multiple sensors like humidity or CO_2 sensors, a button and a color LED. It provides Bluetooth 5 low energy for the communication that was used to connect it to the Raspberry Pi. To communicate with the Universal Explorer, the Mozilla Things Framework was used to generate a Mozilla Thing Description and to provide a REST and WebSocket API server.

6.1.2. Interactions

With all these devices connected to the Universal Explorer, a very basic web application that acted as a WebSocket client was written. This web application visualized the state of the devices with hardcoded images to keep it as simple as possible and contained some automation scenarios written in JavaScript. Figure 6.2 shows the visualization of the devices used.

The following automations were used for the scenario:

- If the Hue motion sensor detects any motion, turn on the Hue lamp and off again after there was no motion detected anymore for a couple of seconds
- If the CO_2 concentration in the room measured by the Thingy:52 is high (above 1400 ppm), turn on the myStrom Switch with the connected fan (and off again as soon as it is below a certain treshold)
- If the button on the Thingy:52 is pressed, measure the color below it with its sensor and set the measured color as the light color of the Hue lamp

This small scenario demonstrated that the Universal Explorer can communicate with various real-world devices using both W3C and Mozilla Thing Descriptions, with a much reduced developer effort compared to having to manually integrate the devices.

[4] `https://www.nordicsemi.com/eng/Products/Nordic-Thingy-52`

6.2. Performance

As the Universal Explorer acts as a gateway and is therefore the middle-man in the communication between the client and the thing, it is important that it adds as little delay as possible to a request. To make sure that this is the case, multiple performance tests were executed.

6.2.1. Test Setup

To measure the performance of the Universal Explorer, the Apache JMeter[5] load testing tool was used. For each interaction tested, 100 requests were sent to both the thing directly and with the Universal Explorer as a gateway in between, with a small delay of 100 milliseconds between the requests. The average, median, minimum and maximum time calculated by JMeter were then compared to calculate the delay that the Universal Explorer introduces.

The test devices consisted of the myStrom Switch and a virtual Counter Thing provided by the Node-WoT library that was described in Section 3.1. Both the Universal Explorer and the virtual Counter Thing were run on the same computer, together with Apache JMeter to measure the delay introduced by the application as close as possible instead of the network delay. The myStrom Switch was connected via WiFi to a wireless router, which was connected to the computer via cable.

6.2.2. Results

6.2.2.1. myStrom Switch

For the myStrom Switch, the Property Report (returns the current power usage) and the Action Toggle (switches the relay state) were used to perform the performance test.

As visible in Table 6.1, the average response time for the direct Property request were about 6 milliseconds and 13 milliseconds over the REST API of the Universal Explorer. Therefore, a delay of 7 milliseconds is introduced here for the processing and forwarding of the request.

	Average	**Median**	**Min**	**Max**
Direct	6 ms	4 ms	3 ms	17 ms
Explorer	13 ms	12 ms	10 ms	25 ms
	+ 7 ms	+ 8 ms	+ 7 ms	+ 8 ms

Tab. 6.1.: Property Report of the myStrom Switch

For the Action request, the direct communication took 292 milliseconds and 313 milliseconds indirect on average. With 21 milliseconds, the delay was much larger. This is shown

[5] `https://jmeter.apache.org`

in Table 6.2.

The reason for this is that the API of the myStrom Switch uses GET requests for everything. However, the W3C Thing Description Working Draft specifies that Action requests have to be done as POST requests. As a workaround, the Universal Explorer tries again using a GET Request if a thing returns a 405 "method not supported" error message. As this requires an additional request, the total time required is increased by a couple of milliseconds.

	Average	**Median**	**Min**	**Max**
Direct	292 ms	281 ms	274 ms	394 ms
Explorer	313 ms	300 ms	291 ms	417 ms
	+ 21 ms	+ 19 ms	+ 17 ms	+ 23 ms

Tab. 6.2.: Action Toggle of the myStrom Switch

6.2.2.2. Virtual Counter

The second testing device used is a virtual Counter Thing. This is a simple JavaScript application that provides a REST API and an automatically generated W3C Thing Description and was run on the same machine that was used to run JMeter. The Property Count (returns the current counter value) and the Action Increment (increments the counter value by 1) were used to test the performance.

For the Count Property, a direct request took 2 milliseconds on average, the Universal Explorer required 8 milliseconds to respond. Therefore, a delay of 6 milliseconds was introduced on average, as visible in Table 6.3.

	Average	**Median**	**Min**	**Max**
Direct	2 ms	1 ms	1 ms	8 ms
Explorer	8 ms	8 ms	6 ms	16 ms
	+ 6 ms	+ 7 ms	+ 5 ms	+ 8 ms

Tab. 6.3.: Property Count of the virtual Counter Thing

A direct Action request took on average 3 milliseconds, an indirect request 10 milliseconds. The delay here is similar with 7 milliseconds, as shown in Table 6.4.

	Average	**Median**	**Min**	**Max**
Direct	3 ms	3 ms	2 ms	12 ms
Explorer	10 ms	9 ms	8 ms	21 ms
	+ 7 ms	+ 6 ms	+ 6 ms	+ 9 ms

Tab. 6.4.: Action Increment of the virtual Counter Thing

6.2.2.3. Summary

To summarize, the Universal Explorer increases the time it takes to complete a regular request by approximately 6 to 8 milliseconds. This is clearly measureable, but not perceptible by users. Faster response times could most likely be achieved over a WebSocket connection. However, the real things would need to communicate over the WebSocket protocol too to really decrease the response time.

7
Conclusion and Future Work

7.1. Conclusion

The Universal Explorer for the Web of Things makes it possible to interact with smart things implementing the W3C or Mozilla Thing Description. The gateway can parse the available interactions of a thing and provides a RESTful API with OpenAPI documentation, a WebSocket API and a JavaScript (TypeScript) API. It serves as a proof-of-concept that a possible new standard like the W3C Thing Description based on open web technologies could make it much easier for developers to connect and interact with multiple smart things. This could allow them to spend their time on actually building useful applications instead of wasting it to connect proprietary systems.

Not only for developers, but especially also for regular consumers the future looks bright. If the possible new Web of Things architecture gains a broad adoption, it could be easily possible to combine smart devices from multiple manufacturers - without being dependent on the app and device ecosystem of a specific manufacturer to support a particular use case.

The main contributions of this thesis are the following:

1. A simple parser and encoder for W3C and Mozilla Thing Descriptions that allows to translate between the two types
2. A JavaScript (TypeScript) API with interactive Virtual Things that each represent a real smart thing
3. A RESTful API with automatically generated endpoints for all the interactions of a Thing, with machine-readable OpenAPI documentation including schemas for the input and output data of each interaction

4. A WebSocket API with real-time, event-based interaction updates and interaction subscription management for each Thing

7.2. Future Work

7.2.1. Web of Things

For the Web of Things to succeed, a broad adoption is necessary. Manufacturers will need to design new devices with direct support for the Web of Things architecture to ensure that they are compatible with other devices instead of building proprietary ecosystems. Large, already established companies could be less interrested to support the possible new standard. It is benefical for them if customers have to buy devices from their ecosystem for them to be compatible with the existing devices. Nevertheless, this could be a chance for smaller companies to increase their market share.

Besides that, the W3C, Mozilla and other organizations involved in the Web of Things should work together and unify their standardization efforts for the architecture for the Web of Things to end up with one, solid standard.

7.2.2. Possible Improvements for the Universal Explorer

7.2.2.1. Support Additional Protocol Bindings

Currently, the Universal Explorer only supports the HTTP protocol, other protocol bindings in a W3C Thing Description file like CoAP or MQTT are being ignored. However, the application was built with expandability in mind. It should therefore be easy to implement and support additional protocols. It would even be possible to provide APIs for the additional protocols and to translate between the different protocols, as the current REST and WebSocket APIs already use the Virtual Thing proxy objects of the common JavaScript API to achieve decoupling. This would allow developers to use regular HTTP REST requests to communicate with a CoAP thing for example.

7.2.2.2. Full Type System and Validation of Data Schemas

For the OpenAPI documentation generation, the Universal Explorer only supports simple data types at the moment. More complex, nested object structures are not supported yet, as the W3C Thing Description currently is a working draft with regular changes. This is something that should be supported in a possible future version. Additionally, the data schemas are not validated in the current version of the Universal Explorer, however in case the real thing does not accept the data of a request a bad request error message is sent to the requester.

7.2.2.3. Scriptable Things

In addition to Virtual Things, the Universal explorer could also offer the possibility to directly script JavaScript Things that are connected to legacy things. This could be

easily added, however as mentioned in Section 3.1 and 3.3 there are already existing solutions available for this, that provide W3C and Mozilla Thing Description compliant Thing servers with APIs. It is therefore already possible to import and parse the Thing Descriptions generated by those libraries. Directly integrating scriptable things would make it obsolete to run an additional server.

7.2.2.4. User Interface and Visualization of Things

A user interface with a visualization of the Things and their interactions would make the Universal Explorer appealing to regular users instead of just developers. With the semantic information that the W3C Thing Description provides, it would even be possible to build a smart, easy to use mashup editor similar to Node-RED[1]. Any future device implementing a W3C Thing Description could then easily be used for complex automations, without the need to understand programming languages.

[1] `https://nodered.org`

A

Installation Manual

This short manual provides directions on how to install and use the Universal Explorer. The application is open source and can be downloaded on GitHub as shown in Appendix B.

A.1. Prerequisites

The latest version of Node.js is required to run the application. The official website nodejs.org[1] allows to download the JavaScript runtime and provides detailed installation instructions for each platform.

A.2. Installation

To install the application, the following command shown in Listing A.1 needs to be run:

```
1 npm install
```

List. A.1: Installation

[1] https://nodejs.org/en

A.2.1. Running Tests

The tests can be run by using the following command visible in Listing A.2:

```
1 npm test
```

List. A.2: Running tests

A.3. Configuration

To communicate with Things from a Mozilla Things Gateway, the following environment variables need to be set:

- **MOZ_BASE**: URL of the Mozilla Gateway
- **MOZ_AUTH**: Mozilla Gateway bearer token

Until the Mozilla Gateway provides a way to generate access tokens for other applications, it is necessary to use the developer tools of a browser to get the bearer token. To do so, the network tools can be used to inspect a request sent to the web application of the Mozilla Gateway, where the authorization header can be found that includes the bearer token.

A.4. Usage

To start the application, the following command shown in Listing A.3 needs to be run:

```
1 npm start
```

List. A.3: Starting the application

The web interface (Swagger UI) can then be accessed on `http://localhost:5000`. After the installation, there are already example Things provided. However, it is necessary to have real devices available to really use the application.

To add devices, the Thing Descriptions must be placed in the correct folder, the devices are then automatically added at the start of the application:

- `public/td` for W3C Thing Descriptions
- `public/td-moz` for Mozilla Thing Descriptions

Things connected to a Mozilla Gateway can be added by getting the corresponding Thing Description by sending a GET request with a "Accept: application/json" header to the `/things` URL of the Gateway and copying the JSON Thing Description to the path above. The response received by the Gateway is an array, it is important to only put one Thing per file. Alternatively it is also possible to script a new Thing by using the Mozilla Things Framework[2].

[2] `https://iot.mozilla.org/things`

To connect things using a W3C Thing Description, it is possible to manually write a Thing Description for devices that provide a HTTP REST API. For the myStrom WiFi Switch[3], a working one is already provided - it is just necessary to adapt the URL to the correct IP address in the local network. Otherwise, it is also possible to script Things using the Node-WoT[4] library, however the Universal Explorer does only support the W3C Thing Description 1.0 JSON-LD serialization format.

[3] `https://mystrom.ch/wifi-switch-ch`
[4] `https://github.com/eclipse/thingweb.node-wot`

B
Repository of the Project

The Universal Explorer is open source and licensed under the permissive MIT license. The repository of the application, where it can be downloaded, can be found on GitHub: `https://github.com/linusschwab/wot-universal-explorer`

Besides the code of the application, the repository contains a readme with badges for the test status (provided by Travis CI[1]), the code coverage calculated by Codecov and the dependency status monitored by the David[2] dependency manager. Additionally the readme contains a copy of the installation manual from Appendix A and a quick introduction to the REST, WebSocket and JavaScript APIs. Figure B.1 shows a screenshot of the repository at its current state.

[1] `https://travis-ci.org/linusschwab/wot-universal-explorer`
[2] `https://david-dm.org/linusschwab/wot-universal-explorer`

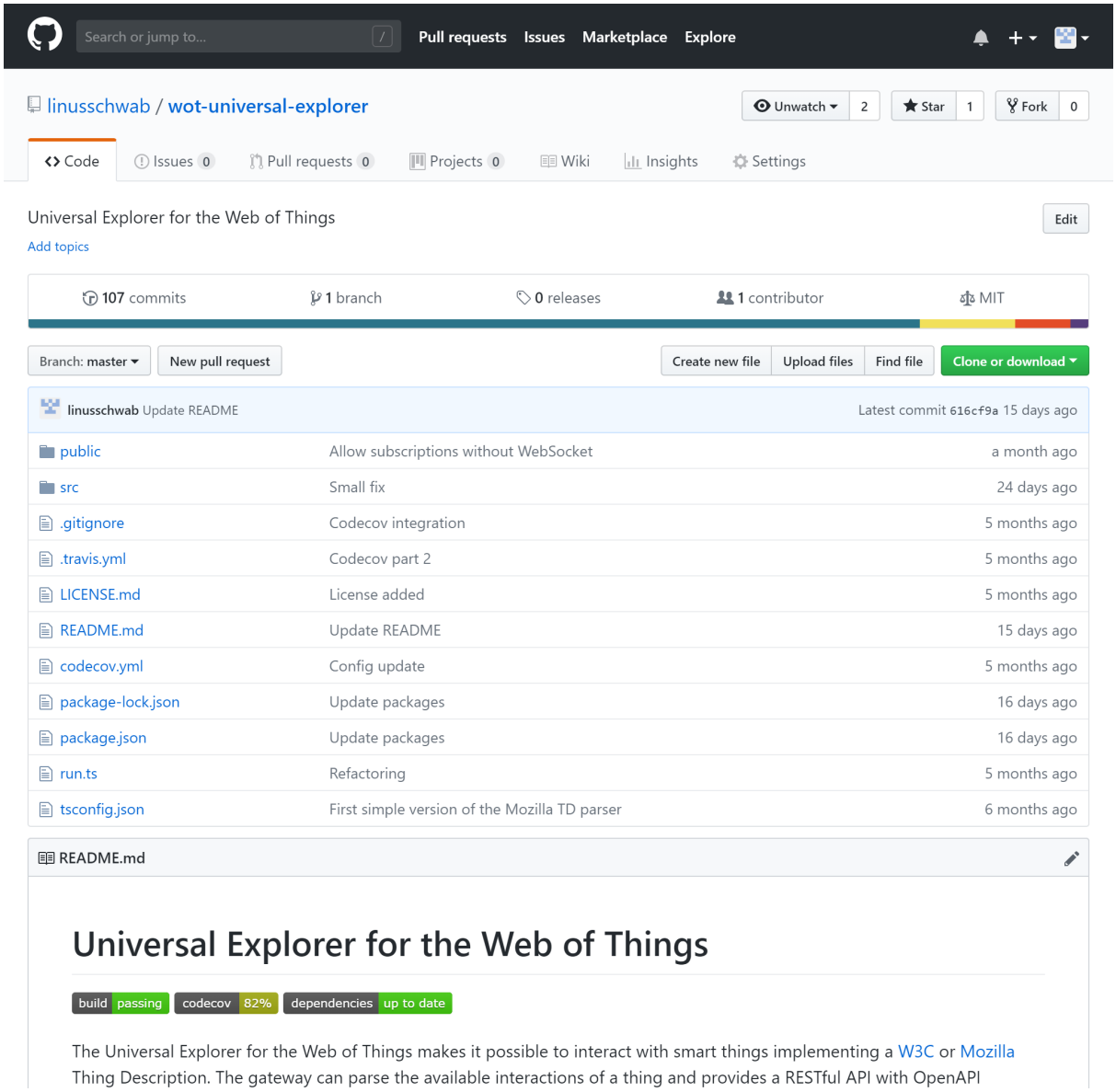

Fig. B.1.: Screenshot of the repository of the project

C
Common Acronyms

API	Application Programming Interface
CoAP	Constrained Application Protocol
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IoT	Internet of Things
IP	Internet Protocol
JSON	JavaScript Object Notation
JSON-LD	JavaScript Object Notation for Linked Data
MQTT	Message Queue Telemetry Transport
MVC	Model View Controller
REST	Representational State Transfer
TD	Thing Description
UI	User Interface
UML	Unified Modeling Language
URI	Unified Resource Identifier
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
WoT	Web of Things

References

[1] Ala Al-Fuqaha, Mohsen Guizani, Mehdi Mohammadi, Mohammed Aledhari, Moussa Ayyash. *Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications*. IEEE Communications Surveys & Tutorials (Volume 17, Issue 4), IEEE, 2015. 1

[2] Dominique Guinard, Vlad Trifa, Friedemann Mattern, Erik Wilde. *From the Internet of Things to the Web of Things: Resource Oriented Architecture and Best Practices*. Springer, 2011. 2

[3] Dave Raggett. *The Web of Things: Challenges and Opportunities*. Computer (Volume 48, Issue 5), IEEE, 2015. 2

[4] Dominique Guinard. *A Web of Things Application Architecture – Integrating the Real-World into the Web*. Doctoral dissertation, ETH Zurich, 2011. 1, 2

[5] Vlad Trifa. *Building Blocks for a Participatory Web of Things: Devices, Infrastructures, and Programming Frameworks*. Doctoral dissertation, ETH Zurich, 2011. 1, 2

[6] Dominique Guinard, Vlad Trifa. *Towards the Web of Things: Web Mashups for Embedded Devices*. WWW (International World Wide Web Conferences), Enterprise Mashups and Lightweight Composition on the Web (MEM 2009) Workshop, 2009. 2

[7] Dominique Guinard, Vlad Trifa, Erik Wilde. *A Resource Oriented Architecture for the Web of Things*. Internet of Things 2010 International Conference (IoT 2010), 2010. 2

[8] Vlad Trifa, Samuel Wieland, Dominique Guinard, Thomas Bonhert. *Design and implementation of a gateway for web-based interaction and management of embedded devices*. International Workshop on Sensor Network Engineering (IWSNE 09), 2009. 2

[9] Soma Bandyopadhyay, Abhijan Bhattacharyya. *Lightweight Internet protocols for web enablement of sensors using constrained gateway devices*. 2013 International Conference on Computing, Networking and Communications (ICNC), IEEE, 2013. 10

[10] Roy Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine, 2000. 17

[11] Bruno Costa, Paulo Pires, Flávia Delicato, Paulo Merson. *Evaluating a Representational State Transfer (REST) Architecture: What is the Impact of REST in My Architecture?*. 2014 IEEE/IFIP Conference on Software Architecture, IEEE, 2014. 17

[12] Stefan Tilkov, Steve Vinoski. *Node.js: Using JavaScript to Build High-Performance Network Programs.* IEEE Internet Computing (Volume 14, Issue 6), IEEE, 2010. 24

[13] Zheng Gao, Christian Bird, Earl Barr. *To Type or Not to Type: Quantifying Detectable Bugs in JavaScript.* 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), IEEE, 2017. 24

[14] Glenn Krasner, Stephen Pope. *A Cookbook for Using the ModeL-View-Controller User-Interface Paradigm in Smalltalk-80.* Journal of Object-Oriented Programming, SIGS Publication, 1988. 24

[15] Urs Hunkeler, Hong Linh Truong, Andy Stanford-Clark. *MQTT-S — A publish/subscribe protocol for Wireless Sensor Networks.* 3rd International Conference on Communication Systems Software and Middleware and Workshops, IEEE, 2008. 10

[16] Carsten Bormann, Angelo Castellani, Zach Shelby. *CoAP: An Application Protocol for Billions of Tiny Internet Nodes.* IEEE Internet Computing (Volume 16, Issue 2), IEEE, 2012. 10

[17] Matthias Kovatsch. *CoAP for the web of things.* 4th International Workshop on the Web of Things (WoT 2013), 2013. 10

[18] A.J. Bernheim Brush, Bongshin Lee, Ratul Mahajan, Sharad Agarwal, Stefan Saroiu, Colin Dixon. *Home Automation in the Wild: Challenges and Opportunities.* ACM Conference on Computer-Human Interaction, 2011. 1

Referenced Web Ressources

[19] Kazuo Kajimoto, Matthias Kovatsch, Uday Davuluru. *Web of Things (WoT) Architecture.* W3C Working Draft, 2017. https://www.w3.org/TR/2017/WD-wot-architecture-20170914/ 2, 12

[20] Sebastian Kaebisch, Takuki Kamiya. *Web of Things (WoT) Thing Description.* W3C Working Draft, 2018. https://www.w3.org/TR/2018/WD-wot-thing-description-20180405/ 6

[21] Michael Koster. *Web of Things (WoT) Protocol Binding Templates.* W3C Working Group Note, 2018. https://www.w3.org/TR/2018/NOTE-wot-binding-templates-20180405/ 10

[22] Zoltan Kis, Kazuaki Nimura, Daniel Peintner, Johannes Hund. *Web of Things (WoT) Scripting API.* W3C Working Draft, 2018. https://www.w3.org/TR/2018/WD-wot-scripting-api-20180405/ 12

[23] Ben Francis. *Web Thing API.* Mozilla, Unofficial Draft, 2018 (accessed 21. June 2018). https://iot.mozilla.org/wot/ 10, 13, 20

Index