An Experimental Evaluation of Software Frameworks for the Web-of-Things

Magnus Ødegård Bergersen

Master's thesis in Software Engineering

Department of Computer science, Electrical engineering and Mathematical sciences, Western Norway University of Applied Sciences

June 2021

Western Norway University of Applied Sciences

Abstract

In this thesis we evaluate the Web of Things technology. The Research-based Innovation scheme in Norway has started a project named Smart Ocean, which explores new technologies to develop ocean monitoring solutions. This is where Web of Things comes into the picture. As Internet of Things tends to results in fragmentation of applications, due to of re-use code, Web of Things has been suggested to provide more standardised solutions. We evaluate four different frameworks developed for the Web of Things. We use case studies provided from the Smart Ocean project to replicate a real case scenario where the frameworks are in use. As we create different prototypes for the Smart Ocean project, an approach is required to evaluate the frameworks, and then in turn the technology. Here we are using the ISO/IEC 25010:2011 standard, which is the standard for software quality requirements and evaluation. We use the software quality model to define evaluation criteria which can be used for this thesis. We evaluate based on maturity, learnability and security, as these are considered the most important aspects from the frameworks. We additionally perform evaluation based on criteria identified during the development process, including automation, documentation and amount of code. Based on the evaluation criteria, we access the state which the technology currently is in. By doing so, we find some interesting results about the frameworks, which also generalise for the technology itself. Our evaluation shows that all frameworks evaluated in this thesis had a few critical flaws, which would be enough to have devastating effects on the case studies. As the frameworks are recommended by the creators for the technology, this means there are some issues involving the matureness of the technology. We conclude in the end that the current state of the technology is too weak to be used for a large scale project like Smart Ocean. We conclude that with the present state of the technology, the Smart Ocean project would be better off using alternative Internet of Things solutions which has taken care of the flaws which the Internet of Things provides.

Acknowledgements

I want to thank my supervisor, Lars Michael Kristensen for providing me with the motivation and support needed to finish this thesis during the Covid-19 pandemic. Without his comments and feedback, this thesis would have been a mess since day one.

Contents

Chapter 1

Introduction

In the last decade, more and more devices have been connected to the Internet. Phones, sensors and even fridges are becoming what we call “smart” by being connected to Internet and have become a part of our daily lives. This means that almost every component we use in our daily lives has reached the world-wide-web. This collection of devices which has been named *the Internet-of-things* (IoT) and comes with a lot of possibilities when combined with cloud computing and machine learning. But “with great power comes great responsibilities” as famous book-writer Stan Lee wrote. The Internet of Things has its flaws with its commercial potential being held back by fragmentation. This is where the concept Web of Things (WoT) [33] comes in order to address this shortcoming by minimizing the needs of understanding multiple IoT technologies and communication protocols.

Having many domains using IoT daily, combined with the fact that IoT is evolving every day, it is prone to have multiple flaws. Investigating a better solution like WoT would be profitable for everyone within this field. Not only will it strengthen the systems developed today, but also support easier development of these systems.

While IoT focuses heavily on the network aspect of getting devices to communicate, WoT assumes that the devices are already connected and focuses on how to build applications. As network connectivity is a field of study itself which is normally separated from software development, creating a framework for software developers help the field of study by decreasing the need of knowledge in the field communication technology.

This thesis investigates the capabilities of the WoT concept, while providing a framework solution for Smart Ocean data services based on IoT and the emerging WoT paradigm.

1.1 Motivation

The motivation underlying this master thesis comes from the Smart Ocean research centre that HVL is partner in. As Smart Ccean deployment demands large amount of resources while at the same time being under resource constraints, creating a framework to simplify the development and deployment process will ease the software technology aspect of the Smart Ocean platform.

The Smart Ocean platform is being developed by the Smart Ocean centre for Research-based Innovation scheme (SFI) in Norway [32] with the goal of keeping Norway as one of the leading countries within technology and operations in the ocean. Not only will it increase Norways position, but also help the world with climate regulation, transportation, food supply, energy production, and life quality. To achieve this goal, the SFI Smart Ocean have planned to enabling autonomy by real-time high quality data, increase oceanic management models and system, creating more profitable ocean industry operations and fact-based ocean resource management alongside their research partners and user partners.

To understand how to create or even use such a framework, one first has to understand the concept of WoT [46]. Guinard defines web-of-things as follows:

“The Web of Things is a refinement of the Internet of Things by integrating smart things not only into the Internet (network), but into the Web Architecture (application)” [11].

This means decreasing the focus on how devices communicate with each other through low level protocols, which by itself would require a lot of maintenance. Using web protocols on the application layer in the OSI model (such as HTTP), as the primary communication protocol simplifies the integration for connecting devices to an application. The OSI model is the open system interconnection model which standardizes the communication protocols. This opens up the possibility for each device to indirectly communicate with each other over the web, either it being the World Wide Web or a local network. This process is done by using so-called ”thing descriptors”, which we will introduce further in this thesis, thing descriptions simplify the way a device is connected to the World Wide Web, making it easier to integrate new devices on the fly.

There already exist several frameworks supporting the WoT approach like the open-source Node-RED project which is a flow-based development tool for visual programming developed by IBM [24]; WebThings from Mozilla which provides an implementation of WoT with a working gateway and a framework for development [21]; Siemens Desigo CC which is a WoT platform from Siemens [7]; and Eclipse Thingweb node-wot which provides a WoT thing description, runtime system, and a scripting API from the Eclipse Foundation [8]. All these software technology solutions aim to provide a flow-based development tool for visual programming of IoT applications. But each framework comes with its pros and cons and must be custom-made for a solution best suited for Smart Ocean technology. For example, Node-RED has issues with tracking states of multiple devices, which would be critical to track sensors on the bottom of the ocean as its not the easiest place to reach and perform maintenance.

Smart Ocean technology is a hot topic as we are slowly moving more and more devices into our oceans. As 71 percent of the earth is covered by ocean and the National Ocean Service estimates that 80 percent is yet unexplored in 2019, this shows that there are plenty of possibilities for technology to reach our waters. [12]

1.2 Research Question and Expected Result

To determine how the WoT frameworks may have impact on the Smart Ocean project, having clear goals on what to achieve is important. In this section we discuss some of the questions related to evolving the web of things technology and how we will be able to achieve these goals.

1.2.1 Research Question

When one creates an application with “things”, the terminology IoT application and not a WoT application is normally used. A ”thing” is defined as follows:

“Devices which have unique identities and can perform remote sensing, actuating and monitoring capabilities” [5].

Even big cloud providers have started showing IoT applications attention in terms of giving the technology a significant amount of services to support such an application. But why has not WoT achieved the same amount of attention? Is it because of its maturity? Because of its complexity? Or is it just a technology too specific for most applications that will have too many flaws for the development? In this thesis, we aim to find the reasons and how useful a WoT framework potentially would be in the context of the Smart Ocean initiative.

This brings forth the research questions of this master thesis:

R1 - What frameworks and platforms currently exists for developing applications based on the Web of Things concept?

R2 - To what extend does the current frameworks and platforms conform to the W3C Web of Things standard [39] and how mature are they?

R3 - How can the sensors envisioned in the Smart Ocean case studies be represented using Web of Things concepts?

R4 - To what extent can current Web of Things frameworks be used to implement SmartOcean case studies?

1.2.2 Expected Result

To address the research questions stated above, this thesis will investigate deeply into the WoT frameworks. By using existing frameworks, showcasing possible flaws early on in the development and implementing different pilot demonstrators for the Smart Ocean initiative.

This brings forth the expected results for this master thesis:

- Create prototypes based on Web of Things frameworks.

- Investigating the Web of Things technology for potential flaws.
- Showcase where the Web of Things technology can save time and resources and reflecting on when one should be using such a framework.

1.3 Research Method

In this thesis, we will create two prototypes with different frameworks for the technology, as mentioned in the expected results. This will test the limits of the technology, and thereby see if any flaws are identified. We will also get the experience needed during the development process to see where the technology could save time, or would lead to an increase in development time. As the prototypes created during this thesis involves multiple frameworks which supports the technology, this means we will get a wide coverage on the technology itself. This will be crucial for investigating if Smart Ocean should spend time on technology or not during their development. And if they decide to go for the technology, then it showcases how the technology can be used for their development. We will be able to determine what Smart Ocean has to do to get the frameworks to be used and if any additional development will be required.

To find the answers to our research question, we first have to figure out a way to evaluate the technology. As we will establish proof of concepts for the technology where few have done something similar leads to lack of any benchmarks to compare with. This means we have to find other ways of undertaking the evaluation. To help us get results in this process, we will be using Browns [6] technology delta evaluation framework as a basis for the evaluation.

In figure 1.1 we can see the framework. It consists of three phases, the descriptive modeling phase, experiment design phase and experimental evaluation phase.

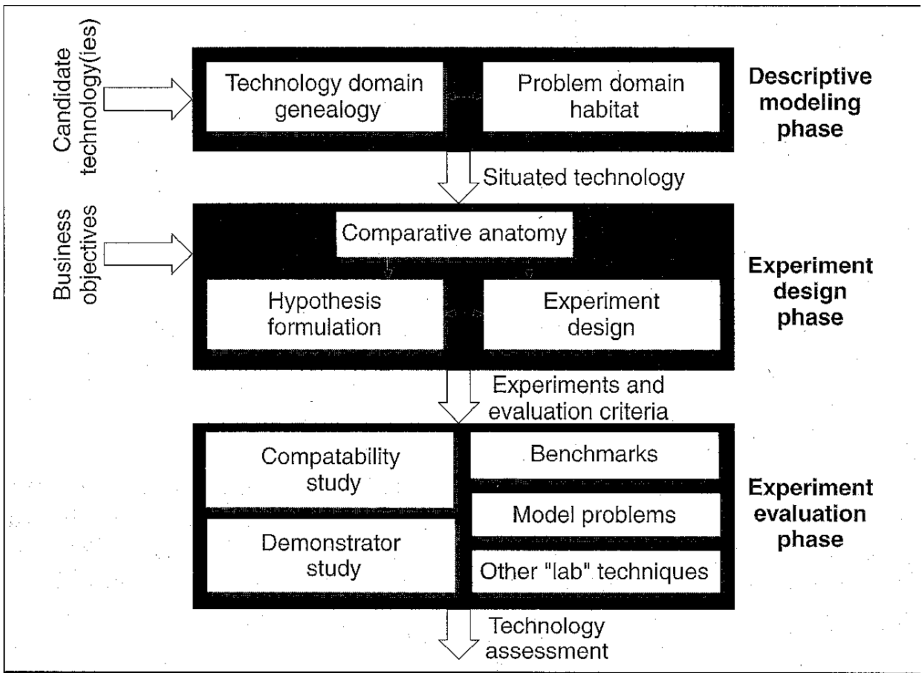


Figure 1.1: Technology Delta Evaluation Framework [6]

The descriptive modeling phase is about discovering the impact of new technologies, and how they improve from their technological ancestor. In this case, the ancestor is the IoT technology. This will be the frameworks for the WoT technology, which we will look at in this thesis.

The second phase is the experiment design phase. This phase is essentially the planning phase of the experimental evaluation. Here we create hypotheses on how to determine the answers to the questions from the descriptive phase. In this thesis, this means how we will be creating prototypes to answer our research questions presented in section 1.2.1.

The last phase is the experimental evaluation phase. This phase is about conducting evaluation on the hypotheses made in the experiment phase to confirm or refute the hypotheses. In our case, this means evaluating the prototypes implemented using the WoT frameworks. We decide to evaluate the WoT frameworks with a software quality analysis. This means we will be analysing the prototypes against criteria for software quality analysis. As the prototypes involve multiple frameworks, we will be evaluating each separately.

The criteria we will use to get our answers are as follows:

- **Maturity** - Evaluating how mature the frameworks are.
- **Documentation** - Here we will be evaluating how well defined the documentation for the frameworks are.
- **Amount of code** - Evaluating how much code is needed to get the frameworks to implement the case studies.

- **Learnability** - Here we will determine how much effort is required to develop with the different frameworks.
- **Automatising** - How much automatisation the frameworks provides for the developer.
- **Security** - How secure the frameworks are, and what vulnerabilities they may have.

1.4 Outline

The rest of this thesis is organised as follows:

Chapter 2: Background covers the technology used in this thesis. We provide details on the technologies and explain how they work.

Chapter 3: Web of Things Frameworks covers the frameworks used to implement the WoT technology.

Chapter 4: Smart Ocean Evaluation Case Studies covers the case studies provided from the Smart Ocean. Here we evaluate how we shall be implementing these, and how it will look like using the Web of Things standard.

Chapter 5: Prototype Design covers the design of the prototypes created for the Smart Ocean Case Studies. This chapter goes through the design of the prototypes created and how they work.

Chapter 6: Implementation and Deployment goes through the implementation of the prototypes created in this thesis, as well as how to deploy them. Here we provide details on how the prototypes were made, and how one could implement and deploy the prototypes themselves.

Chapter 7: Evaluation is where we evaluate the frameworks considered in this thesis. We will be using the ISO standard for evaluating software quality to evaluate the frameworks. By doing so, we can access what state the WoT technology is.

Chapter 8: Conclusion and Future Work is where we conclude based on the information gathered from the evaluation chapter. We will also be covering what one could do in the future with the WoT technology and related work.

Chapter 2

Background

WoT comes with several associated technologies which needs to be clarified before presenting the technology as a whole. This chapter covers the most common and important background technologies for WoT to work.

2.1 The World Wide Web

To understand how most of the technology used in this thesis works, one first has to understand how devices on the World Wide Web works. In this section, we cover the basics of the communication protocols used on the world wide web and what request/response methods are used. This section also covers some of the benefits which comes with developing software in the context of the World Wide Web. This will be essential for the technologies considered in this thesis.

2.1.1 OSI Model - Application Layer

The Open System Interconnection (OSI) model consists of seven different conceptual layers for interconnection [3]. The very top layer is the application layer which is the one in focus in this master thesis. This layer consists of protocols used for communication between applications running on devices. This layer supports a device being identified, authenticated, its state, the integrity, privacy, syntax rules and data presented for the users in a readable manner. This constitute an important aspect for the communication between devices without focusing on other aspects, such as, IP addresses, direct communication methods or how the device will be connected to the World Wide Web/Internet.

As mentioned, the application layer has different protocols to ensure the policies. The most common one are:

- Hypertext Transfer Protocol (HTTP) – For accessing data on the world wide web
- Domain Name System (DNS) – For IP identification
- File Transfer Protocol (FTP) – For transferring files

- Simple Mail Transfer Protocol (SMTP) – For mails

The most important protocol for this thesis will be HTTP, as it is the most widely used communication protocol in the frameworks considered.

2.1.2 Hypertext Transfer Protocol – HTTP

The Hypertext Transfer Protocol, also known as HTTP, is one of the most used communication protocols located at the application layer. It is also the protocol which defines the communication over the World Wide Web. HTTP is a stateless protocol, which means that no session data/information is stored between each interaction. As the WoT technology moves communication towards the application layer of the OSI model, this means that it will be the most essential protocol to understand. The protocol involves two devices, one client and one server.

The client is responsible for initiating the communication. It can either be a web browser requesting a web page or a software component which wants to access information from a web Application Programming Interface (API). An API is an interface which defines how the interaction between components is to be handled.

The server on the other hand is responsible for handling the requests from the clients. As the HTTP protocol is stateless this means that the server handles the requests based on the information received. This means authentication, information or data needs to be included in the requests for the server to give the appropriate response.

For the server to understand the request from the user, there are a few things one has to address first. One has to follow the syntax of the protocol and the servers API. This is needed for the server to return understandable information to the client. The typical HTTP request message involves a request method, request URI or host along with a path, the HTTP version, request headers and an optional message body.

The request method defines the action one desires to do on a resource at the server. There exist eight different methods:

- GET – Retrieve data from a server
- HEAD – Asks for the response headers without the body
- POST – Create resource on a server
- PUT – Update an existing resource on the server
- DELETE – Delete a specified resource
- CONNECT – Establish a tunnel with the server
- OPTIONS (HTTP Version 1.1) – Retrieve the communication options from the server
- TRACE – Message loop-back testing

The request header is additional information one can send to the server, alongside the request method and the message body. This can be additional information like tokens for authorization, user agent of the browser and an upgrade request which upgrades the protocol over the initial HTTP protocol.

Uniform Resource Identifier - URI

A Uniform Resource Identifier (URI) is a sequence of characters which identifies a resource. This is the barebone of the identification of the resources located on the World Wide Wb. Commonly, an URI is composed of a Uniform Resource Locator (URL) and a Uniform Resource Name (URN).

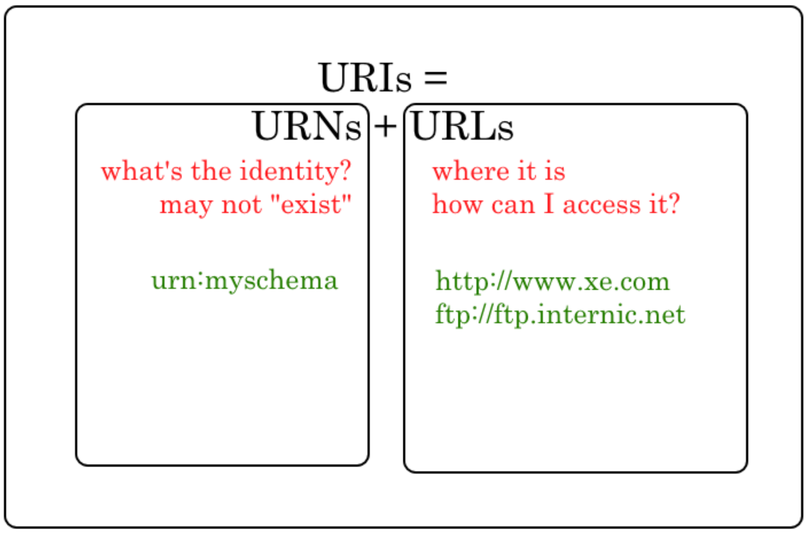


Figure 2.1: Explaination of the URI [37]

The URL is an identifier together with a desired protocol to access resources over a network, which can be seen in figure 2.1. The URN is used together with the URL to locate where on the server the specific resource resides.

WebSockets

WebSocket [35] is a protocol used for creating a persistent bi-directional communication channel between a server and a client. This can be used instead of HTTP communication to decrease the amount of request/response messages. To create a WebSocket between a client and server, one first has to request such a communication. By using the Upgrade header in the request as mentioned in the HTTP section, one can asks the server if it supports WebSocket communication. Once the connection is established, one can stream messages in a bi-directional manner. The connection can either be secure or non-secure by specifying this in the request header.

2.1.3 Internet of Things

Bahga and Midsetti [5] defines Internet of Things as follows;

> "A dynamic global network infrastructure with self-configuring capabilities based on standard and interoperable communication protocols where physical and virtual "things" have identities, physical attributes, and virtual personalities and use intelligent interfaces, and are seamlessly integrated into the information network, often communicate data associated with users and their environments."

In other words, creating systems on the World Wide Web, that are normally not connected to the Internet. By using multiple devices to provide data in large numbers helps us create advanced applications with more functionalities. With the support of devices, we can more efficiently automatise applications, as they can be set up to do functionality based on conditions.

IoT is also commonly used alongside cloud services. As sensors will eventually provide too much data to be stored. Saving it on the cloud where the storage is unlimited would be the most optimal way to create way to create IoT applications.

2.1.4 HyperText Markup Language - HTML

The HyperText Markup Language (HTML) is the markup language for showcasing information in a web browser. It is defined by Mozilla as the most common building block for the World Wide Web [14]. In this way one can easily showcase documents and information as a server for any client which requires this information. It provides a user readable way to understand information sent over the web without having a huge understanding of code. It is most commonly sent over the HTTP protocol, or its secure version HTTPS but can also be sent over other sources if needed.

2.1.5 JavaScript Object Notation - JSON

JavaScript Object Notation (JSON) is defined by Mozilla as a standard text-based format for representing structured data based on JavaScript objects [47]. Instead of providing a readable markup language like HTML, one can use JSON to provide information in a format more suitable for an application to use. This decreases the time one has to use on retrieving information from a server and provides fully functional objects in an instant to be used in programming.

2.1.6 Representational state transfer - REST

Representational state transfer (REST) is an architectural style for creating web services over the Internet. Alongside this architectural style comes six constraints defined by Roy Feilding which needs to be followed for an interface to be called RESTful [30]. The constraints are as follows:

Client-Server

Separating the client from the server. This means moving the business logic away from the client to the server and forcing the client to work with the server to retrieve data.

Statelessness

Each request from the client is a new request. This means that the server never keeps track of the state from the last session and needs information alongside the request. This in turn means that the state is tracked by the client.

Cacheability

The possibility for the user to cache data from an earlier session makes the service a lot more robust and scalable. But not all data can be cached, and one has to specify which data can be cached or not on the server side.

Layered System

As the client has no knowledge of what is going on internally on the server, this means that one can hide different business logics like databases, external services or internal services behind the server as layers without the client needing to know.

Uniform interface

Uniform interface means defining a uniform interface between the client and server which both can use. This means that the server can easily read the data from the client and the other way around.

Code on demand

This one is optional, but REST services offer the possibility to download code and execute them on the client as applets or JavaScript.

2.1.7 Multicast DNS - mDNS

Multicast DNS is a technology created to locate devices on the local network. The technology was created by Apple [17] in 2013, and is normally used alongside IoT applications on a local network. The way the protocol works, is that it sends an IP multicast query message to the local network, asking for the devices on the network to identify themselves. After a device has retrieved the query message, it then sends back its IP address along a message. This way we can automatically find devices on the local network.

2.2 Web of Things

Guinard and Trifa [11] defines the Web of Things as a specialization of the Internet of Things which focuses on moving the communication between the things towards the application layer of the OSI model. This means that the framework will reduce the work required for a developer to integrate multiple devices into the same application, by moving the communication directly into the application. Providing an URI for describing each device and its functionality makes applications a lot easier to implement, as one no longer has to spend time getting each device with its unique communication protocol to integrate into a system. Each device may have its own reasons to choose a specific protocols, such as signal range, network connection, and data usage.

The World Wide Web Consortium (W3C) has defined the architecture for the Web of Things technology [39]. It is based around four fundamental building blocks which will be introduced in the following subsections. W3C is an international community where people work together to develop web standards [2].

2.2.1 Thing Description

For an application to retrieve and communicate with new unknown devices, it makes use of a things descriptor. This is done by retrieving the thing descriptor from a unique URI which defines a thing. The W3C defines a things description as having four main components; textual metadata, interaction affordances, schemas, and web links [43].

Within the WoT technology stack there exists a predefined vocabulary for the metadata. This covers everything from the data schema the thing uses, to security measurements the thing uses, and the external links exposed by a "thing". Using a Thing Descriptor Processor to process the metadata retrieved and deserialize the thing description, opens up for the application to directly interoperate with a thing. The metadata deserialized from the thing descriptor can be used as objects which can be manipulated and read at any time in an application. We call this process to consume a thing description.

```
{
    "@context": "https://www.w3.org/2019/wot/td/v1",
    "id": "urn:dev:ops:32473-WoTLamp-1234",
    "title": "MyLampThing",
    "securityDefinitions": {
        "basic_sc": {"scheme": "basic", "in":"header"}
    },
    "security": ["basic_sc"],
    "properties": {
        "status" : {
            "type": "string",
            "forms": [{"href": "https://mylamp.example.com/status"
                }]
        }
    },
    "actions": {
        "toggle" : {
            "forms": [{"href": "https://mylamp.example.com/toggle"
                }]
        }
    },
    "events":{
        "overheating":{
            "data": {"type": "string"},
            "forms": [{
                "href": "https://mylamp.example.com/oh",
                "subprotocol": "longpoll"
            }]
        }
    }
}
```

Listing 2.1: Thing Description Example from W3C [43]

In listing 2.1, we can see an example of a thing description. It has an ID to define the thing, a title for the thing and a scheme identifying which version of the W3C it uses. It also has the interaction affordances listed as properties, actions and events. The interaction affordances has types to explain which type of data it works with, and a link to its endpoint to where we can access the data. It also provides the security on how to access the thing descriptor, which we will be covering in section 2.2.4.

The interaction affordances work the same way as metadata by using a Thing Descriptor Processor, but focuses on providing functionality for the user to interact with the thing. This can be everything from properties which gives the user the option to manipulate the data, actions to provide remote execution of functions, and events which triggers different actions when received.

The data schemas of the thing description are there to provide the user an affordance for the development. It provides different metadata which we can add to our affordance. It also conforms to the standard for the thing descriptions and defines the syntax which the thing description has to follow. The affordance provided can then be deserialized into data objects to provide the user a guideline on what actions can be performed on the thing.

The web links of the thing is at it sounds; links exposed by the thing which is used to define its relation between other associated things. This can be anything from a lamp which has a defined link to its switch.

2.2.2 Binding Templates

The second building block which W3C defines in the architecture of WoT is the binding template [40]. This is essential to have in order to get the thing descriptor to deserialize data in a way that the frameworks will understand. As mentioned earlier, each thing tends to have its own protocol for communication. WoT focuses on simplifying this by using binding template as an adapter between each thing.

Figure 2.2 illustrates how the binding template works. The example is of a read property with a simple HTTP binding. The WoT consumer sends a read-property operation from the consumed thing, which then gets translated into a HTTP request. The exposed thing has HTTP template configured to retrieve HTTP requests, then translates the data retrieved from the sensor and sends it back with the same binding as received. Its important to mention that we can have multiple binding templates to the same property, which translates into the necessary protocol on demand.

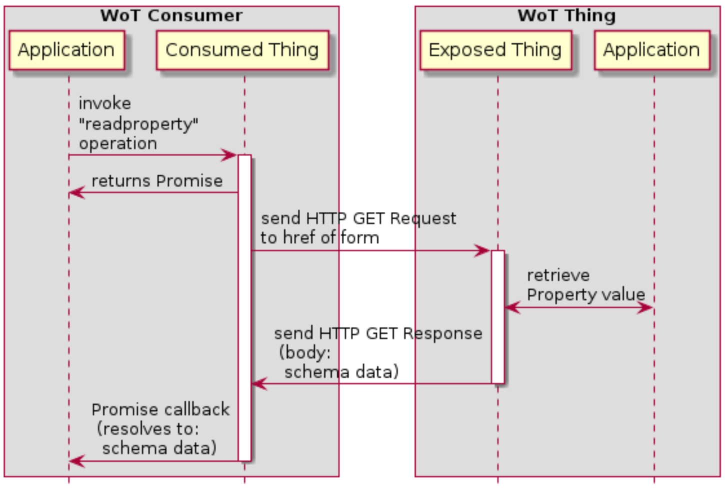

WoT Consumer
WoT Thing
Application
Consumed Thing
Exposed Thing
Application
invoke "readproperty" operation
returns Promise
send HTTP GET Request to href of form
retrieve Property value
send HTTP GET Response (body: schema data)
Promise callback (resolves to: schema data)

Figure 2.2: Binding Template Example from W3C [40]

A binding template is closely related to the binding instances from the thing descriptions and the binding implementation from the protocol bindings. The binding implementation for the protocol binding is the way the runtime implements the binding protocols like HTTP, Websockets, MQTT or CoAP. By using a binding template, the thing descriptor can understand the data being retrieved and sent. One can say it act as a translator for the different devices.

2.2.3 Scripting APIs

The third building block which W3C defines is the scripting API [41]. This part is not crucial for the WoT frameworks but is highly recommended to implement. Standardizing the API for the things to use decreases the effort needed to integrate them into an application.

There exist three use cases for the functionality provided by scripting API's. The first case is how to consume a thing. This functionality provides an easy way to create classes which defines the physical things. From setters/getters for the properties, to actions and events as defined in section 2.2.1.

The second case is about exposing the thing to the World Wide Web. This include functionality like removing or adding different definitions of the thing to the API, and adding a new property for a specific device. For example, adding a new property like serial number to a product with QR code sold from a supermarket. The third and last use case is discovery. This case resolves around locating other things from the thing being used. This can either be through Bluetooth, local LAN or semantic queries. The process of discovering devices can either be executed with a specified timed interval or started and stopped manually. By doing this, one can for example add a new thing to a local network which an existing thing can look for.

2.2.4 Security and Privacy Guidelines

The fourth and last building block which W3C defines is the security and privacy guidelines [42]. This building block, as the name implies, evolves around providing the security information for the thing descriptor to work and describes what specific security measures the thing has. This can be anything from authentication mechanisms to security measurements done through the protocol used. For the thing descriptor to be available to work, the thing needs this information to communicate with the descriptor. As each thing on the Internet provides a new surface point for attackers to reach, this means that the use of advanced mechanisms to secure the endpoints of each thing is essential. This does not mean that WoT introduces any new security mechanisms, but instead provides support for the most common security mechanisms. This can range from simple secure shell mechanisms to end-to-end encryption like HTTPS.

Chapter 3

Web of Things Frameworks

In this thesis, we investigate the usage of existing frameworks, in order to assess their applicability for Smart Ocean applications. Thoroughly going into them, finding their pros and cons, gives the possibility to investigate how they would work as part of Smart Ocean applications. Some of the frameworks will already at this stage be excluded for further investigation, if they either are not usable, outdated, or lack what is needed to be usable in a wide scale application based on sensors deployed in the ocean. All the frameworks have been selected based on the recommendations from W3C [33], as they are the main contributors of the development of WoT.

3.1 The Node-RED Project

Node-RED created by IBM is a flow-based development tool which is currently part of the JavaScript Foundation [1]. Node-RED being based on flow-based programming means that it focuses on creating blackboxes around the devices constituting the system. This means that one only has to focus on what is input to the thing and what data it then provides. Node-RED also simplifies the dataflow between each thing by creating a visual representation for a user for an easier understanding of the system.

Node-RED being a flow-based development tool does not make it directly conforming to the definition of a WoT framework. The primary reason for this is that it lacks a thing description which translates the data from the things. However, the Node-RED project has done measures to provide this functionality. The Node-RED-Nodegen tool can mimic this by creating the things in the Node-RED system based on data provided from an OpenAPI (swagger) document. This helps the developer with the implementation of new nodes and can act as the thing descriptor adapter needed for a fully functional web of things framework [25].

One limitation with the Node-RED-Nodegen framework is that the functionality of connecting devices to the framework is currently under development. The development of this has been going on since December 2019. This makes the extension unusable for the current project for adding the thing descriptors and devices to the Node-RED framework.

Node-RED makes it easy to use customized add-ons created by external developers. This means that if the framework lacks any functionality which would be needed for a solution, one can simply go to the add-on library of the Node-RED framework and download the extension [18].

Four external developers have created an extension to provide reading functionality of exposed thing descriptors. The extension is called node-red-contrib-wot [10], and provides the core consuming functionalities to use the Node-RED framework as a way to connect devices to the framework. It provides four Node-RED nodes to read properties from a thing descriptor, write properties to a thing descriptor, invoke actions and subscribe to a things events. To use the framework, one must use an existing exposed thing descriptor of a device and then the framework can be used by reading thing descriptor provided. By doing this, we can use the framework for consuming thing descriptors which we are to create in this thesis and use the flow-based development tools that Node-RED provides.

Figure 3.1 shows the different nodes that Node-RED by itself provides. A Node-RED project is comprised of multiple nodes which provides their own functionality. This can be anything from customized functions to a simple switch case. By doing this, one can make advanced flow based applications customized to ones needs. In this thesis, we mainly focus on the usage of the extension node-red-contrib-wot, with the addition of the debug node and the WebSocket node. The debug node is a simple way to get console information from a node.

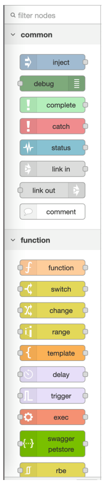
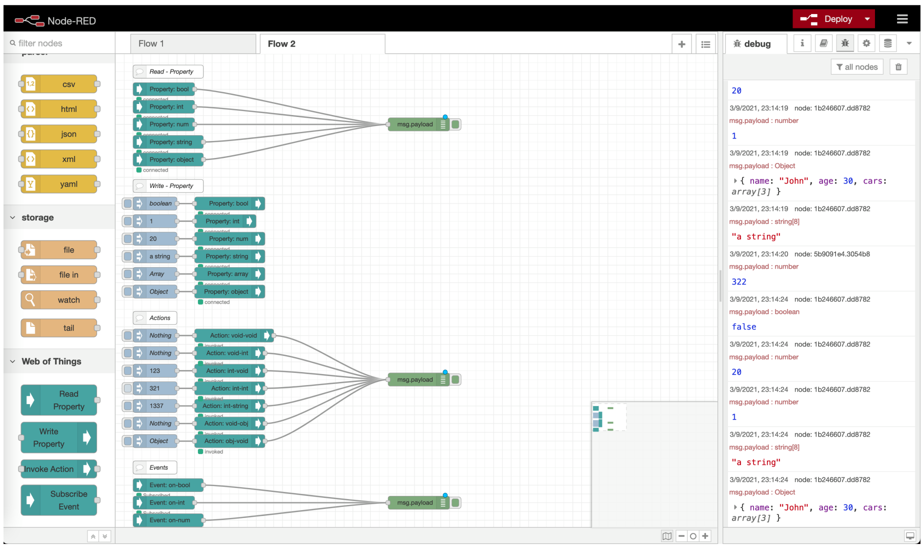


Figure 3.1: Example nodes for Node-RED

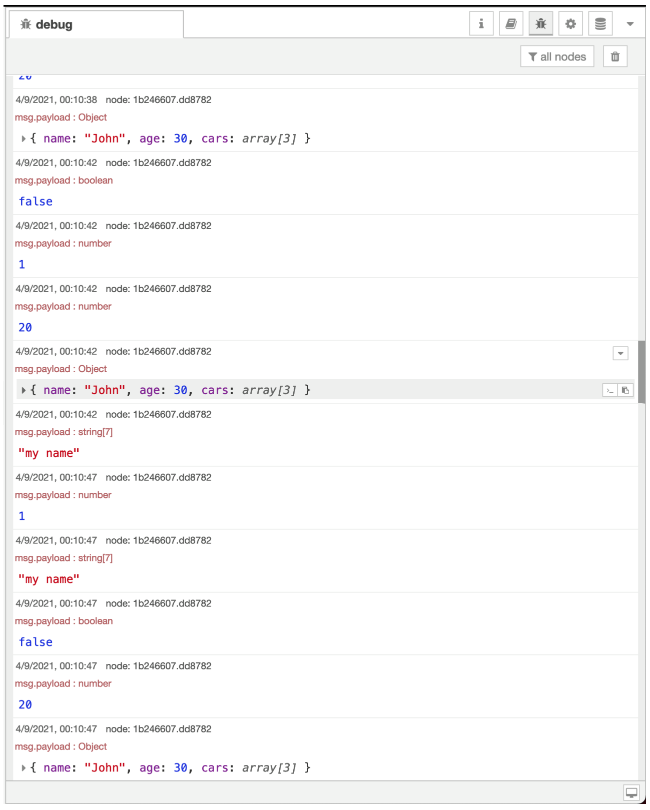
In figure 3.2 we can see a technical example on how to use Node-RED-Nodegen. By using an example thing description obtained via Eclipse ThingWeb; `http://plugfest.thingweb.io:8083/testthing`, we can showcase how this will work in the solution to come for Smart Ocean applications.



```
20
3/9/2021, 23:14:19   node: 1b246607.dd8782
msg.payload : number
1
3/9/2021, 23:14:19   node: 1b246607.dd8782
msg.payload : Object
▸{ name: "John", age: 30, cars:
array[3] }
3/9/2021, 23:14:19   node: 1b246607.dd8782
msg.payload : string[8]
"a string"
3/9/2021, 23:14:20   node: 5b9091e4.3054b8
msg.payload : number
322
3/9/2021, 23:14:24   node: 1b246607.dd8782
msg.payload : boolean
false
3/9/2021, 23:14:24   node: 1b246607.dd8782
msg.payload : number
20
3/9/2021, 23:14:24   node: 1b246607.dd8782
msg.payload : number
1
3/9/2021, 23:14:24   node: 1b246607.dd8782
msg.payload : string[8]
"a string"
3/9/2021, 23:14:24   node: 1b246607.dd8782
msg.payload : Object
▸{ name: "John", age: 30, cars:
array[3] }
```

Figure 3.2: Technical example of Node-RED with Node-RED-Nodegen

On the right side of the figure 3.2, we see the debug monitor gathering information from the nodes connected to the test thing, which is scaled up in figure 3.3. As we can see in the debug log, the nodes from node-red-contrib-wot fetches the sensor data from the things descriptor every third second. We can see node-red easily keeps track of the data provided from the test thing and works as intended. This means that one can use the nodes to fetch the data and manipulate it to ones needs. This can either be saving it in a database or doing processing functions on the data provided.



```
4/9/2021, 00:10:38   node: 1b246607.dd8782
msg.payload : Object
▸{ name: "John", age: 30, cars: array[3] }
4/9/2021, 00:10:42   node: 1b246607.dd8782
msg.payload : boolean
false
4/9/2021, 00:10:42   node: 1b246607.dd8782
msg.payload : number
1
4/9/2021, 00:10:42   node: 1b246607.dd8782
msg.payload : number
20
4/9/2021, 00:10:42   node: 1b246607.dd8782
msg.payload : Object
▸{ name: "John", age: 30, cars: array[3] }
4/9/2021, 00:10:42   node: 1b246607.dd8782
msg.payload : string[7]
"my name"
4/9/2021, 00:10:47   node: 1b246607.dd8782
msg.payload : number
1
4/9/2021, 00:10:47   node: 1b246607.dd8782
msg.payload : string[7]
"my name"
4/9/2021, 00:10:47   node: 1b246607.dd8782
msg.payload : boolean
false
4/9/2021, 00:10:47   node: 1b246607.dd8782
msg.payload : number
20
4/9/2021, 00:10:47   node: 1b246607.dd8782
msg.payload : Object
▸{ name: "John", age: 30, cars: array[3] }
```

Figure 3.3: Debug log from Node-RED

In the middle of figure 3.2, we can see the interaction affordance on the test thing. This is how we manipulate and fetch data from the thing. It can be anything from properties to actions and events. The affordances in this example is all nodes connected to the debug screen. To understand how everything works, we first has to take a deeper look at the affordances in this example.

In the figure 3.4, we see the interaction affordance property. Here we can see how the write properties function, by using injection nodes to inject messages to the test things thing descriptor. It then uses the desired protocol to communicate with the device, which can be anything from HTTP to WebSockets. Once the URL of the produced thing descriptor is added to the properties, it then automatically fetches the thing descriptor and provides the functionality that the thing provides. Reading from the properties is done in a similar fashion. Instead of injecting the nodes with data, we just connect a read property node to the desired thing descriptor and then do the desired action with the data retrieved.

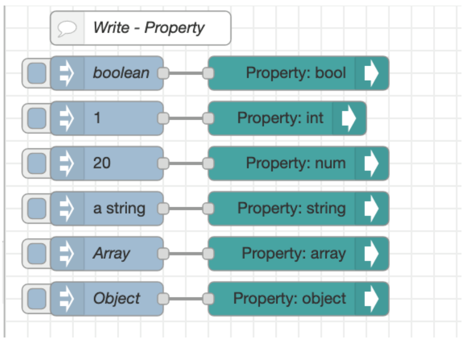


Figure 3.4: Properties in Node-RED

In figure 3.5 we can see how events are used for Node-RED. The way it works is by looking for any events in the consumed thing descriptor provided to Node-RED. Afterwards, the framework will by itself figure out which protocols are used to provide the information from the sensors and ask the user which protocol should be used in your application. The events then subscribes to the sensor and awaits a response. The framework takes care of checking if the connection is up, and warns the user if the framework would loose connection to the thing.

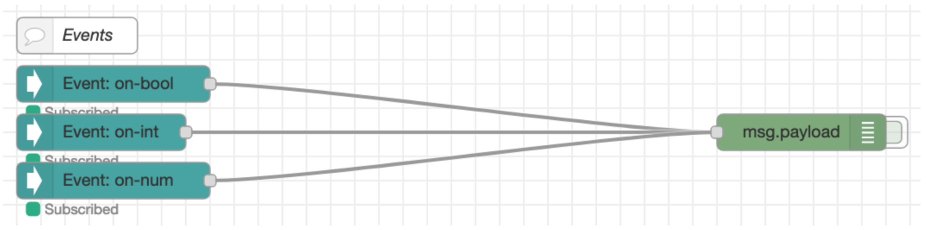


Figure 3.5: Events in Node-RED

Both the events and properties are instances of the node-red-contrib-wot which showcases how we can make Node-RED compatible with WoT technology. Every node created from this extension is an individual endpoint on the provided thing descriptor. The extension also provides the functionality of doing actions with the things provided. This is the same as writing to the property of a thing, but instead focuses on interacting with specific functions provided from the device.

3.2 WebThings from Mozilla

WebThings is an open source software distribution made by Mozilla which provides a gateway implementation and framework for implementing the WoT technology. The framework is a collection of reusable software components which is there to provide a thing API. The thing API provided is supposed to be a representation of the first and second building block within WoT, i.e, the scripting API and thing descriptors. The gateway provided by WebThings is software for gateways on a network which allows any user to monitor and control the things over the Internet [22].

The primary issue with this framework is that Mozilla has decided to implement their own thing descriptor schema to be used for the application. This implies that the solutions created from this framework is not compatible with any of the solutions which follows the schema from W3C. It follows the same rules set as W3C with properties, events and actions, but is created in a simpler way to make it easier for a new developer to pick up. This means that creating solutions within this framework requires the gateway to be used for consuming the things exposed by the framework they provide. Mozilla has given the developers open source access to the framework with support for multiple programming languages such as; JavaScript, Python, Java, Rust, C/C++ and MicroPython.

The second issue with this framework is the fact that it is targeting the domain of smart houses and not Smart Ocean. While the gateway provides a fair sense of customization, it may still not be enough for adapting it to wide scale smart ocean application. One would need to rewrite the gateway for custom solutions. This would be possible as the gateway is open-source. But it would also mean one would need to do this on an individual basis, as Mozilla does not provide any support or documentation on how to make changes and manipulate the gateway.

In figure 3.6 we can see the gateway in use, where we can observe the information gathered from sensors. This gateway can both be accessed from a local network and the World Wide Web, as Mozilla provides an internal Amazon Web Service (AWS) server for hosting the gateway service. The gateway can be used alongside a map of the house, to create a virtual map on where the sensors may be located. Alongside this, one can implement custom ruleset for the sensors to do different actions when needed.

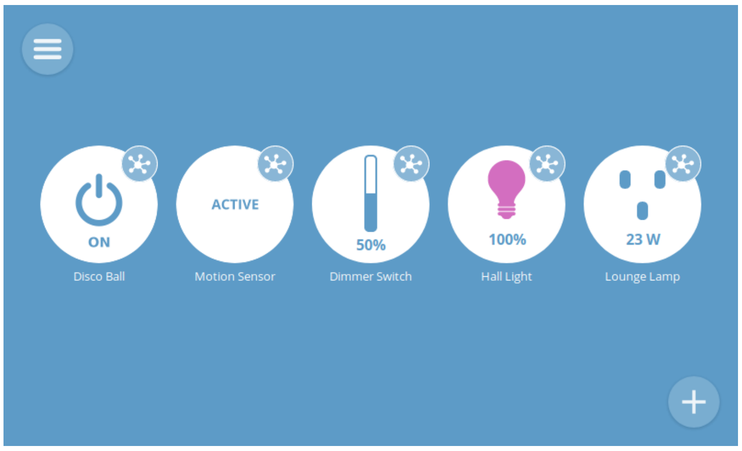


Figure 3.6: Mozilla WebThings Gateway [44]

Once the sensor is on the same local network as the gateway, then the gateway will start using mDNS to locate the sensor automatically. This is done by the gateway sending out query messages on the network, asking the sensors to identify themselves. Then it will give the user the option to add the device to the gateway, or exclude it. If the device is added, then the user will get the options provided in figure 3.7. The sensor in this figure is a test device, with the options to edit its properties. This showcases how one can use the gateway to both read data and write data to a sensor.

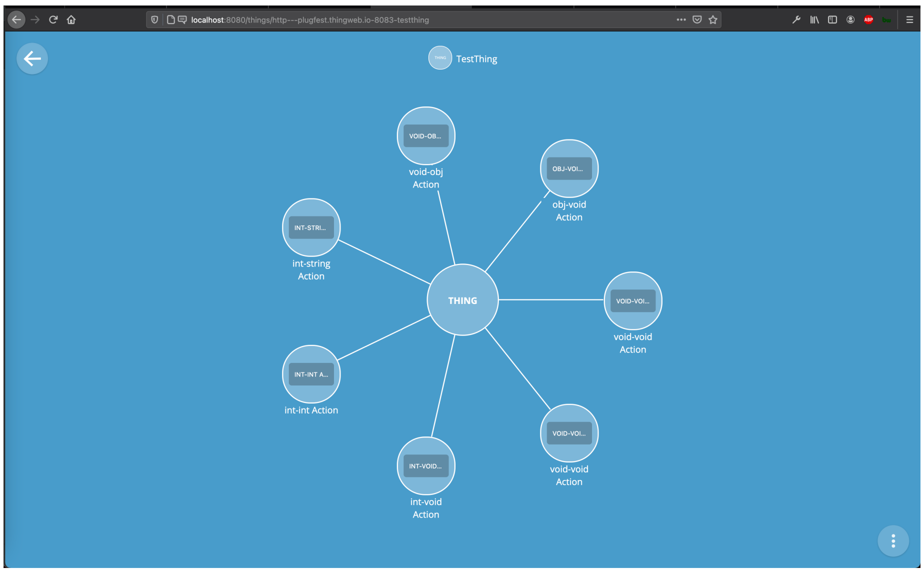


Figure 3.7: Mozilla WebThings Gateway TestThing

Once the sensors are up and running, then it will provide a built-in logging system to monitor the data over time. This means we can use the gateway logging system as a database to both monitor and validate the data provided from the sensors over time. Figure 3.8 shows how the information is provided and how we are able to pick which data we need to monitor.

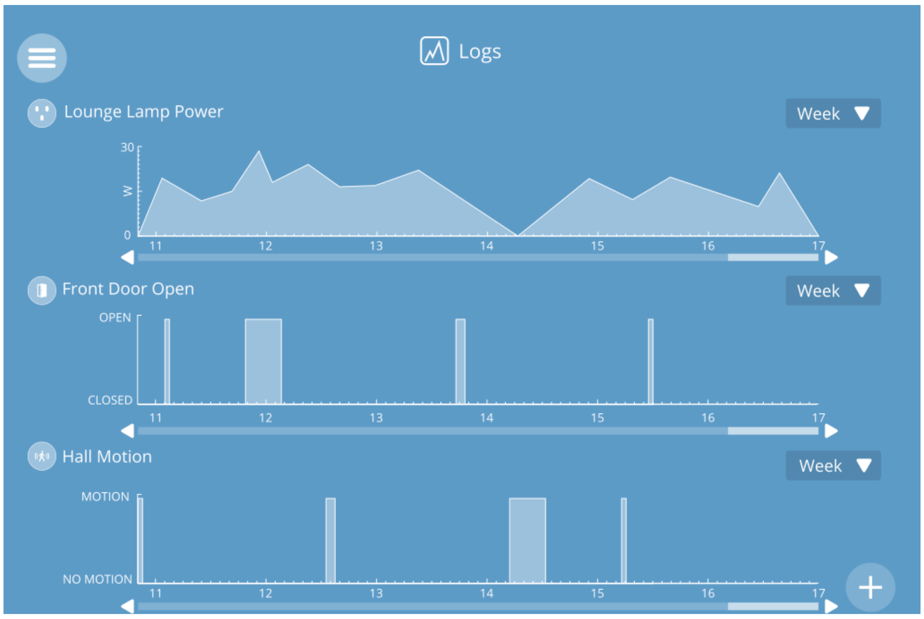


Figure 3.8: Mozilla WebThings Logging [15]

The gateway also provides the functionality to map where the different sensors are located which is shown in figure 3.9. This map can be anything from a house to an ocean as long as its a scalable vector graphics (svg) file. This will be handy for a project like Smart Ocean, to understand where each sensor is located in the ocean. Once the map is uploaded, one can click and drag the sensors in the system to the desired location.

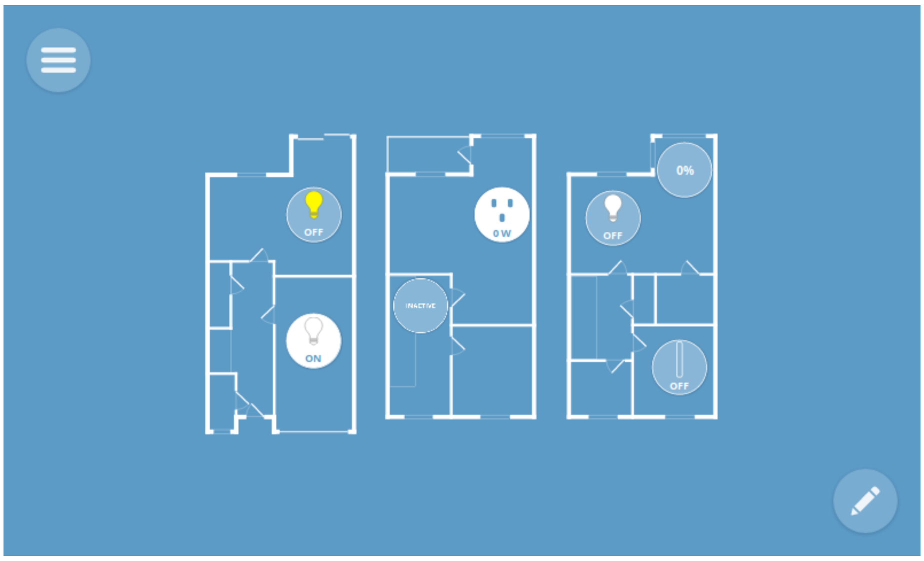
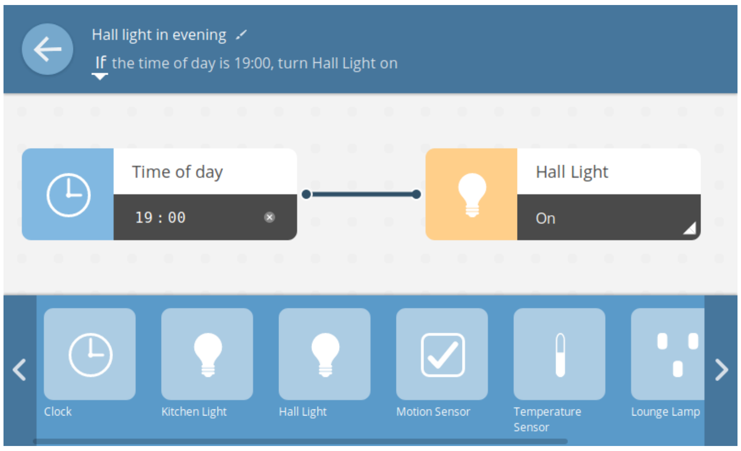


Figure 3.9: Mozilla WebThings Mapping [44]

Last but not least, the gateway also provides the functionality to set custom ruleset for each device as we can see in figure 3.10. This can be anything from setting a custom event to trigger once a threshold has been met, to automatically update any data on the sensors. As this project does not have any sensors which needs to be interacted with, we will not go into further details on this. But if the sensors need to be interacted with automatically, then this functionality is built in and ready to be used.


Hall light in evening

If the time of day is 19:00, turn Hall Light on

Figure 3.10: Mozilla WebThings Ruleset [44]

3.3 Siemens Desigo CC

Desigo CC is a platform made by Siemens. Its core functionality is to simplify the process of integrating things into their platform. It is also available for integrating different datapoints and functions along the things and thereby to create a manageable system for a developer [7].

Desigo CC is a standalone platform which makes it harder to integrate into an existing application. It is also not an open source project, making custom made solutions harder to be used for Smart Ocean in the future. Not only that, but it is also behind a paywall, which makes it less suitable for research purposes.

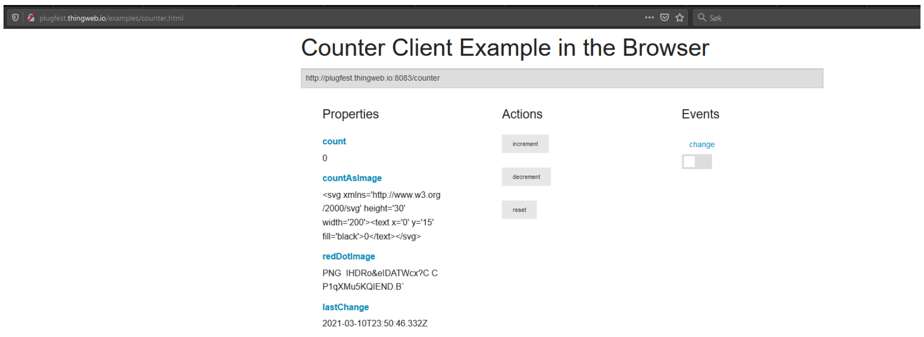
3.4 Eclipse ThingWeb

ThingWeb is Eclipse's implementation of the WoT technology. As Eclipse is a part of the W3C workgroup which means ThingWeb follows W3C's definition of WoT. The framework has a component called note-wot to replicate the thing descriptor of the W3C's WoT architecture. This involves different protocol bindings to create an adapter used to implement new things. ThingWeb also implements a runtime which is equivalent to the scripting-API needed for the WoT technology to work as intended [8].

In addition, ThingWeb includes a directory which provides an interface to register thing descriptors and a WebUI for controlling the system of provided things. Both the directory and the runtime for exposing things are open source projects, making them ideal to be used for research. The primary limitation of this framework occurs at the chosen language for development, as it is only using JavaScript. This makes it difficult to be picked up for developers unfamiliar with the language. Especially since the framework lacks proper documentation for developing thing descriptors from scratch, and further more exposing these things.

But the fact that Eclipse decided to use the schema made from W3C, means that it can be used alongside other frameworks such as Node-RED for consuming the exposed thing descriptors. This opens up the flexibility of customising the application for ones needs, and combine the frameworks when developing applications. This means using the WoT framework makes the technology compatible with each other without any extra effort.

In figure 3.11, we can see the web consumer example from Eclipse with a produced example thing descriptor that Eclipse provides for testing out frameworks. This shows how the thing descriptor from Eclipse can easily be used in any solution supporting the W3C's definition of WoT. The thing descriptor is produced by Eclipse, and consumed by the web client. The ThingWeb framework also supports the possibility to expose multiple thing descriptors at the same time, which means we do not have to create individual applications for each device.


plugfest.thingweb.io/examples/counter.html
Søk
Counter Client Example in the Browser
http://plugfest.thingweb.io:8083/counter
Properties
Actions
Events
increment
decrement
reset
change

count
0
countAsImage
<svg xmlns='http://www.w3.org/2000/svg' height='30' width='200'><text x='0' y='15' fill='black'>0</text></svg>
redDotImage
PNG IHDRo&eIDATWcx?C C P1qXMu5KQIEND.B`
lastChange
2021-03-10T23:50:46.332Z

Figure 3.11: ThingWeb Web Example

Eclipse also provides a CLI to check whether a thing description is valid. This is done by posting a thing description produced by an application, or self made into the playground which Eclipse provides at `http://plugfest.thingweb.io/playground/`.

Usually when creating a WoT application, we will have multiple sensors. The framework provides support for the functionality to expose multiple thing descriptors from the same instance. This means we do not need to create individual instances for each sensors, and can keep all the thing descriptors at one place.

3.5 W3C's Web of Things framework

W3C has also made some efforts to implement a framework, but it is currently in the experimental implementation phase and has not been updated since 2016 [38]. It is supposed to follow its definition of the framework, but it seems like the project was handed over to the other contributors in the web of things interest group.

The framework is based around the JavaScript framework NodeJS. The framework covers the thing descriptor, implementation of the framework and registering new devices to the system. When a thing has its properties updated, the system then sends a message to every associated thing about its new information.

Chapter 4

Smart Ocean Evaluation Case Studies

The Smart Ocean project has defined four representative case studies to demonstrate how the components communicate within the integrated system. By virtualising these case studies through the use of some of the frameworks studied in this thesis, gives the Smart Ocean project a guideline on how WoT may be put into practical use. We currently do not have access to physical sensors, which makes virtualizing the devices the most feasible approach. For future implementation of the frameworks on the devices, a few adjustments has to be made, but the general thing descriptors will stay the same. As all of the things used in this project are sensors, this means that we can ignore the use of actions in the thing descriptors. Below we provide further details on each of the case studies as well as how the thing descriptors will look like for each case.

The thing descriptors are shortened to make them more readable, but can be accessed fully in appendix A.

4.1 Pilot Demonstrator 1 - Local scale environmental monitoring

The description of the case study is as follows [26]:

> A multipurpose local-scale wireless network of autonomous sensors for monitoring of oceanographic and seabed environmental parameters will be established around an aquaculture plant. Candidate measurement parameters include current profiling, O2 and CO2 concentrations, gravity, gas leakage, pH, pressure, temperature, salinity and turbidity[...] PD1 will contain and demonstrate all major Smart Ocean monitoring system components and functionalities.

What we can interpret in this description is that it is a simple multi-sensor device with the properties O2, CO2 concentrations, gravity, pH, pressure, temperature, salinity and turbidity. It is also able to detect gas leakage, which will be implemented as an event in the thing description seen in listing 4.1.

```
    "id":"PD1:thing",
    "title":"PD1",
    "description":"Pilot Demonstrator 1",
    "properties":{
        "CO2":{
            "type":"integer",
            "description":"current CO2 value",
            "observable":false ,
            "readOnly":true,
            "writeOnly":false
        },
        "Temperature":{
            "type":"integer",
            "description":"current Temperature value in Celcius",
            "observable":false ,
            "readOnly":true,
            "writeOnly":false
        },
        "pH":{
            "type":"integer",
            "description":"current pH value",
            "observable":false ,
            "readOnly":true,
            "writeOnly":false
        },
        "Gravity":{
            "type":"integer",
            "description":"current G force value in m/s^2",
            "observable":false ,
            "readOnly":true,
            "writeOnly":false
        },
        "Salinity":{
            "type":"integer",
            "description":"current Salinity value in parts per
                thousand",
            "observable":false ,
            "readOnly":true,
            "writeOnly":false
        }
    },
    "events":{
        "GasLeakage":{
            "type":"boolean",
            "description":"Warning from gas leakage"
        }
    },
```

Listing 4.1: Thing descriptor of Pilot Demonstrator 1

4.2 Pilot Demonstrator 2 - Mesoscale environmental monitoring

The description of the second case study is as follows [26]:

> A real time, integrated and scalable ocean multipurpose observing system will be developed and demonstrated using acoustic technologies, for acoustic tomography to observe mean ocean temperature and water circulation; geo-positioning observations from underwater autonomous sensors; and monitoring the underwater acoustic environment. Time series of three-dimensional ocean parameters will be provided and combined with oceanographic point or profiling measurements and high-resolution dynamical ocean models through assimilation[...] Test site will be covering approximately 5 km x 5 km.

This sensor is more advanced than the one in pilot demonstrator 1 as we will handle three-dimensional data. But in the end, it is just data which has to be provided from a thing descriptor as seen in listing 4.2. As this sensor do not have any critical messages to provide, implementation of events is not required. The properties which has to be implemented in the thing descriptor is as follows; acoustic environment, acoustic tomography, geo-positioning, oceanograpic point or profiling measurements.

```
    "id":"PD2:thing",
    "title":"PD2",
    "description":"Pilot Demonstrator 2",
    "properties":{
        "AcousticTomopraphy":{
            "type":"integer",
            "unit":"hertz",
            "description":"current Acoustic Tomography in Hertz",
            "observable":false,
            "readOnly":true,
            "writeOnly":false
        },
        "AcousticEnvironment":{
            "type":"integer",
            "unit":"hertz",
            "description":"current Acoustic Environment in Hertz",
            "observable":false,
            "readOnly":true,
            "writeOnly":false
        },
        "GeoPositioning":{
            "type":"integer",
            "unit":"coordinates",
            "description":"current Geo-Positioning in coordinates",
            "observable":false,
            "readOnly":true,
            "writeOnly":false
        },
        "OceanGraphicPoint":{
            "type":"integer",
            "unit":"coordinates",
            "description":"current Acoustic Environment in Hertz",
            "observable":false,
            "readOnly":true,
            "writeOnly":false
        }
    },
```

Listing 4.2: Thing descriptor of Pilot Demonstrator 2

4.3 Pilot Demonstrator 3 - Integrity measurements offshore wind

The description of the third sensor is as follows:

> PD3 will be established and used for research, development, testing, and demonstration of sensors for integrity monitoring of (bottom-mounted) and floating wind turbine structures[...] Sensors for inspection and evaluation of cement grouting integrity will involve local acoustic resonance ("point") methods (ART), and development of distributed GUW measurement methods for wide spatial coverage.

This sensor does not have any events that needs to be tracked as seen in listing 4.3. It is not as advanced as the previous sensor to virtualize, which means we only have to provide the following properties: local acoustic resonance and GUW measurements.

```
    "id":"PD3:thing",
    "title":"PD3",
    "description":"Pilot Demonstrator 3",
    "properties":{
        "AcousticResonance":{
            "type":"integer",
            "unit":"hertz",
            "description":"current Acoustic Resonance in Hertz",
            "observable":false,
            "readOnly":true,
            "writeOnly":false
        },
        "GUW":{
            "type":"integer",
            "description":"current GUW measurements",
            "observable":false,
            "readOnly":true,
            "writeOnly":false
        }
    },
```

Listing 4.3: Thing descriptor of Pilot Demonstrator 3

4.4 Pilot Demonstrator 4 - Integrity measurements oil and gas

The description of the fourth sensor is as follows, where DAS means distributed acoustic sensing and GUW means guided wave measurements:

PD4 will be established and used for research, development, testing, and demonstration of sensors for integrity monitoring of oil and gas installations. Flow induced pipeline vibrations may be monitored using DAS. Pipeline fatigue will be investigated using GUW. Autonomous gas leakage detection systems will involve acoustic methods.

This sensor is similar to the one in pilot demonstrator 1, as now we have to consider implementing events seen in 4.4. The event that need to be implemented is gas leakage. The properties which needs to be implemented in the thing descriptor is as follows; Pipeline vibrations (DAS), GUW measurements and gas leakage monitoring.

```
    "id":"PD4:thing",
    "title":"PD4",
    "description":"Pilot Demonstrator 4",
    "properties":{
        "PipelineVibration":{
            "type":"integer",
            "unit":"hertz",
            "description":"current Pipeline Vibrations in Hertz",
            "observable":false ,
            "readOnly":true,
            "writeOnly":false
        },
        "GUW":{
            "type":"integer",
            "description":"current GUW measurements",
            "observable":false ,
            "readOnly":true,
            "writeOnly":false
        }
    },
    "events":{
        "GasLeakage":{
            "type":"boolean",
            "description":"Warning from gas leakage"
        }
    },
```

Listing 4.4: Thing descriptor of Pilot Demonstrator 4

Chapter 5

Prototype Design

By investigating the frameworks mentioned earlier, we have identified four frameworks/platforms for the four pilot demonstrators. The selected frameworks are the closest one can get for implementing the WoT technology without doing significant modification to the existing frameworks. Both prototypes will have their things exposed and a framework for consuming this information. Each prototype will handle each pilot demonstrator as they were individual sensors. This mean the prototypes will have four sensors connected to them.

As we can see in table 5.1, it is clear why the following frameworks have been chosen. Both Siemens Desigo CC and W3C's framework have significant cons to not being considered in this thesis. As for the rest, we have decided to mix some of the frameworks as both have some cons as seen in table 5.1 that makes it hard for them to be independent applications. The only framework that is standalone is Mozilla WebThings.

All the sensors in the prototypes are virtual sensors which only provides dummy data, as we do not have access to the sensors which Smart Ocean is going to use. This means all the data provided is just random numbers provided from the frameworks exposing the sensors.

Frameworks	**Pros**	**Cons**
Node-RED	Easy implementation of exposed thing descriptors made Tons of useful tools to ease development and data flow	Lacks the possibility to expose API's in an easy way
Mozilla WebThings	Easy to use and set up A lot of built in functionality from the start Supports multiple programming languages for exposing a thing	Does not follow W3C's standards Is meant for smart houses Difficult implement new functionalities in the gateway
Siemens Desigo	Unknown	Hidden behind a paywall
Eclipse Thingweb	Easy to expose a thing Has multiple binding templates ready to use Has good tutorials for development	Lacks documentation for different things within the framework Only supports Javascript and Typescript Has some issues with Events in HTML
W3C's framework	Unknown	Outdated Does not work properly

Table 5.1: Pros and Cons table for the frameworks

5.1 Prototype 1 - Node-RED and Eclipse Thingweb

The first prototype uses Node-RED for consuming thing descriptors created by Eclipse Thingweb. The reason for this, is as mentioned earlier in this thesis, that Node-RED does still not easily provide the functionality of producing thing descriptors. But it is not totally useless, as we will use it for consuming thing descriptors produced by Eclipse Thingweb. Because of this, combining both the frameworks together into a single prototype creates a prototype with many possibilities for adaptation.

5.1.1 Architecture

The prototype is as mentioned a combination of two frameworks. The ThingWeb framework is solely for exposing the thing descriptors of the pilot demonstrators. The reason for this is because ThingWeb does not provide an easy way to consume thing descriptors. This is where the Node-RED framework comes in, which helps us make sense of the information gathered from the pilot demonstrators. The reason behind the usage of the Node-RED framework is that making ThingWeb's prototype consume a thing was not intuitive for development, as it lacked proper documentation. The current prototype does not involve any databases or monitoring frameworks for the data. This means that the prototype solely evolves around the communication between the two frameworks, and how they work together.

If we take a closer look at figure 5.1, we can see how the data flow of the prototype works. The sensors are as mentioned, virtual sensors on the ThingWeb application. ThingWeb then makes sure to expose the data the sensors provide to four different thing descriptors. Node-RED then has a scheduled interval on when to retrieve the data from the thing descriptors and sends HTTP requests to the exposed thing descriptors for the data. The data is then sent back, either as plain HTTP messages, or in a WebSocket which binding template we decided to use. Once the data has reached the consumed thing descriptor on the Node-RED framework, it then translates the data into the value which Node-RED needs. In this case, just the payload of the HTTP or WebSocket message.

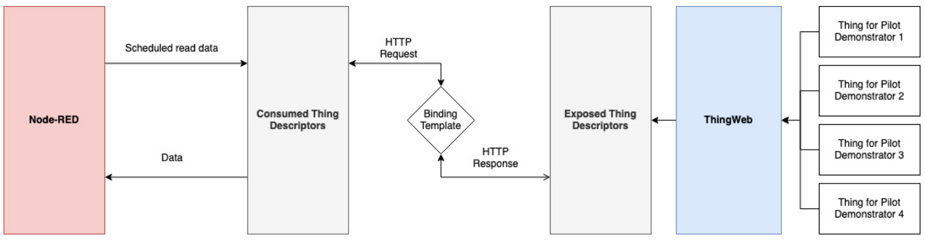

Node-RED
Scheduled read data
Data
Consumed Thing Descriptors
HTTP Request
Binding Template
HTTP Response
Exposed Thing Descriptors
ThingWeb
Thing for Pilot Demonstrator 1
Thing for Pilot Demonstrator 2
Thing for Pilot Demonstrator 3
Thing for Pilot Demonstrator 4

Figure 5.1: Data flow diagram for prototype 1

ThingWeb framework

The first part of the prototype evolves around the use of the ThingWeb framework. It is a standalone framework which produces exposed thing descriptors from the thing descriptor provided for the sensors. It gathers the information from the sensors and sends it out to the exposed thing descriptor. This framework takes care of both the properties and events which the sensors produce. ThingWeb also takes care of the binding templates for the prototype. It produces a thing descriptor with endpoints supporting both HTTP and Websockets. We decided to go for two protocols instead of one since the ThingWeb framework creates errors when trying to emit multiple events from the same endpoint with the HTTP protocol, which the developers did not intend people to do. Luckily if any other web protocols are needed, it is as easy as making the thing descriptor create a new server that emits data with the protocol desired. The only thing we would need is an open port which the desired protocol commonly uses. ThingWeb supports the following protocols to be added as binding template; CoAP, HTTP, Modbus, MQTT, NETCONF, OPC UA and WebSockets. It also allows the user to expose content of a text file. Nothing in the initial code structure or thing descriptor has to be changed to support more protocol usage. The framework is executed using Node, as the framework is built in JavaScript, and can be set up on any computer which supports this. This makes it well-suited for a gateway device, as this does not demand a lot of computer resources.

In appendix listing A.5 we can see the thing descriptor which the framework produces for the first pilot demonstrator. We will be taking a closer look at this one, as this is the most advanced device of the four. The other thing descriptions can also be found in appendix A.

Lets start with going more detailed into the first thing description. In listing 5.1 , we can see the beginning of the thing descriptor. Here we define which version of the W3C scheme we will be using for syntax. Afterwards we make sure to give the thing an unique identifier to separate it from the three other pilot demonstrators. Then we provide a simple title and description of which pilot demonstrator it is. This is the only metadata which is mandatory to get the thing description produced and working.

```
{
    "@context":[
        "https://www.w3.org/2019/wot/td/v1",
        {
            "@language":"en"
        }
    ],
    "id":"PD1:thing",
    "title":"PD1",
    "description":"Pilot Demonstrator 1",
```

Listing 5.1: Interaction Affordance in the thing description of the first pilot demonstrator

As we can see in listing 5.2, the framework links every interaction affordances with an endpoint on the server which the framework produces. This is done by linking each affordance to an URL which can be used while consuming the thing, as we can see in the thing descriptor, where each property and event is linked towards an specific URL. As mentioned earlier in this thesis, we currently have no use for actions or manipulating data directly on the sensors. This means we can set all the properties to read only and make sure they cannot be manipulated. We also ensure that the properties cannot be written directly towards on the sensors in the thing description, which makes the framework reject any post request towards the thing.

```
        "CO2":{
            "type":"integer",
            "description":"current CO2 value",
            "observable":false,
            "readOnly":true,
            "writeOnly":false,
            "forms":[
                {
                    "href":"http://192.168.1.36:8080/PD1/properties/CO2"
                        ,
                    "contentType":"application/json",
                    "op":[
                        "readproperty"
                    ],
                    "htv:methodName":"GET"
                }
            ]
        },
```

Listing 5.2: Interaction Affordance in the thing description of the first pilot demonstrator

We can see in listing 5.3 how the framework creates links to all the properties and events which the things provides, which can be accessed. This will come in handy for consuming the thing descriptors in the future, to ensure we have all the information which the sensors provide, even when the sensor is being updated.

```
    "forms":[
        {
            "href":"http://192.168.1.36:8080/PD1/all/properties",
            "contentType":"application/json",
            "op":[
                "readallproperties",
                "readmultipleproperties",
                "writeallproperties",
                "writemultipleproperties"
            ]
        }
    ],
```

Listing 5.3: Properties endpoint in the thing description of the first pilot demonstrator

When we take a look at the events of the thing, we can see the WebSocket server has correctly been set up by the binding template provided by the framework, as shown in listing 5.4. This is because the framework would not set up any endpoints with the protocol if it did not work. As mentioned earlier, we can still access the event through HTTP, but that will eventually crash the ThingWeb framework. This is not possible to remove, as setting up a new binding protocol ensures all the data on the sensor will be exposed through that protocol. As for now, we will ignore this endpoint, and keep it out of our prototype.

```
    "events":{
        "GasLeakage":{
            "type":"boolean",
            "description":"Warning from gas leakage",
            "forms":[
                {
                    "href":"http://192.168.1.36:8080/PD1/events/
                        GasLeakage",
                    "contentType":"application/json",
                    "subprotocol":"longpoll",
                    "op":[
                        "subscribeevent"
                    ]
                },
                {
                    "href":"ws://192.168.1.36:8081/PD1/events/GasLeakage
                        ",
                    "contentType":"application/json",
                    "op":"subscribeevent"
                }
            ]
        }
    },
```

Listing 5.4: Event in the thing description of the first pilot demonstrator

In the end of the thing description seen in listing 5.5, we can see the security aspect of the thing descriptor and how this works. As the extension from Node-RED does not easily support security manipulation, we skip this for the prototype. But if we were to add any security between the sensors and the application consuming the thing, then this would be the place to start.

```
    "securityDefinitions":{
        "nosec_sc":{
            "scheme":"nosec"
        }
    }
}
```

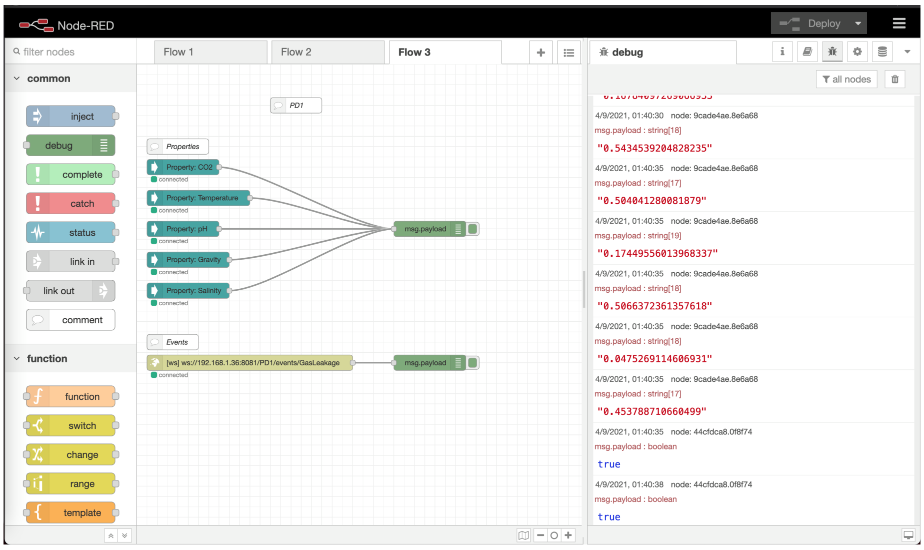
Listing 5.5: Security in the thing description of the first pilot demonstrator

Node-RED

The second part of the prototype is the Node-RED framework. This is used as a replacement for the ThingWeb's method to consume a thing. Its sole purpose is only to gather the data from the thing descriptor and handle it (in our case, only showcases it in a debug log). As mentioned earlier, the thing descriptor sets up endpoints to access the data from the sensors. The framework primarily uses the node-red-contrib-wot to be useful with handling things, as the framework itself does not yet easily support the technology. Node-RED has mentioned they will support the framework in the future, but has yet to develop this functionality. Meanwhile, the extension node-red-contrib-wot will be a useful replacement to show what can be done with Node-RED. To add the things, we have to manually enter the address provided from the ThingWeb framework, which is not as simple as mDNS. Once this has been done, the extension takes care of the rest, and provides the user with all the functionalities which the thing has.

In figure 5.2 we can see how the first pilot demonstrator can be used with Node-RED. The figure shows all the properties being read automatically by the Node-RED framework, and that the gas leakage events are being emited once the

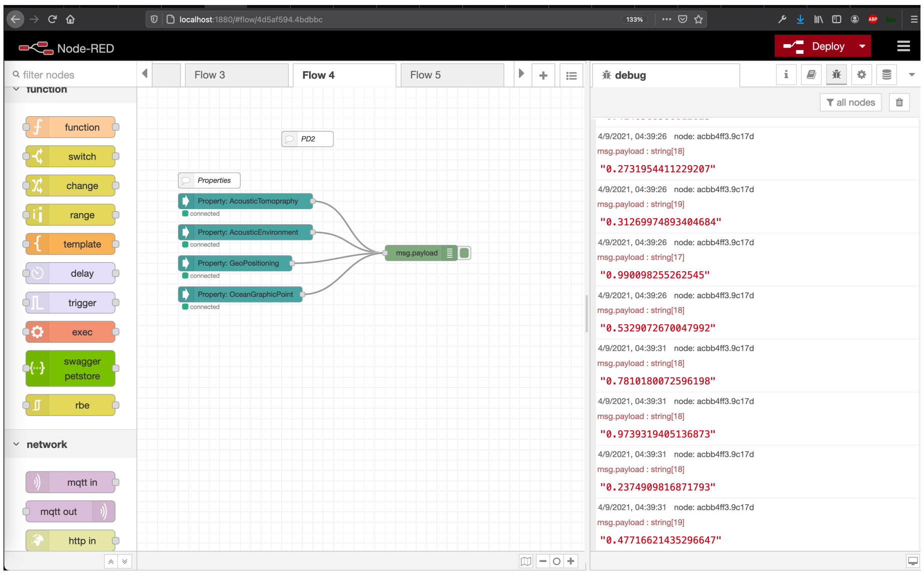
CO2 level becomes to high. The event is being transmitted by the WebSocket protocol as mentioned earlier, to prevent the framework from suddenly crashing during runtime. This means that the connection between the frameworks are continuously talking to each other to ensure the event safely reaches the Node-RED framework. The data on the right side of the figure is just dummy data between 0 and 1 to showcase how the frameworks works together.



```
4/9/2021, 01:40:30   node: 9cade4ae.8e6a68
msg.payload : string[18]
"0.5434539204828235"
4/9/2021, 01:40:35   node: 9cade4ae.8e6a68
msg.payload : string[17]
"0.504041280081879"
4/9/2021, 01:40:35   node: 9cade4ae.8e6a68
msg.payload : string[19]
"0.17449556013968337"
4/9/2021, 01:40:35   node: 9cade4ae.8e6a68
msg.payload : string[18]
"0.5066372361357618"
4/9/2021, 01:40:35   node: 9cade4ae.8e6a68
msg.payload : string[18]
"0.0475269114606931"
4/9/2021, 01:40:35   node: 9cade4ae.8e6a68
msg.payload : string[17]
"0.453788710660499"
4/9/2021, 01:40:35   node: 44cfdca8.0f8f74
msg.payload : boolean
true
4/9/2021, 01:40:38   node: 44cfdca8.0f8f74
msg.payload : boolean
true
```

Figure 5.2: Node-RED Pilot Demonstrator 1

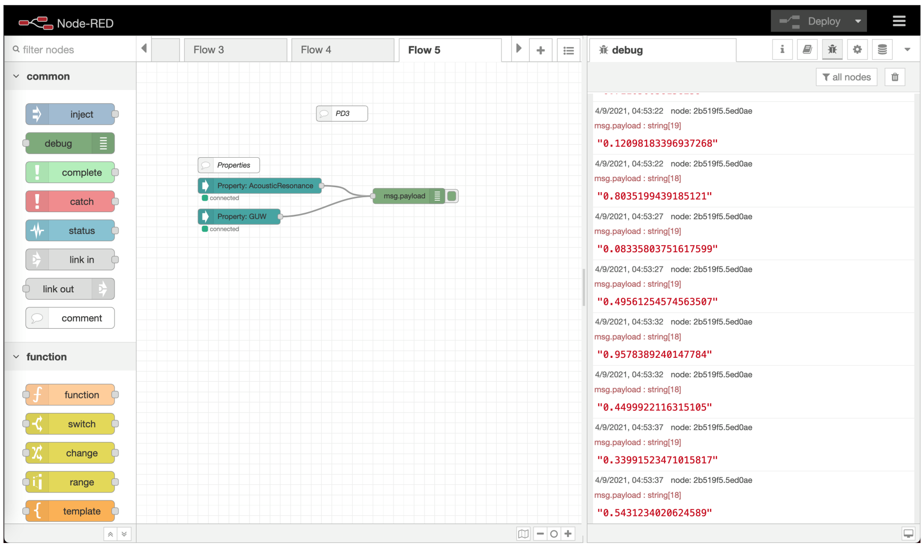
For the second pilot demonstrator seen in figure 5.3, we can see it looks similar to the first pilot demonstrator. Only this time we have four properties with readonly functionality.



```
4/9/2021, 04:39:26   node: acbb4ff3.9c17d
msg.payload : string[18]
"0.2731954411229207"
4/9/2021, 04:39:26   node: acbb4ff3.9c17d
msg.payload : string[19]
"0.31269974893404684"
4/9/2021, 04:39:26   node: acbb4ff3.9c17d
msg.payload : string[17]
"0.990098255262545"
4/9/2021, 04:39:26   node: acbb4ff3.9c17d
msg.payload : string[18]
"0.5329072670047992"
4/9/2021, 04:39:31   node: acbb4ff3.9c17d
msg.payload : string[18]
"0.7810180072596198"
4/9/2021, 04:39:31   node: acbb4ff3.9c17d
msg.payload : string[18]
"0.9739319405136873"
4/9/2021, 04:39:31   node: acbb4ff3.9c17d
msg.payload : string[18]
"0.2374909816871793"
4/9/2021, 04:39:31   node: acbb4ff3.9c17d
msg.payload : string[19]
"0.47716621435296647"
```

Figure 5.3: Node-RED Pilot Demonstrator 2

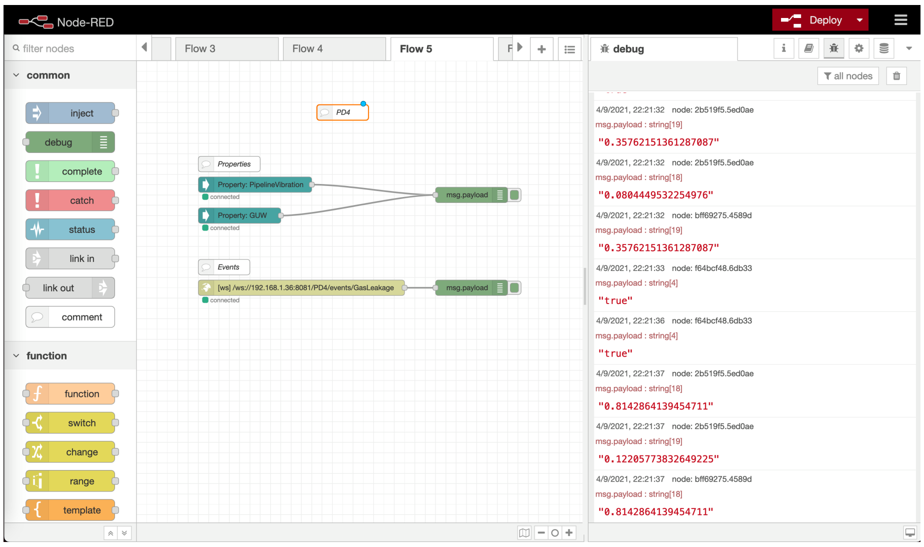
The third pilot demonstrator seen in figure 5.4, we can see its as simple as only two properties assigned the read property nodes. This showcases that even the smallest sensors have an easy way into the Node-RED framework, and can be easily manipulated by the same framework.



```
4/9/2021, 04:53:22   node: 2b519f5.5ed0ae
msg.payload : string[19]
"0.12098183396937268"
4/9/2021, 04:53:22   node: 2b519f5.5ed0ae
msg.payload : string[18]
"0.8035199439185121"
4/9/2021, 04:53:27   node: 2b519f5.5ed0ae
msg.payload : string[19]
"0.08335803751617599"
4/9/2021, 04:53:27   node: 2b519f5.5ed0ae
msg.payload : string[19]
"0.49561254574563507"
4/9/2021, 04:53:32   node: 2b519f5.5ed0ae
msg.payload : string[18]
"0.9578389240147784"
4/9/2021, 04:53:32   node: 2b519f5.5ed0ae
msg.payload : string[18]
"0.4499922116315105"
4/9/2021, 04:53:37   node: 2b519f5.5ed0ae
msg.payload : string[19]
"0.33991523471015817"
4/9/2021, 04:53:37   node: 2b519f5.5ed0ae
msg.payload : string[18]
"0.5431234020624589"
```

Figure 5.4: Node-RED Pilot Demonstrator 3

The fourth and last pilot demonstrator seen in figure 5.5 looks similar to the first pilot demonstrator. The only difference is the amount of properties. It as well has a connection with the ThingWeb API through WebSocket for the event.



```
4/9/2021, 22:21:32   node: 2b519f5.5ed0ae
msg.payload : string[19]
"0.35762151361287087"
4/9/2021, 22:21:32   node: 2b519f5.5ed0ae
msg.payload : string[18]
"0.0804449532254976"
4/9/2021, 22:21:32   node: bff69275.4589d
msg.payload : string[19]
"0.35762151361287087"
4/9/2021, 22:21:33   node: f64bcf48.6db33
msg.payload : string[4]
"true"
4/9/2021, 22:21:36   node: f64bcf48.6db33
msg.payload : string[4]
"true"
4/9/2021, 22:21:37   node: 2b519f5.5ed0ae
msg.payload : string[18]
"0.8142864139454711"
4/9/2021, 22:21:37   node: 2b519f5.5ed0ae
msg.payload : string[19]
"0.12205773832649225"
4/9/2021, 22:21:37   node: bff69275.4589d
msg.payload : string[18]
"0.8142864139454711"
```

Figure 5.5: Node-RED Pilot Demonstrator 4

All the thing descriptors are executed on the same server, and trying to reach the framework will give the user the option to pick which sensor to use. This showcases that even with new sensors being added, it will not impact the other sensors currently in use. This also means that the gateway which the framework is put up on can support multiple sensors in an easy way.

Its important to address that the thing descriptors produced by eclipse ThingWeb are not solely locked to Node-RED. As a matter of fact, any new framework that is produced by the standards of W3C will be compatible with the prototype created in the ThingWeb framework. This makes the prototype quite robust, in the case where its easy to update and customize to changes in the future. This also opens up the possibility to move to a better framework if someone would create one.

5.2 Prototype 2 - Mozilla Webthings and Mozilla's Gateway

The second prototype uses both frameworks provided by Mozilla. The WebThings framework is used to produce a thing descriptor which the gateway can pick up and use in the gateway prototype. The prototype uses dummy data as mentioned earlier to simulate the pilot demonstrators and how a real world application would be implemented and used in a real scenario. The framework is used to create an endpoint for the gateway to communicate with the sensors from the deep sea.

5.2.1 Architecture

For the prototype to work, we first had to use the framework to develop thing descriptors which followed the schema provided by Mozilla. All the thing descriptors created by the Mozilla WebThings framework can be found in appendix A. Once the thing descriptors were made, the framework took care of exposing them. By doing this, the framework produces a JSON object which looks similar to a classic REST API, which can be seen in listing A.9. The exposed JSON file provides different endpoints for collecting data from the properties and events exposed by the framework. Trying to use this exposed API created by the framework alone would demand a lot of work for the developer to use, and would not necessary be robust. This is where the gateway comes in. The gateway can be set up on any Linux device with ease, which provides both a local and a URL which can be accessed over the World Wide Web. The gateway provides a user interface for handling the pilot demonstrators, and supports the use of custom rules and add-ons either provided by Mozilla or self developed.

The data flow of this prototype looks the same as the data flow in prototype one. This means we can revisit figure 5.1, and change Node-RED for the Mozilla Gateway, and the Eclipse ThingWeb with the Mozilla WebThings framework.

The gateway provides the discovery functionality for adding self-made thing descriptors, alongside the sensors which the framework supports. As long as the produced thing descriptor follows Mozilla's scheme, it will use mDNS to automatically find the things exposed on the local network. This is not completely necessary for this project. The reason is that the sensors will most likely be added through IP addresses as the sensors will be located on a different network than the gateway.

To get the prototype to work, we first have to produce a thing descriptor for the sensors with Mozillas scheme. This means finding the corresponding properties compared to what data the sensor provides. As the schema is made for smart homes, this means we have to adapt it to work with the data provided from the Smart Ocean sensors, as some properties are not completely supported by the gateway. We can reuse the property LevelProperty to provide information which can be interpreted by the gateway, as this is a standard way to provide information in form of a number. The sensors also lacks the functionality to do actions, as they are only measurement sensors. This means we can make sure all the properties are set to readonly, and ignore the usage of Mozillas action statements.

Logging

The gateway gives a visual representation of the properties, which can be seen in figure 5.6. This can be manipulated to show the data that the user needs, anything from CO2 levels to GUW. The user then gets the choice of tracking the properties by minutes, hours, days and weeks. This log can also be accessed in the files of the gateway framework to be stored for later usage, as the gateway tends to delete the logs to prevent small gateways to overextend their capacity. There is currently no other way to visualise the data, which the framework supports.

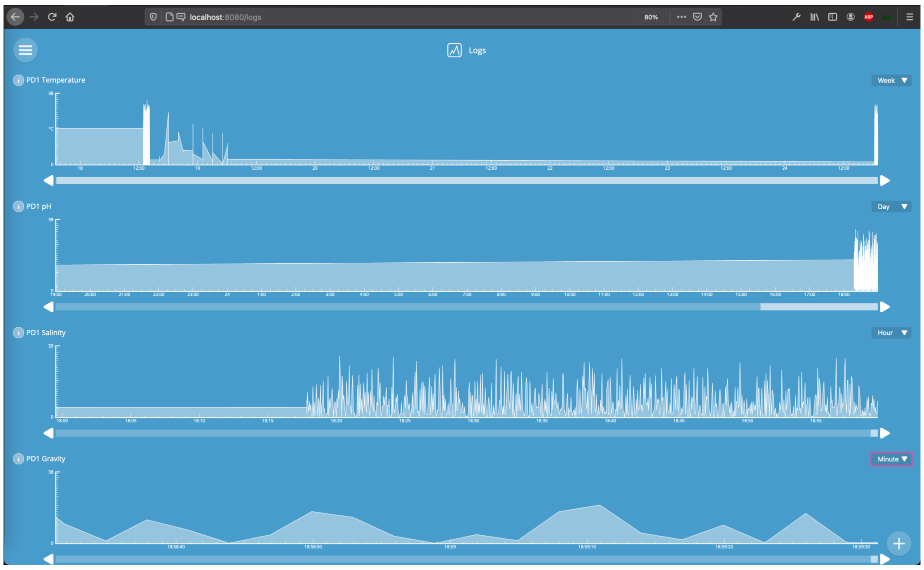


Figure 5.6: ThingWeb Logging

Mapping

In figure 5.7 we can see how the mapping functionality can be used in a real case scenario. By uploading a map of where the sensors will be placed, one can then drag the sensors to the desired location for a better understanding of the infrastructure. Unfortunately the sensors can not be scaled down in size, which means including more sensors will make the map hard to read in the future as the map will be overfilled. For this thesis, this functionality wont be as useful, as the sensor will only be available to show one property on the map. And as seen earlier, there is not a good way to specify which property this will be.

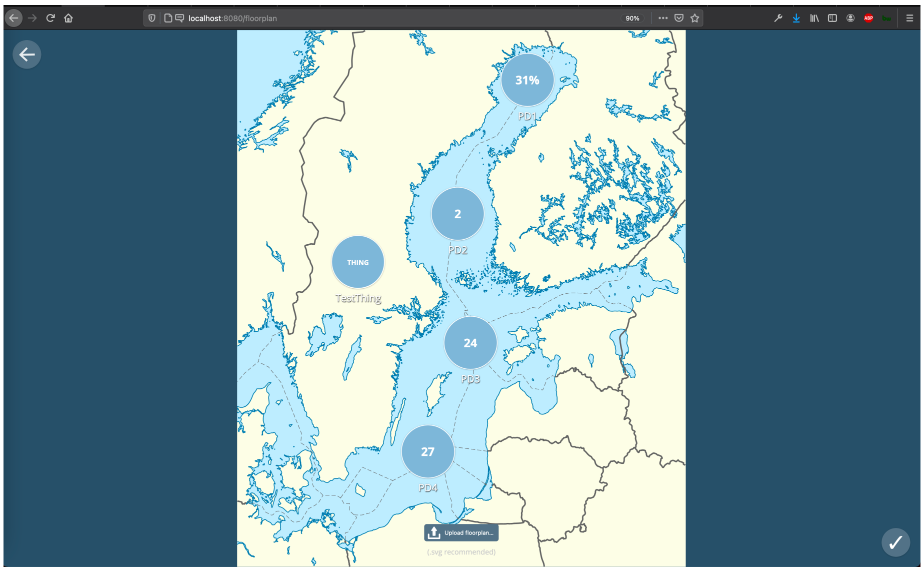


Figure 5.7: ThingWeb Mapping

Pilot Demonstrators

The listing A.9 is the thing descriptor for the first pilot demonstrator created for this prototype. As we can see, this thing descriptor is a little different from the standard WoT thing descriptor. It looks similar and works the same way as the thing descriptors created by W3C, but only uses the necessary fields which the gateway needs. This is because Mozilla has stated that the W3C's scheme is too generic, which is not optimal for their platform. This means their framework only supports WebSockets and HTTP, which can not be changed. It also does not use forms to provide the URL of the endpoint of the property, which tends to be dynamic if suddenly the IP addresses change. Because of this, Mozilla is using a dynamic approach of linking all the endpoints with URN's, as seen in listing 5.6. But it is also composed of five properties and one event to provide the functionality which the sensor provides, just like in the ThingWeb and Node-RED prototype.

```
            "links":[
                {
                    "rel":"property",
                    "href":"/properties/Temperature"
                }
            ],
```

Listing 5.6: Linking in the thing description for WebThings of the first pilot demonstrator

As we can see in listing 5.7, every property and event gets their own referential link to access the data provided from the sensor. The JSON-LD annotation @type is used for the gateway to get information from Mozillas scheme to ensure the thing descriptor is created in the correct way. The JSON-LD @type is essential for the framework to work with the gateway, as it ensures the thing descriptor provided will follow the syntax which the gateway uses.

```
    "properties":{
        "Temperature":{
            "unit":"degree celsius",
            "@type":"TemperatureProperty",
            "description":"The current temperature in celsius",
            "readOnly":true,
            "links":[
                {
                    "rel":"property",
                    "href":"/properties/Temperature"
                }
            ],
            "title":"Temperature",
            "type":"number"
        },
```

Listing 5.7: Properties in the thing description for WebThings of the first pilot demonstrator

All the sensors have their own dashboard showcasing the sensor data from the things, which also can be used along logging, rule-set and mapping. All the examples look quite similar to each other, with the difference being the different properties and events. What is special about each case is that in this prototype, each property has to be assigned a type for the gateway to understand what type of data to showcase.

Figure 5.8 shows a clearer picture how the first pilot demonstrator is used in the gateway. The number in the center is a random property picked by the framework from the properties provided for the gateway. For the time being it seems like this cannot be changed to something specific. As we can see in the figure, temperature, CO2 level has been assigned visual boxes alongside the value from the sensor. The other properties did not have any specific type which the Mozilla scheme supported. This meant we had to use LevelProperty to show the data, without any visuals.

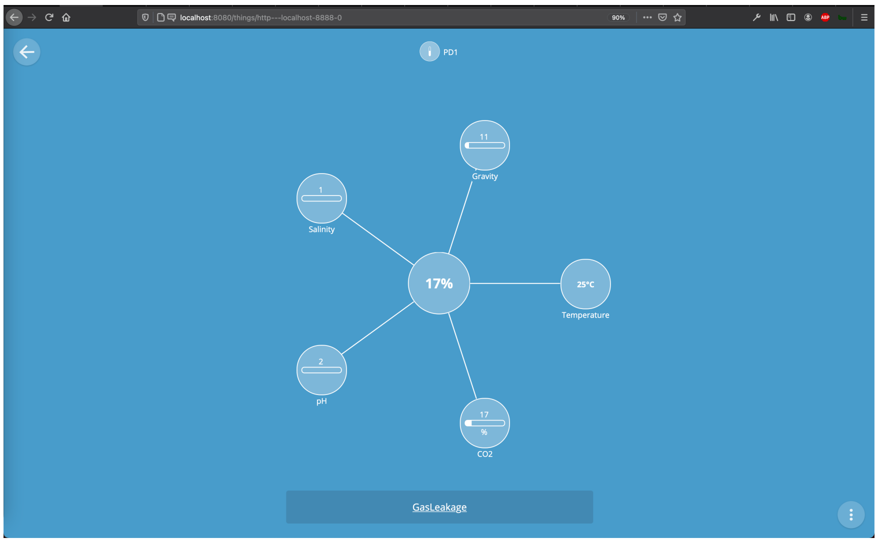


Figure 5.8: Pilot Demonstrator 1 in the WebThings Gateway

In the second pilot demonstrator seen in figure 5.9, we can see the LevelProperties in action, as well as the FrequencyProperty datatype as the framework supports.

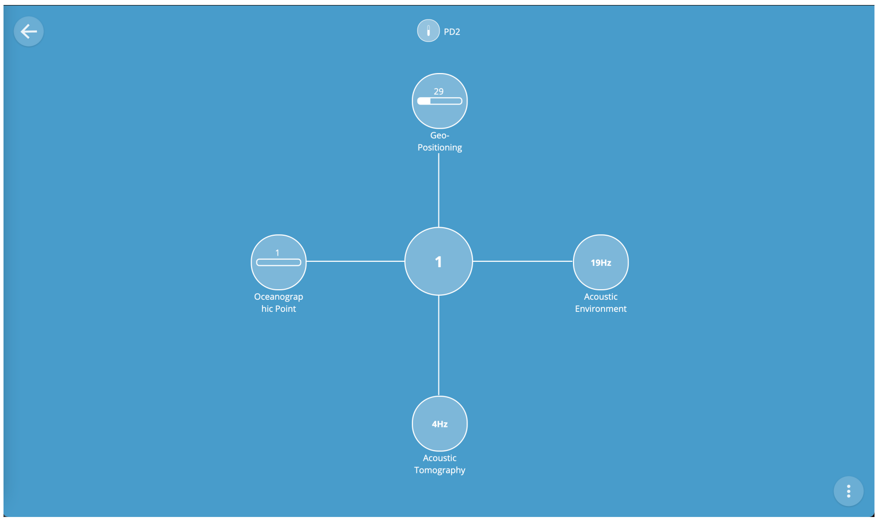


Figure 5.9: Pilot Demonstrator 2 in the WebThings Gateway

The third pilot demonstrator is the smallest one out the four, as we can see in figure 5.10. It uses both the FrequencyLevel and LevelProperty as datatypes for the properties.

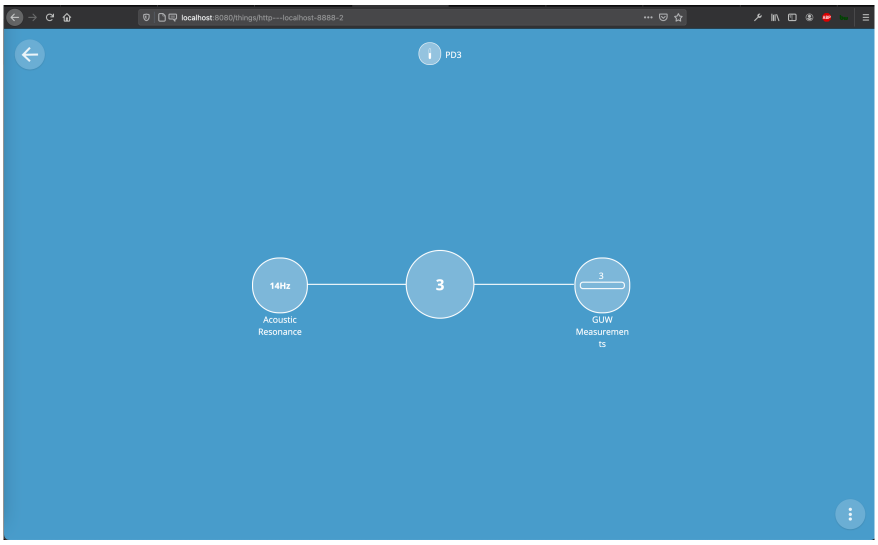


Figure 5.10: Pilot Demonstrator 3 in the WebThings Gateway

Last but not least, we have the fourth pilot demonstrator as seen in figure 5.11. This one also only has two properties as the last one, with the event of gas leakage as well.

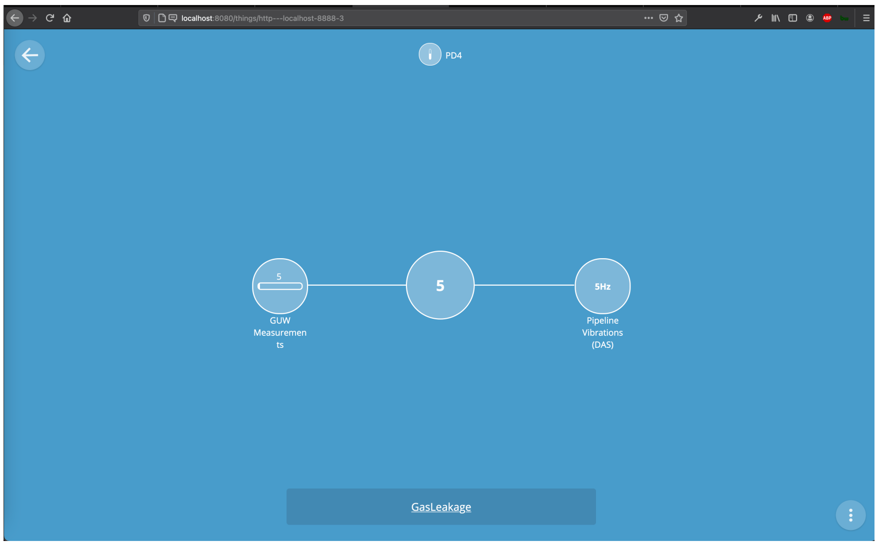

localhost:8080/things/http---localhost-8888-3
PD4
5
GUW Measurements
5
5Hz
Pipeline Vibrations (DAS)
GasLeakage

Figure 5.11: Pilot Demonstrator 4 in the WebThings Gateway

We can also see in the figures 5.8 and 5.11 that the gas leakage event works as intended. The gas leakage will update itself by sending updates to the gateway when the condition is met. The gateway then sends a pop-up to the gateway with the condition and a timestamp for the event. This event can also be set to a custom ruleset, to do actions as desired when the condition is met.

Chapter 6

Implementation and Deployment

In this chapter we go through the prototypes introduced in chapter 5 and explain how they are implemented and how they can be deployed. As both of the prototypes has frameworks not requiring code, this means that this chapter is important to execute and replicate the prototypes.

6.1 Implementation

In this section, we cover what the code in the prototypes does, and what one can do to manipulate them to ones needs. We will also showcase how to set up the different frameworks.

6.1.1 Prototype 1 - Node-RED and Eclipse Thingweb

The first prototype is based on the ThingWeb framework and the Node-RED framework. We start by addressing the ThingWeb framework as it is important to expose a thing before consuming them.

Implementation of the ThingWeb

The ThingWeb framework gives a simple tutorial on how to expose a thing [36], but this is the only documentation they provide. This means we have to start looking through the examples they provide to figure out how the framework actually works. The framework is quite simple once we see how the different components works. It is put together by a servient which handles the different binding servers provided by the framework.

Listing 6.1 shows the code needed to get the framework to create a servient with a HTTP and WebSocket server. In lines 1-5 of the code we can see our four pilot demonstrators being implemented. We will be going into further details on those below. The most important part is line 7, where we integrate the framework into our prototype. With this comes the tools we need like binding protocols, server objects and our way to expose the pilot demonstrators. This implementation grabs the framework as a library, downloaded by the npm package manager. We do not need to have the entire framework in our project, which makes the project a lot easier to read. In lines 10-21, we can see how we use the framework to set up a HTTP server and WebSocket server as explained in chapter 5. This is how we use the binding protocols provided by the framework. We can easily change this based on which protocols we need to consume. In lines 24-29, we initiate the thing descriptors for the pilot demonstrators, and passes the ThingWeb framework along to further use its functions in the thing descriptors. This is how we make the framework instantiate several thing descriptors at the same time.

```
 1  //Implementation of the Thing
 2  TD1 = require("./dist/TD1.js").WotDevice
 3  TD2 = require("./dist/TD2.js").WotDevice
 4  TD3 = require("./dist/TD3.js").WotDevice
 5  TD4 = require("./dist/TD4.js").WotDevice
 6
 7  Servient = require("@node-wot/core").Servient
 8
 9  //Importing of the bindings
10  HttpServer = require("@node-wot/binding-http").HttpServer
11  WebsocketServer = require("@node-wot/binding-websockets").
        WebSocketServer
12
13  //Creating the instances of the binding servers
14  var httpServer = new HttpServer({port: 8080});
15  var websocketServer = new WebsocketServer({port: 8081});
16
17  //Building the servient object
18  var servient = new Servient();
19  //Adding the different bindings to the server
20  servient.addServer(httpServer);
21  servient.addServer(websocketServer);
22
23  //Starting the servients
24  servient.start().then((WoT) => {
25      td1 = new TD1(WoT, TD_DIRECTORY);
26      td2 = new TD2(WoT, TD_DIRECTORY);
27      td3 = new TD3(WoT, TD_DIRECTORY);
28      td4 = new TD4(WoT, TD_DIRECTORY);
29  });
```

Listing 6.1: The servient server in the ThingWeb framework

The code for the thing descriptors made for the pilot demonstrators has been shortened for presentation purposes, but can be found complete in appendix B. We will only cover the code for the first pilot demonstrator, as all the pilot demonstrators are quite similar.

The things created through the framework is implemented in TypeScript, with Node compiling it to JavaScript at run time. This can be changed based on what JavaScript version one desires in the tsconfig file of the prototype. This makes the development process a little easier, as TypeScript tends to be more understandable for a developer to use. We could also directly use the framework in a JavaScript prototype.

The package.json file takes care of specifying the required packages like the framework and the desired binding templates to be used during development. We also have to make sure to implement the right binding template framework in this file, if one desired another protocol for the exposed thing descriptor.

In listing 6.2, we can see how the thing descriptor is implemented into the framework. It looks the same as the thing descriptor presented in chapter 4 on the case studies. This thing descriptor is made by following WC3's scheme and has to be created and put into the framework for it to work. This means adding any new properties, also requires manipulation of the thing descriptor. The thing descriptor is the metadata of the sensor, which is not automatically created. There also exists possibilities to add different parameters to the properties and events like required, format and const variable. This is where we have to add the security description before implementing it through the framework. Its important to note that if the W3C's scheme gets updated, this means we have to change the versioning in the @context parameter, and update the thing descriptor accordingly. Creating the thing descriptor is what will take the most amount of effort to do, as one has to analyse what is needed for the prototype beforehand, and create the thing descriptor accordingly.

```
        this.WoT.produce(
            {
        "@context": [
          "https://www.w3.org/2019/wot/td/v1",
          { "@language": "en" }
        ],
        id: "PD1:thing",
        title: "PD1",
        description: "Pilot Demonstrator 1",
        properties: {
          CO2: {
            type: "integer",
            description: "current CO2 value",
            observable: false,
            readOnly: true
          },
          Temperature: {
            type: "integer",
            description: "current Temperature value in Celcius",
            observable: false,
            readOnly: true
          },
          pH: {
            type: "integer",
            description: "current pH value",
            observable: false,
            readOnly: true
          },
          Gravity: {
            type: "integer",
            description: "current G force value in m/s^2",
            observable: false,
            readOnly: true
          },
          Salinity: {
            type: "integer",
            description: "current Salinity value in parts per
                thousand",
            observable: false,
            readOnly: true
          }
        },
        events: {
          GasLeakage: {
            type: "boolean",
            description: "Warning from gas leakage"
          }
        }
      }
```

Listing 6.2: Thing Descriptor for the first pilot demonstrator in the ThingWeb framwork

After the framework has been instantiated, we can expose the different properties and events with desired values, as seen in listing 6.3. This is where we take care of adding the right properties and events to the different placeholders in the thing descriptor. This means that once the framework exposes the thing descriptor, each interaction affordance gets linked to a function in the prototype which emits a value.

```
        ).then((exposedThing)=>{
      this.thing = exposedThing;
      this.td = exposedThing.getThingDescription();
        this.add_properties();
      this.add_events();
      this.thing.expose();
      if (tdDirectory) { this.register(tdDirectory); }
        });
    }
```

Listing 6.3: Exposing the thing descriptor in ThingWeb

Once the framework has been instantiated and exposed, we also have to make sure it gets registered to the framework as seen in listing 6.4. This is done by using the ThingWeb framework to register the WoT thing to the HTTP and WebSocket servers created. If this fails, it will post the error to the console and try again within 10 seconds. This is useful if the prototype would require some time to set up its functions to retrieve the data from the sensors on the bottom of the ocean.

```
    public register(directory: string) {
        console.log("Registering TD in directory: " + directory)
        request.post(directory, {json: this.thing.
            getThingDescription()}, (error, response, body) => {
            if (!error && response.statusCode < 300) {
                console.log("TD registered!");
            } else {
                console.debug(error);
                console.debug(response);
                console.warn("Failed to register TD. Will try again
                     in 10 Seconds...");
                setTimeout(() => { this.register(directory) },
                    10000);
                return;
            }
        });
    }
```

Listing 6.4: Registering the thing descriptor to the ThingWeb framework

Adding data to the interaction affordances, is done by the code shown in listing 6.5 and listing 6.6. Currently they are only random sensor data values and random intervals, but once the sensors on the ocean floor has been set up, one just have to link these towards the data. As seen in listing 6.5, we set an interval for each second to change the value CO2 in the thing descriptor. Listing 6.6 is similar, where we can see an event being emitted once a random number above 50 is given. This is only to showcase how the constant update of values will impact the prototype, as we currently do not have access to the smart ocean sensor data.

```
    private add_properties() {
        //Random values generated to the properties to emit
        setInterval(() => {
      this.thing.writeProperty("CO2", Math.random().toString());
    }, 1000);
```

Listing 6.5: Adding properties to the thing descriptor

```
    private add_events() {
    //Interval set for when the event will emit
    setInterval(() => {
      if (Math.floor(Math.random() * 101) > 50) {
        this.thing.emitEvent("GasLeakage", true);
      }
    }, 3000);
    }
}
```

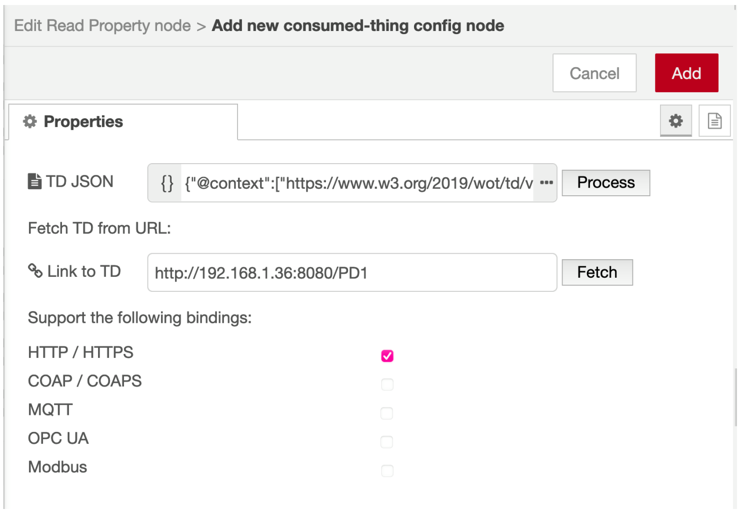
Listing 6.6: Adding event to the thing descriptor

Configuration of the Node-RED

The configuration of the Node-RED framework is a bit different from the Thing-Web implementation. The reason is that Node-RED does not require any code to run, because its a visual dataflow tool. This means we only have to get the framework to run, install the required extensions, and then use the nodes provided. Explanation on how to do this will be given further into this thesis.

Once this is done, we can start using the Node-RED framework along with the thing description from the ThingWeb. The first thing we have to do is to figure out the IP address which the ThingWeb framework is hosted at. Once this is done, we can drag out the write property node and a WebSocket in node from the node selection menu.

We then doubleclick the readproperty node to open the edit window for the node. Here we continue clicking on the pencil beside thing. Once we are here, we get the screen seen in figure 6.1. Now we can add the IP address of the first pilot demonstrator for the node to consume it. It will then automatically fetch the thing descriptor from the ThingWeb framework and the node will act as the sensor itself.


Edit Read Property node > **Add new consumed-thing config node**

Cancel Add

Properties

TD JSON {} {"@context":["https://www.w3.org/2019/wot/td/v ... Process

Fetch TD from URL:

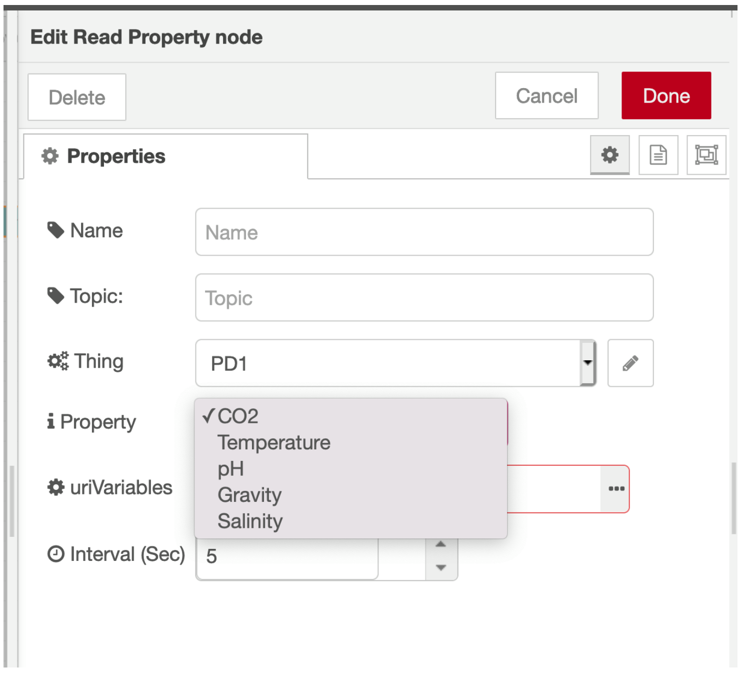
Link to TD http://192.168.1.36:8080/PD1 Fetch

Support the following bindings:

HTTP / HTTPS	✓
COAP / COAPS	
MQTT	
OPC UA	
Modbus	

Figure 6.1: Node-RED readProperty config screen

From here we can head back to the property edit window. If done correctly, the properties and settings will automatically be added. Then we can pick which property we want to showcase in the property dropdown box, as seen in figure 6.2. After this is done, we can select the interval in which the Node-RED framework fetches the data.


Edit Read Property node

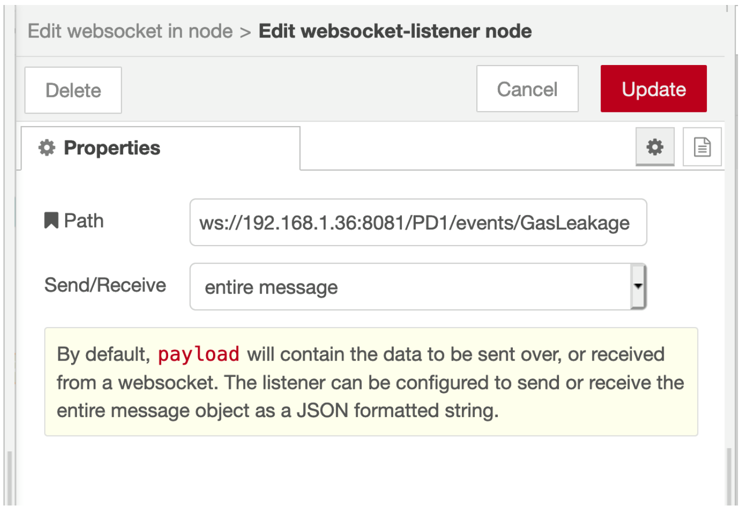
Delete Cancel Done

Properties

Name	Name
Topic:	Topic
Thing	PD1
Property	✓CO2 / Temperature / pH / Gravity / Salinity
uriVariables	
Interval (Sec)	5

Figure 6.2: Node-RED readProperty edit screen

For the WebSocket node, we doubleclick it to open up the edit window as shown in figure 6.3. We then have to go into the thing descriptor and find the specific endpoint for the WebSocket and add this to its path. Once this is done, click done and the node should be fully functional.



Path: ws://192.168.1.36:8081/PD1/events/GasLeakage

Send/Receive: entire message

By default, `payload` will contain the data to be sent over, or received from a websocket. The listener can be configured to send or receive the entire message object as a JSON formatted string.

Figure 6.3: Node-RED WebSocket edit screen

In the end, we grab the desired nodes to manipulate the data provided from the ThingWeb framework. In our case, we just use a simple debug node to make sure the connection is set up properly and that the ThingWeb framework sends out the data we desire.

6.1.2 Prototype 2 - Mozilla WebThings and Gateway

The second prototype is based entirely on the frameworks provided by Mozilla. This means that the frameworks work quite well with each other in terms of interoperability. As the gateway is a standalone framework for creating smart houses, this means it can also run stand-alone without the use of the WebThings framework. The gateway was originally not built with WoT in mind, but has given the developers the tools needed to use it.

The prototype is split into two parts as mentioned earlier. One application for exposing things as thing descriptors, and one standalone application for the gateway itself. The gateway is already a stand-alone application, which means we only have to configure it to allow WoT devices to connect to it.

Mozilla WebThings

We will start by going through the implementation of the WebThings application. As Mozilla WebThings supports multiple programming languages, we have decided to go with Java, as its the programming language most developers are familiar with. We will start going through the code, explaining what each section does, and how one can reuse it. The four pilot demonstrators looks similar in implementation, which is why we will only cover the first pilot demonstrator.

As Java does not support the use of frameworks like JavaScript packages seen in our last prototype, this means we have to import the entire framework into our application to get it to work. The implementation looks similar to the ThingWeb implementation, in the way where we have to manually create a thing descriptor for each sensor before we can start exposing them.

We now go through the code and explain how we use the framework to expose things through WoT. We use the framework to create a thing object as seen in listing 6.7. This object is going to take care of keeping track of all the interaction affordance for the sensor. It allows us to save all the properties and events in a mapping list in the object. This object is also going to handle the exposure of the thing descriptor made.

```
39          Thing thing = new Thing("PD1",
40                  "PD1",
41                  new JSONArray(Arrays.asList("MultiLevelSensor")),
42                  "Sensor for underwater technology");
```

Listing 6.7: Defining the WoT thing in WebThings

Then we have to add the properties to the thing object. The properties do have to follow the schema from Mozilla, the same way as ThingWeb framework had to follow WC3's schema. An example on how this can be done can be seen in listing 6.8, where we add the name of the property and its parameters. We also link the property to the object, alongside the public value data from the sensor. As we can see in the code, we also make sure to instantiate the value to 0 as seen in line 95 to prevent errors.

```
88          JSONObject temperatureProperty = new JSONObject();
89          temperatureProperty.put("@type", "TemperatureProperty");
90          temperatureProperty.put("title", "Temperature");
91          temperatureProperty.put("type", "number");
92          temperatureProperty.put("description", "The current
                temperature in celsius");
93          temperatureProperty.put("unit", "degree celsius");
94          temperatureProperty.put("readOnly", true);
95          temperatureLevel = new Value<>(0.0);
96          thing.addProperty(new Property(thing, "Temperature",
                temperatureLevel, temperatureProperty));
```

Listing 6.8: Adding properties to the thing descriptor in WebThings

In this prototype, we add random numbers to simulate how its possible to provide the data, as seen in listing 6.9. This function is currently just a placeholder, but if the prototype should be used, then this is where one adds the function to retrieve the data from the actual sensor.

```
166     private static double readTemperature() {
167         return Math.abs(70.0d * Math.random() * (-0.5 + Math.random
                ()));
168     }
```

Listing 6.9: Updating properties for the thing descriptor in WebThings

The implementation of adding events to the thing object works the same way as the adding properties as seen in listing 6.10. The only difference being that we do not assign any value to the event, as it will be emitted once a condition is satisfied. This means we can add the condition in any other function, and emit it alongside updating the property value.

```java
        //Event
        JSONObject gasLeakageProperty = new JSONObject();
        gasLeakageProperty.put("description",
                "Event for GasLeakage");
        gasLeakageProperty.put("type", "boolean");
        thing.addAvailableEvent("GasLeakage", gasLeakageProperty);
```

Listing 6.10: Adding events to the thing descriptor in WebThings

Once we have added all the necessary properties and events to our things, we can start the things functions as seen in listing 6.11. This is done by starting a thread in the object, which fetches the new data every 3 seconds. Once the data has been fetched, we also have to make sure to notify the thing to emit an update by calling the notifyOfExternalUpdate function, to change the value in the exposed thing. We can also see how the process emitting events work, as we add a condition and update the event with a value.

```java
        // Start a thread that polls the sensor reading every 3
            seconds
        new Thread(() -> {
            while (true) {
                try {
                    Thread.sleep(3000);
                    // Update the underlying value, which in turn
                        notifies
                    // all listeners
                    double newLevel = readCO2();
                    double newGravity = readGravity();
                    double newpH = readpH();
                    double newTemperature = readTemperature();
                    double newSalinity = readSalinity();
                    double newTurbitity = readTurbidity();

                    //Event handler
                    if (CO2level.get() < 20) {
                        thing.addEvent(new GasLeakageEvent(thing,
                            true));
                    }

                    //Update Value
                    CO2level.notifyOfExternalUpdate(newLevel);
                    gravityLevel.notifyOfExternalUpdate(newGravity)
                        ;
                    pHLevel.notifyOfExternalUpdate(newpH);
                    temperatureLevel.notifyOfExternalUpdate(
                        newTemperature);
                    salinityLevel.notifyOfExternalUpdate(
                        newSalinity);
                    turbidityLevel.notifyOfExternalUpdate(
                        newTurbitity);

                } catch (InterruptedException e) {
                    throw new IllegalStateException(e);
                }
            }
        }).start();
        return thing;
    }
```

Listing 6.11: Starting the thing in WebThings

Mozilla Gateway

We now consider the gateway application of the prototype. As mentioned earlier, this step only requires configuration of the gateway for the WebThings framework to work with it. We will explain in chapter 6.2 how to set up the gateway.

First step is to make sure we have the WebThings add-on installed on the gateway. To do this, we have to go to the add-on section of the gateway, and search for the add-on. By adding it, it will configure the gateway automatically to fetch WebThings devices and use them. If done correctly, the add-on menu should look like figure 6.4. We can also add other add-ons to support our devices in this menu, as Mozilla provides many add-ons both for developers and non-developers.

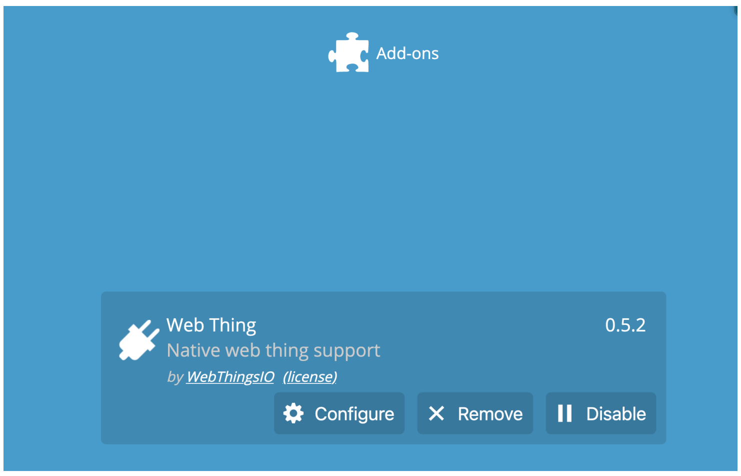


Figure 6.4: Mozilla Gateway Addon screen

After this process has been completed, we can go back to the sensors menu of the gateway and start adding the sensors. By clicking the "add new device" button, the gateway will automatically start using mDNS to locate any things on the network, as seen in figure 6.5. But if the sensors are not located on the network, then we have to add them manually by clicking "Add by URL..." and get the IP address from the device running the ThingWeb framework.

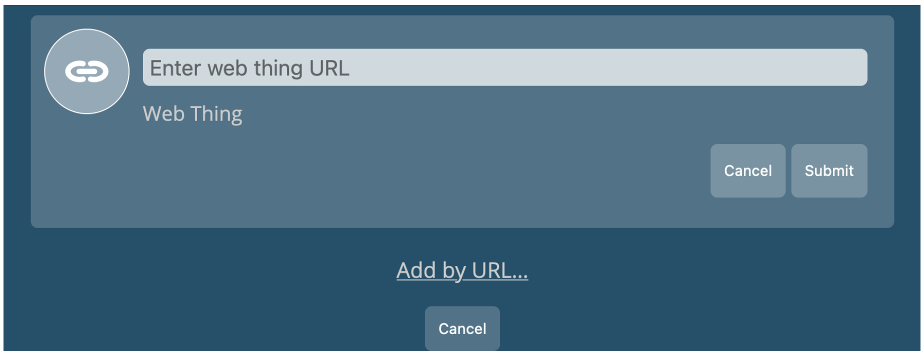


Figure 6.5: Mozilla Gateway using mDNS

If the sensors are located on the same network, it will show the four things we created from the ThingWeb framework as seen in figure 6.6. From here we can rename the things and change the property to Multi Level Sensor or Custom Thing. By clicking the save button, we will have access to use the devices in our prototype.

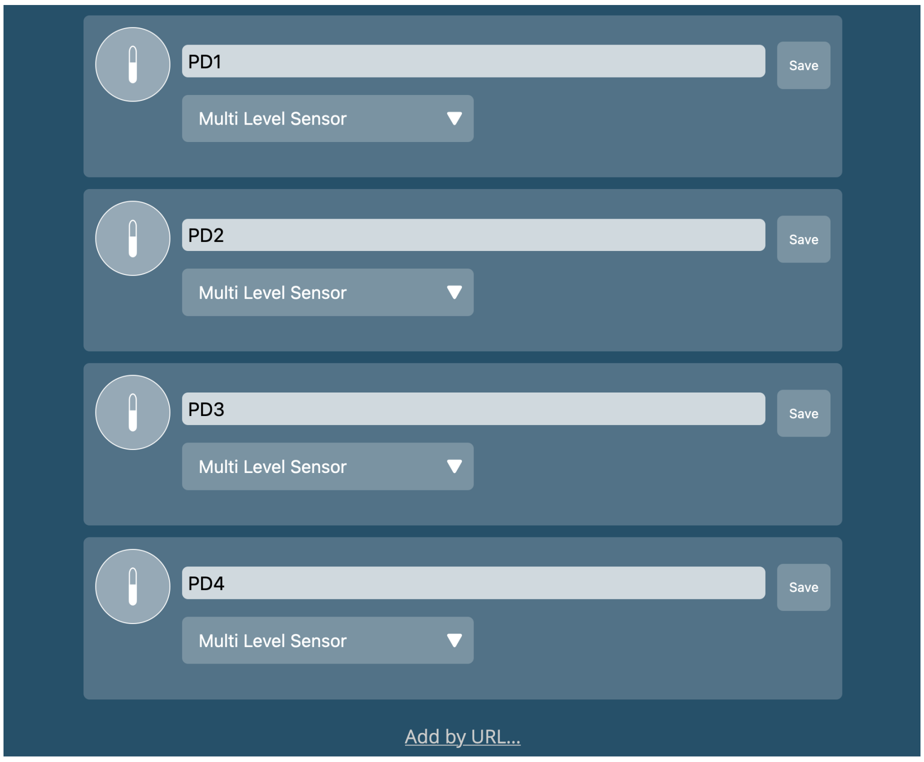


Figure 6.6: Mozilla Gateway found the things by mDNS

From here, we can start using the gateway to add rules to the sensors properties, adding them to a custom map and start logging specific properties. The final touch to make the prototype accessible from the World Wide Web is to set a specific domain. We head over to the domain menu in the gateway, and get a menu like the one in figure 6.7. Here we can set a local URL to be accessed on the same network, and a remote access URL to accessed from anywhere in the world.

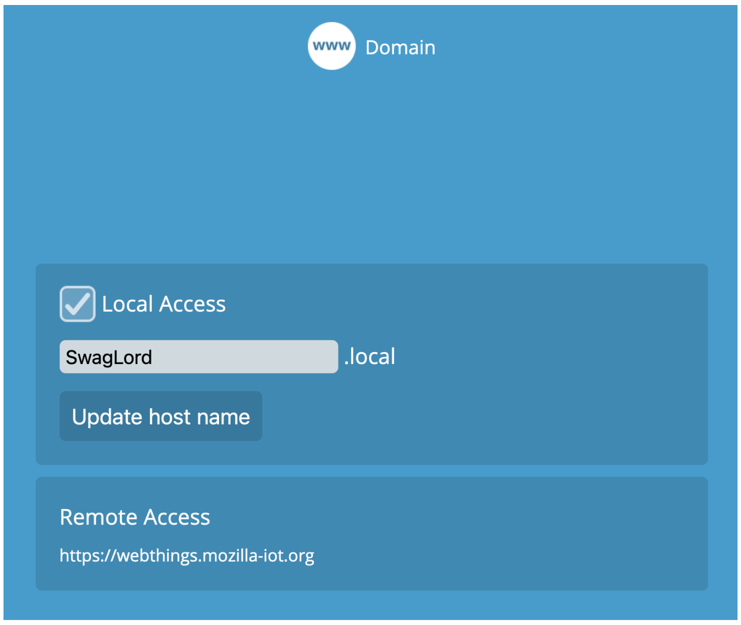


Figure 6.7: Mozilla Gateway domain menu

6.2 Deployment

We have now covered how the prototypes are implemented based on the frameworks. In this section we explain how to deploy the prototypes. The tutorial for deployment is written for Linux and macOS, as Windows has a different way of deployment. The optimal prototype would be to run a virtual machine or docker on Windows to get the prototypes working.

6.2.1 Prototype 1 - Node-RED and Eclipse Thingweb

We start by explaining how to deploy the first prototype. Follow the following steps to set up the prototype on your own device;

1. Start by going to the appendix B and download the ThingWeb code and Node-RED json export file.
2. Install the latest version of Node.js by typing the following commands in a terminal;

```
$ curl -sL https://deb.nodesource.com/setup_10.x | sudo -E
    bash -
$ sudo apt-get install -y nodejs
```

3. Locate the ThingWeb project on your machine and type the following to install the dependencies, build the project and run the code;

```
$ npm install
$ npm run build
$ npm run start
```

4. Install the Node-RED framework on your device by typing the following into a terminal;

```
$ sudo npm install -g --unsafe-perm node-red
```

5. Install the node-red-contrib-web-of-things extension for the Node-RED framework by typing the following into a terminal;

```
$ npm install node-red-contrib-web-of-things
```

6. Start the Node-RED framework by typing the following into a terminal

```
$ node-red
```

7. Open up a browser and head to the IP address `http://localhost:1880`.

8. Go to the dropdown menu in the Node-RED framework and click on import flows. From here, import the JSON export file downloaded earlier.

6.2.2 Prototype 2 - Mozilla WebThings and Gateway

For the second prototype, we have to set up a Java environment to get the WebThings framework to run. If we follow these steps, one will get the same prototype as seen in this thesis;

1. Start by going to the appendix B and download the WebThings code.

2. Install a JDK from a official site [13].

3. Head into the project and locate the Pilot Demonstrators.java in a terminal and run the following code;

```
$ javac PilotDemonstrators.java
```

4. Run the project by running the following code in terminal;

```
$ java PilotDemonstrators.java
```

5. Download the Gateway application WebThings github [45] and follow the instructions on the GitHub to get it running.

6. Open up a browser and head to `https://localhost:4443`, and follow the instructions for creating your Gateway

7. Head to the Gateway menu and head to the add-ons section. Download the WebThings addon.

8. Head back to the main page at the Gateway, and it should locate the devices.

Chapter 7

Evaluation

In this chapter, we cover the different evaluation criteria needed to answer the research questions introduced in chapter 1.2.1. As we have created two proof of concept prototypes without the possibility of comparing to existing solutions with benchmarks, this means that we have to find qualitative criteria to evaluate the technology.

7.1 Evaluation Criteria

Before we can evaluate the technology, we have to define the criteria to be used for evaluation of the prototypes. In this section, we explain which criteria we decided to use, and how we will use them to undertake the evaluation. The process will involve a scoring system between one to five, where five is the best score while one is the lowest, to get a final evaluation of both the prototypes. The summary score in this chapter is a score based on our experience during development with the frameworks. Our evaluation criteria is based on the evaluation from the ISO/IEC 25010 standard which is the system and software quality requirements and evaluation standard [16]. It provides a quality active model with different categories for analysing software quality. We have selected three categories which will become useful for analysing the frameworks in this thesis.

7.1.1 Maturity

This criteria evolves around the maturity of the frameworks. This is one of the three categories provided by the ISO 25010 standard and considers to which degree the component meets the needs for reliability under normal operation. DZone gives a good tutorial on which factors to look at to measure the maturity of software [28], which will be used for the evaluation. We will be using its software maturity model to analyze the frameworks used for the prototypes. This means giving a subjective judgement on the frameworks, based on the information they provide. We will also be going trough the World Wide Web to figure out how many people are actually talking about the frameworks, and estimate how many people that actually use the frameworks. Another important aspect of analysing the frameworks will be to check how often they update the frameworks, and how long time different features take to be implemented. We will also be doing an estimation on how many people work on the frameworks, to assess how popular the frameworks are relative to each other.

7.1.2 Documentation

The documentation criteria evolves around the information the framework provides for the developer. Here, we will analyze what kind of documentation each framework provides, how difficult it is to follow and what the developer has to do to use the framework. We will also be covering how much the developer has to do outside the documentation to get the different frameworks to work, as well as comments in the framework code.

7.1.3 Amount of code

In this criteria, we consider the amount code that needs to be written by the developer. Here we will analyze the different frameworks helping functions, and how much code the frameworks saves the developer from writing compared to if the developer had to code the solution from scratch. This criteria also evolves around how much code is needed to get the different frameworks to work. As two of the frameworks; Node-RED and Mozilla Gateway requires, no code for their usage, this step will not be considered for these frameworks as they wont be useful evaluating them for this criteria.

7.1.4 Learnability

Here we will be evaluating the learning curve of the frameworks, and what one needs to learn to eventually use the frameworks. This is also one of the categories provided by the ISO 25010 standard and assess to which degree a system can be developed by the developer to achieve its goal of learning to be used. We will be discussing this criteria alongside the documentation criteria and how the frameworks works to evaluate how difficult it would be for a new developer to start using the framework. We also need to involve the WoT technology here, as we will be using it alongside the frameworks.

7.1.5 Automation

In this criteria we will evaluate how much automation the framework is able to provide. This can be anything from functions to automatically taking care of some action, to how easy the build process for the framework is. We will also go into details on the level of automation which the different frameworks provides.

7.1.6 Security

Here, we will be evaluating the security of the different frameworks. This is one of the categories provided from the ISO 25010 standard and consider to which degree the system protects the information and data. Here, we will analyze which known security holes each framework has based on the top ten OWASP security vulnerabilities [27]. This means using third party software like SonarQube and OWASP ZAP to analyse the frameworks in the prototype and get into details on where different security holes could be located, as finding the vulnerabilities ourselves would take too long time. SonarQube analyses the code for vulnerabilities and known bugs which could be exploited. While OWASP ZAP takes care of analysing the browser applications for known security issues. We will also be covering security issues found during development and usage of the frameworks. The size of the smart ocean project is large, which makes this criteria quite important for evaluating if the frameworks should be used or not.

7.2 Evaluation of frameworks

In this section we only cover the frameworks used for our prototypes, as the rest has not been touched in this thesis. This will involve the four frameworks Mozilla WebThings, Mozilla Gateway, Eclipse ThingWeb and Node-RED.

7.2.1 Eclipse ThingWeb

First out is the Eclipse ThingWeb framework. As this framework demanded code, this means we will be analysing the amount of code needed to get the framework to work.

Maturity

We start by evaluating the Eclipse Thingweb by going through the criteria explained in the section 7.1. The first criteria to be evaluated is the maturity of the framework. We can easily see by going through the GitHub repository of the framework that it is not as big as we would expect for being the framework that closest follows W3C's definition of WoT. The framework has gone through 16 releases, with the first release in 2018. It is still being updated regularly, with the latest update released in March 2021. It does not have the biggest numbers of contributors for being an open-source framework, with only 21 contributors. For being one of the leading frameworks within WoT, having such small numbers seems to indicate a low maturity for the framework. We can further see this by looking at the amount of people talking about the framework on one of the most popular site for developers, Stack Overflow [34]. With a google search we only get about eight pages, which means that there is close to none forum posts about people discussing bugs with the framework, or how to use it.

Given the information gathered and how much popularity the framework has gotten over the years, means its maturity is still on the weak side. It neither looks like the frameworks maturity and popularity will increase a lot in the coming years. For this reason, we give the framework a two out of five in maturity rating.

Maturity Score

●●○○○

Documentation

The next criteria we will be looking at is the documentation of the framework. The framework has little to none documentation by itself, and references itself to the W3Cs definition of WoT. This means that the framework uses the definition from W3C as documentation for the development of the framework. This creates some issues as W3C does not cover everything the ThingWeb framework does or how to use the framework itself. For example, how to implement different binding-templates. W3C does not cover this in technical details, which means we have to go through the examples from the ThingWeb framework in order to understand how the framework is technically used. The idea of using the W3C as documentation is not a bad idea as it gets quite technical defining the different building blocks for the WoT technology. But problems occur once we get a little bit more technical than what W3C intended their documentation to cover. Based on the information out there, we give the framework a two out of five.

Documentation Score

● ● ○ ○ ○

Amount of code

For the third criteria, we will be looking at the amount of code needed to get the framework to work. From the implementation of the framework in subsection 6.1.1, we get the general idea on how much code is needed to get a basic solution working. The framework takes care of most of the functionalities needed to get a WoT produced thing descriptor working, with just few new code lines. Most of the code comes from the implementation of a thing descriptor, where we have to go through the definition created by W3C, and create one according to its scheme. The framework then takes care of the rest. It also converts TypeScript to JavaScript, which decreases the amount of code which is needed, as the compiler makes sure the code is converted correctly. If we look at the framework used in the prototype, we can see that pilot demonstrator one only required 129 lines of code. Based on this, we conclude that the amount of code to create an application with the framework gets the score four out of five.

Amount of code Score

● ● ● ● ○

Learnability

The learnability of the framework works alongside the second criteria of documentation. As the documentation is not properly made, this means that the learning curve is a little more steep than what is necessary for the framework. The framework is almost straight forward from the examples the framework provide. But it also means that the developer needs to have some background knowledge on TypeScript to understand the core functionalities of the framework, and how the different functions work alongside each other. The learning curve also increases based on how mature the framework is, as there is few people out there who can help new developers. For this reason, we give the learnability criteria an two out of five, as its not very friendly towards new developers.

Learnability Score

● ● ○ ○ ○

Automation

There is little automation provided by the framework except from the servient automatically exposing a thing descriptor once its made. We can in a way say the framework converts TypeScript to JavaScript also is an automation process. Because of this, we will give the framework only three out of five on this criteria.

Automation Score

● ● ● ○ ○

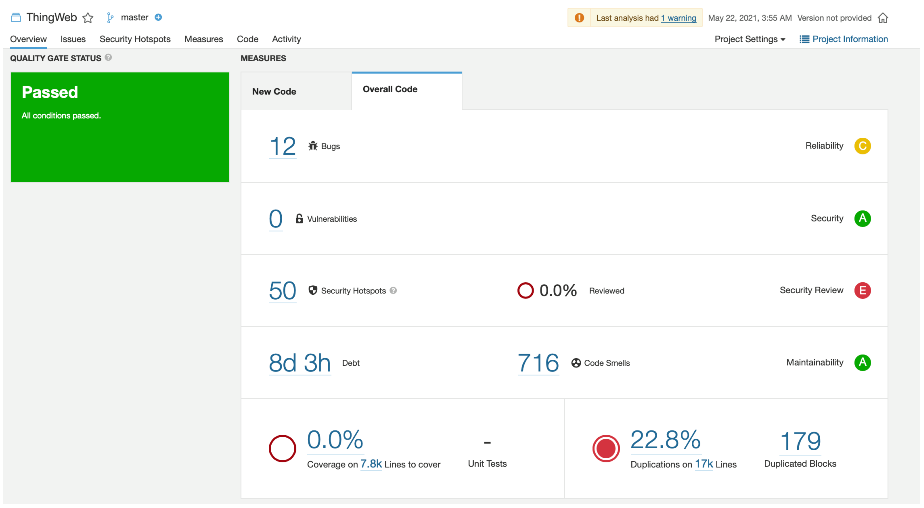
Security

This framework provides no user interface through a browser which means we can omit the usage of browser when analyzing the security on the framework. We therefore shift our focus to bugs in the code and bad code created by the developers. If we scan the latest version of the framework with a third party software like SonarQube, we can see how much work the framework needs to be perfected. Along this, the framework supports the possibility for adding security to the thing descriptor. This means that if the thing needs to be accessed publically, we can add authentication directly to the thing descriptor, making security through data flow safer and customisable.

As we can see in figure 7.1, the framework is close to perfection with no critical vulnerabilities detected in the code. It has a few common bugs and some code smells which could easily be fixed and which does not cause any critical vulnerabilities. The fact that the code has no vulnerabilities found by SonarQube means it is quite stable compared to a lot of frameworks made. The security hotspot is just code which the developer needs to make sure does not leak any sensitive information to the public. By going through the code ourselves, we can see there is nothing critical to comment on. As for this, we give the framework a four out of five on the security aspect of the evaluation.

Security Score

● ● ● ● ○


ThingWeb master

Last analysis had 1 warning May 22, 2021, 3:55 AM Version not provided

Overview Issues Security Hotspots Measures Code Activity

Project Settings Project Information

QUALITY GATE STATUS

Passed

All conditions passed.

MEASURES

New Code | Overall Code

12 Bugs			Reliability C
0 Vulnerabilities			Security A
50 Security Hotspots	0.0% Reviewed		Security Review E
8d 3h Debt	716 Code Smells		Maintainability A
0.0% Coverage on 7.8k Lines to cover	- Unit Tests	22.8% Duplications on 17k Lines	179 Duplicated Blocks

Figure 7.1: SonarQube scan of the ThingWeb framework

Summary

Based on the earlier criteria, we can conclude the framework is not the most useful one out there. In particular, it does not have a lot of popularity for development. But the positive thing is that the developers who are updating the framework manage to keep the framework quite stable. Based on this information, we conclude the framework gets an overall score of three out of five.

Summary Score

● ● ● ○ ○

7.2.2 Node-RED

The second framework to be analysed is the Node-RED framework. As this framework provides a browser dashboard for the user means, we will be using OWASP ZAP to scan for vulnerabilities.

Maturity

Node-RED is a more mature framework than Eclipse ThingWeb. The framework got 119 different released versions on GitHub, with the latest being from May 2021. The project is being worked on by over 100 people at its open source GitHub repository. The framework also got over 1900 tags on Stack Overflow for different problems developers have encountered. But if we were to see how the framework handled WoT, then the process would be a little different. In order to get the framework to support the technology, we have to use external extensions alongside the framework. The external framework only has four contributors, with no official versions released for the public. As this extension is needed to get the technology to work with the framework, we also have to consider it as a part of the evaluation. Because of this, we decided to give the framework and extension a maturity level of three out of five.

Maturity Score

● ● ● ○ ○

Documentation

Just like the last evaluation criteria, we have to evaluate both the extension and the framework as they are both needed to make the WoT technology work. First we start by looking at the documentation for the Node-RED framework. The Node-RED framework has plenty of tutorials, YouTube guides, example project and installation tutorials for multiple platforms. Every single node has a detailed explanation to them and how to use them, and they even have frequently asked questions on how one can use the different nodes. The documentation is written in a way which is easy to go through. Compared to the rest of the frameworks looked at in this thesis, this may be the best written documentation.

The extension is quite minimalistic in its documentation. It only provides some simple images on how to use the external nodes, and some tips on how to use the extension. It only covers the bare minimum to get the extension to work, which is sufficient for its purpose. All in all, if we look at the documentation of the framework and the extension combined, its safe to say it at least deserve a four out of five on documentation level.

Documentation Score

● ● ● ● ○

Learnability

As the documentation is fearly well written, this means the only thing one has to figure out is how one wants to use the framework. The framework demands no code, only the knowledge on how to get the application started, on which there is plenty of documentation. This is positive, as it means the developer can focus on learning ThingWeb and not spending a lot of time trying to get a second framework to work. One still has to understand how to code on a minimum level to use Node-RED, which means basic understanding of if-cases, rules and variables. Beyond this, the framework has one of the simplest learning curves. And because of this, we give the framework a five out five in learnability.

Learnability Score

● ● ● ● ●

Automation

The framework itself takes care of automating most of the things with its nodes. The only thing we have to configure to get it to work is the configuration of the nodes. The extension also adds a new level of automatisation of the framework, where it automatically fetches the exposed thing descriptor and consumes it for the developer. It then automatically shows all the functionalities the thing has, and gives the user the option to choose which one to use. The framework itself is made for data flow, which means its core purpose is to simplify a lot of actions by automatisation. Because of this, we give the framework five out of five for automation.

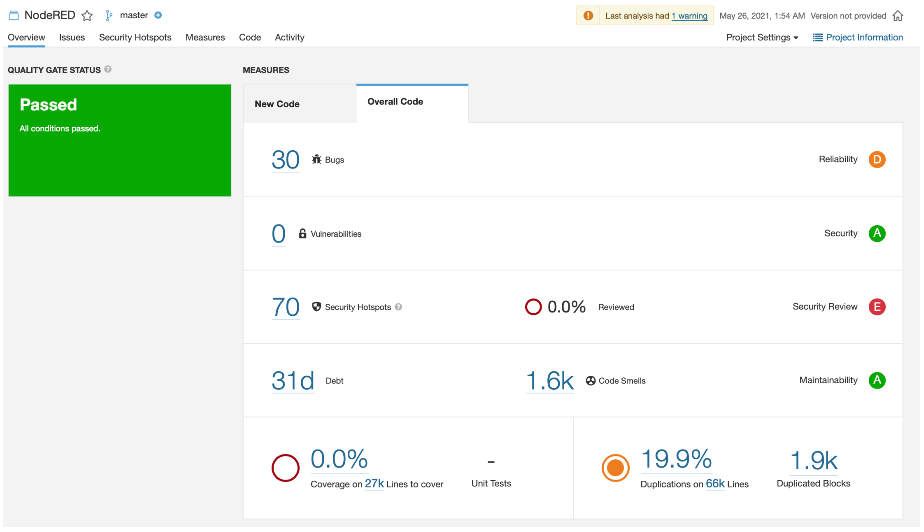
Automation Score

● ● ● ● ●

Security

The framework itself can be deployed locally on a computer or to the cloud. This means that security in the browser is not too important if ran locally on a private network and being unavailable for the public. But once we decide to open this service to access it from outside the local network, we need to start thinking about the security of the software. This means we have to analyse the framework to check for any critical bugs or errors which could lead to data leakage or remote access. There is no authentication to access the dashboard, which means that once one has access to the dashboard, one has access to all the data it provides. This could lead to some serious data leakage.

If we take a closer look at the framework with SonarQube as seen in figure 7.2, we can see that the framework itself is pretty stable with few bugs. The bugs given are just simple developer bugs which would not cause any critical security errors. The framework itself is getting a D from the SonarQube framework, but by looking through these, it would not cause any huge issues based on what the Node-RED framework will be used for. The reason why the score is so low, is based on how critical the framework evaluates the bugs found. The software should never have any bugs to begin with, but none of the bugs found would make a huge impact on the overall security of the framework.


NodeRED master

Last analysis had 1 warning May 26, 2021, 1:54 AM Version not provided

Overview Issues Security Hotspots Measures Code Activity

Project Settings Project Information

QUALITY GATE STATUS

Passed

All conditions passed.

MEASURES

New Code Overall Code

30 Bugs Reliability D

0 Vulnerabilities Security A

70 Security Hotspots 0.0% Reviewed Security Review E

31d Debt 1.6k Code Smells Maintainability A

0.0% Coverage on 27k Lines to cover

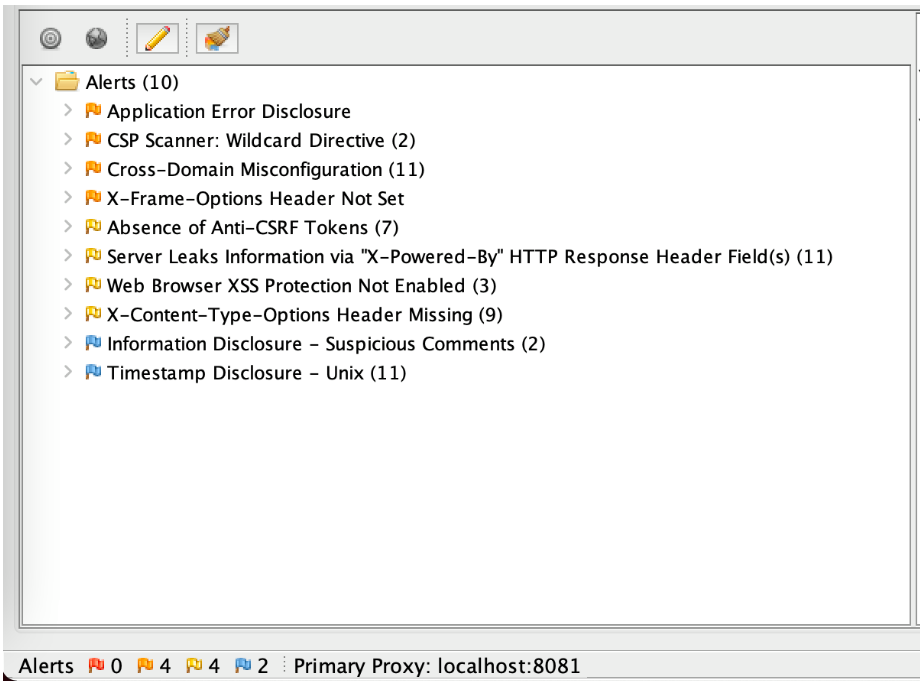
- Unit Tests

19.9% Duplications on 66k Lines

1.9k Duplicated Blocks

Figure 7.2: SonarQube scan of the Node-RED framework

As mentioned earlier, we also have to do a OWASP ZAP scan on the framework as seen in figure 7.3, to investigate if the web solution itself could cause any critical security errors if retrieved. It gives out some errors where the framework itself is not securely made to prevent cross site scripting attacks. But if the attacker has retrieved the dashboard, a cross site scripting attack would not be too critical, as the attacker would already have access to all its data.


Alerts (10)
- Application Error Disclosure
- CSP Scanner: Wildcard Directive (2)
- Cross-Domain Misconfiguration (11)
- X-Frame-Options Header Not Set
- Absence of Anti-CSRF Tokens (7)
- Server Leaks Information via "X-Powered-By" HTTP Response Header Field(s) (11)
- Web Browser XSS Protection Not Enabled (3)
- X-Content-Type-Options Header Missing (9)
- Information Disclosure - Suspicious Comments (2)
- Timestamp Disclosure - Unix (11)

Alerts 0 4 4 2 Primary Proxy: localhost:8081

Figure 7.3: OWASP ZAP scan of the Node-RED dashboard

Because of these findings from OWASP ZAP and SonarQube, we give the framework a three out of five while evaluating the security aspect of the framework. The main reason is that there has been no countermeasures to prevent any attackers from doing malicious things once they reach the dashboard. This could be fixed by making the user authenticate themselves before entering the dashboard.

Security Score

● ● ● ○ ○

Summary

The framework itself is pretty well made for its intention, and the support of the WoT extension makes it even more suitable for the technology. The framework does its job quite well, and is the most popular framework we have been looking into in this thesis. When it comes to the security aspect of the framework, it need some additional work, especially if it is going to be used for the smart ocean framework. Because of this, the framework gets a score of four out of five.

Summary Score

● ● ● ● ○

7.2.3 Mozilla WebThings

The third on the list is the Mozilla WebThings framework. This framework is the extension provided by Mozilla to get the WoT technology to work with a Mozilla Gateway.

Maturity

The maturity of Mozilla WebThings is a little weak compared to the other frameworks we have discussed. The framework is split into multiple programming languages where each has their own level of maturity. Because of this, we will only be looking at the framework used in this thesis, which is the Java framework implementation. The framework itself got 20 official releases on their GitHub page with 7 different contributors. The framework got no tags on Stack Overflow, which means finding people who has the same issue as we, was not the simplest process. The framework also has some bugs which takes a while for the developers to fix. This means we have to customize the framework ourselves for different purposes. The framework is made for smart houses, and are not robust for creating custom solutions like the one in this thesis. Because of this, we give the framework a one out of five in maturity evaluation.

Maturity Score

● ○ ○ ○ ○

Documentation

The documentation for this framework it is almost non-existent. The only documentation we were able to find, is the information from the GitHub repository. The problem here is that it only gives small examples on how the framework is used, which leads to the developer needing to do a lot of research of the code to make it work. This becomes a problem when trying to create custom solutions, as they only provide the core information to get it to work for smart houses. Only two examples on how to use the framework is provided, which makes it hard to create applications that deviates significantly from the examples. The code also has little to none documentation for its functions. This means its a simple step to evaluate the documentation of the framework, which only gets one out of five in the documentation criteria.

Documentation Score

● ○ ○ ○ ○

Amount of code

The framework provides all the classes and objects needed, which means we only have to implement a limited amount of code to get an application working. We needed exactly 114 lines of code to get the first pilot demonstrator to work with the framework, and this includes some boilerplate code. The classes works as intended, and are easy to use alongside a developer tool. Beyond this, we only have to create functions to retrieve the data from the sensors. As Java does not have the same functionalities with external packages like TypeScript, means that this process will demand a little more code than what the ThingWeb framework would need. Because of this, we give the framework a three out of five for the amount of code needed to get it to work.

Amount of code Score

● ● ● ○ ○

Learnability

As mentioned in the documentation criteria, the framework itself does not have any proper documentation. This means we have to learn everything from scratch to create custom things for the Mozilla Gateway to use. There are two examples to learn from, which makes the process at least a little easier. But the fact that the framework has so few comments for their functions as well, makes it hard to learn. The only reason one would decide to use this framework would be because it is written in Java, which a lot of developers tends to have some knowledge about. Because of this, we give the framework one out of five for the learnability criteria.

Learnability Score

● ○ ○ ○ ○

Automation

The sole purpose of this framework is to automatically create a produced thing descriptor which can be consumed by the gateway. For this reason, the framework takes care of exposing the thing once the thing descriptor has been obtained and connected to properties to provide information. The framework then takes care of automatically updating the endpoints on the exposed thing descriptor with the necessary data. Because of this, we give the framework a three out of five for automatising.

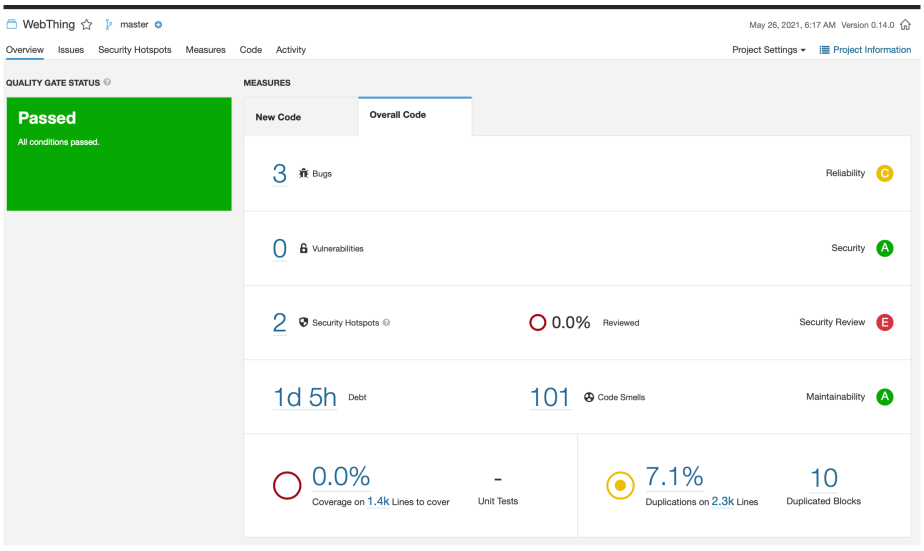
Automation Score

● ● ● ○ ○

Security

As the framework only exposes the thing descriptors made, which means we do not have to consider browser exploitation with OWASP ZAP. The primary problem with the framework is the fact that we cannot give the things any security as ThingWeb. This means that the thing will be public accessible if we do not add any external security to the framework, except the HTTPS protocol. This can be a security problem, if being overlooked in the final solution, as HTTPS alone is not sufficient to ensure that data is confidential.

From the SonarQube scan of the framework seen in figure 7.4, shows that the framework only has three small bugs, which does not provide any major security alerts. For being the framework with the least maturity, it still provides a solution without any critical vulnerabilities.



WebThing master

May 26, 2021, 6:17 AM Version 0.14.0

Overview Issues Security Hotspots Measures Code Activity

Project Settings Project Information

QUALITY GATE STATUS

Passed

All conditions passed.

MEASURES

New Code | Overall Code

3 Bugs — Reliability C

0 Vulnerabilities — Security A

2 Security Hotspots — 0.0% Reviewed — Security Review E

1d 5h Debt — 101 Code Smells — Maintainability A

0.0% Coverage on 1.4k Lines to cover — - Unit Tests

7.1% Duplications on 2.3k Lines — 10 Duplicated Blocks

Figure 7.4: SonarQube scan of the WebThings framework

Based on what we obtained, we provide the framework with a score of two out of five at security criteria.

Security Score

● ● ○ ○ ○

Summary

All in all, the framework itself is the only way to get the WoT technology to work with the Mozillas Gateway. Even though it is not the most mature, easy to learn or best documentation, it still does its job with providing the technology to Mozillas Gateway. But there is also a few issues that comes along with the usage of this framework as discussed in the design of the prototype. As long as the framework continues to be developed, it may one day reach a stage where it can be used without all the issues. We give the framework an overall score of two out of five.

Summary Score

● ● ○ ○ ○

7.2.4 Mozilla Gateway

The last framework considered is the Mozilla Gateway. Just as the Node-RED framework, Mozilla gateway provides a dashboard which has to be analysed with OWASP ZAP. But as its a standalone application it does not need any code to function. This means that we will exclude the amount of code criteria.

Maturity

The Mozilla gateway has been in the development since 2017 and has over 26 official releases. It has been contributed to by 123 people. The framework itself has gotten a lot of popularity from different blogs and is one of the more known frameworks out there for smart house development. There also exists plenty of tutorials on how to use the frameworks by both Mozilla themselves and from external people on YouTube. The gateway has gotten significant support from external developers who has created custom add-ons for the framework. In 2019, there was around 97 add-ons created by external developers and Mozilla. As Mozilla is quite well known, this also helps with the popularity of the framework itself. Because of this, we give the framework five out of five in maturity.

Maturity Score

● ● ● ● ●

Documentation

For the documentation of the Mozilla Gateway, Mozilla has added detailed documentation with pictures for everything from user guides to developer guides. They even provide detailed documentation for creating custom add-ons. It also gives detailed explanation on how to deploy the gateway on the cloud. For this thesis, we only looked through the guide for setting up a device with Mozilla Gateway, as we only needed it to be available to use it with Mozilla WebThings. It lacks some documentation for developers on how to customise the functionalities that the gateway provides, like where the data logs are saved. Beyond this, it provides all the information needed both for developers and a common user with sensors. For this, we give the framework a four out of five in documentation.

Documentation Score

● ● ● ● ○

Learnability

As the framework provides detailed documentation on how to set up the framework, even for a standard developer, it is the framework looked at in this thesis having the best learnability. One just need to follow Mozillas picture guides for setting up the gateway, and it provides a wizard for setting it up for the users needs. As it is the simplest framework to get going, we give it five out of five in learnability.

Learnability Score

● ● ● ● ●

Automation

First thing we can address is the fact the framework automatically finds the sensors created from WebThings on the network. It also automatically provides a database with visual representation for the user on the dashboard. In addition to this, it automatically sets up an AWS server to provide a log in for the user, once the gateway has been created. The framework basically sets up everything one would need for creating a smart home, with simple installations. Because of this, we give the framework five out of five in automation.

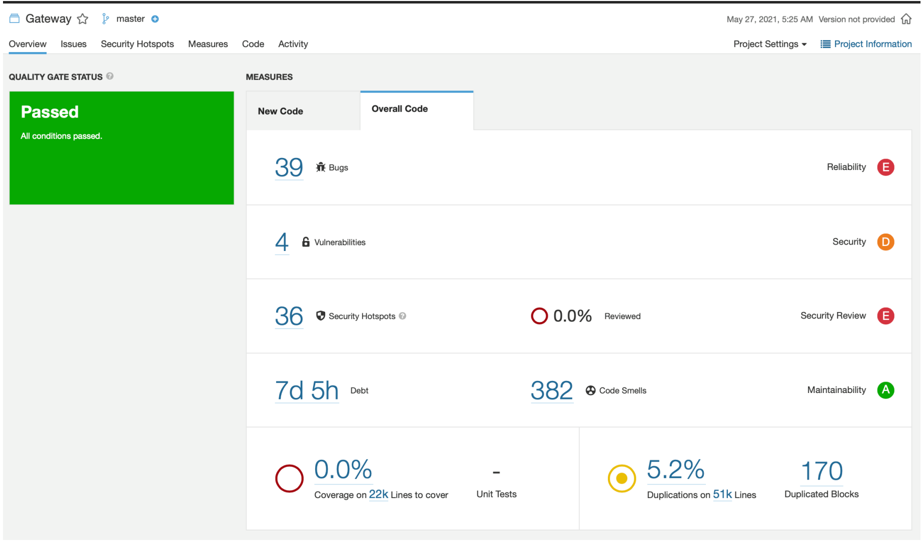
Automation Score

● ● ● ● ●

Security

The framework is automatically set up on the public network for the user on installation. But instead of being available to be accessed by everyone like the Node-RED framework, it sets up an authentication for the user to log in. This authentication is just a single factor authentication, which means there is no external confirmation to log in like a phone code. Hence technically attackers could brute force their way into the framework if they really wanted to access it. Another problem comes alongside their choice of moving the security to the HTTPS protocol. Currently, there are no measurements in the solution to prevent attacks like man-in-the-middle attacks, where an attacker can act as the gateway and get all the confidential information from the sensors. The solution to this would be to add a tunneling for remote access to the gateway which would require a lot of maintenance and work, which Mozilla lacks support for.

If we take a closer look at the SonarQube scan for the framework as seen in figure 7.5, we can see that this may be the most insecure framework we looked at. It got four vulnerabilities in the code, which leads to cross site request forgery attacks. It also encrypts the session token with a very weak encryption, making it easier for an attacker to steal the session and log in as the user. In addition, the framework has a few of bugs, even bugs that SonarQube consider critical bugs which has not been seen from the other frameworks. The SonarQube scan identifies some functions in the framework that does not work with the code provided, as well as return functions which returns nothing. Furthermore, there is a significant amount of small and major bugs which SonarQube picked up, which could easily break the framework. For a framework with this maturity, having all these bugs should not even be possible. This means all the people who uses the framework are prone to attack, which in this case is a lot of users.


Gateway master

May 27, 2021, 5:25 AM Version not provided

Overview Issues Security Hotspots Measures Code Activity

Project Settings Project Information

QUALITY GATE STATUS

Passed

All conditions passed.

MEASURES

New Code | Overall Code

39 Bugs — Reliability E

4 Vulnerabilities — Security D

36 Security Hotspots — 0.0% Reviewed — Security Review E

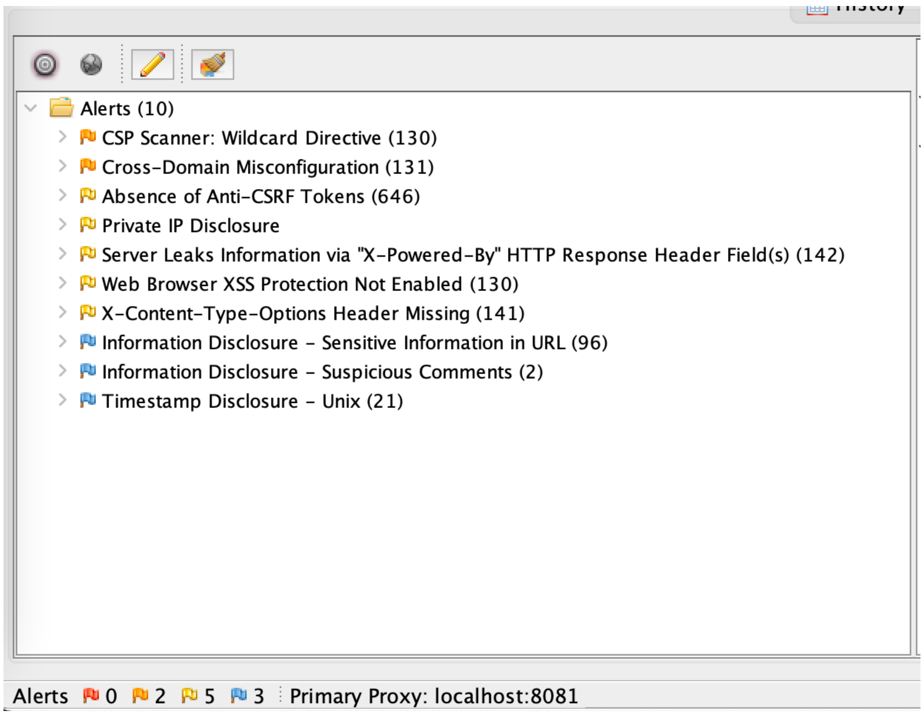
7d 5h Debt — 382 Code Smells — Maintainability A

0.0% Coverage on 22k Lines to cover — - Unit Tests

5.2% Duplications on 51k Lines — 170 Duplicated Blocks

Figure 7.5: SonarQube scan of the Gateway

As this is a browser framework, we also have to do some security checks for the browser. If we look at figure 7.6, which is the OWASP ZAP scan for the gateway, we can see all the bugs from the SonarQube in action. It provide tons of dead links in both its code and scripts. It also complains about the cross site configuration not being properly set, meaning an attacker could use third party APIs to read the information from the gateway as an authenticated user. It also shows that the private IP address has been leaked through the gateway, which means that once the attacker know the public address for the user, they can start abusing the gateway, or the computer hosting the site. Also, it identifies that cross site request forgery attacks could be done towards the site, as it has not properly set its configuration against it.



```
Alerts (10)
  CSP Scanner: Wildcard Directive (130)
  Cross-Domain Misconfiguration (131)
  Absence of Anti-CSRF Tokens (646)
  Private IP Disclosure
  Server Leaks Information via "X-Powered-By" HTTP Response Header Field(s) (142)
  Web Browser XSS Protection Not Enabled (130)
  X-Content-Type-Options Header Missing (141)
  Information Disclosure - Sensitive Information in URL (96)
  Information Disclosure - Suspicious Comments (2)
  Timestamp Disclosure - Unix (21)

Alerts 0 2 5 3   Primary Proxy: localhost:8081
```

Figure 7.6: OWASP ZAP scan of the Mozilla Gateway

The fact that this framework tries to make some security effort against attackers, at the same time as having these bugs means that the developers who contributed to this framework lacked the proper knowledge to create a safe software. If we could find all these vulnerabilities by only using two third party security scans, this means that the framework is very prone to attacks. This could lead to some serious data leakage, which could be critical for the Smart Ocean project. For a common user with a few hobby sensors, this would not be to critical. For this, we give the framework a one out of five for the security, being the worst out of the four we looked into.

Security Score

● ○ ○ ○ ○

Summary

For the summary of the Mozilla Gateway, we can see that the framework has some serious pros and cons. It does its work fairly well for being made for smart houses, but could lead to some serious problems while being used for a Smart Ocean project. For being a framework created by Mozilla itself, this means that this gateway was probably made for experimental purposes, and not for business solutions. We can conclude in the end that the framework gets a three out five in summary score. And we can clearly conclude that the framework itself should not be used for a project like Smart Ocean. As for the usage of WoT, we also need to consider that Mozilla has created their own version of the technology, which does not follow the W3C standard. They have also stated that they will not be following W3C in the future, meaning if one would use this framework because of the technology, there exists better solutions.

Summary Score

● ● ● ○ ○

Chapter 8

Conclusion and Future Work

Throughout this thesis, we have experimentally evaluated four WoT frameworks recommended by W3C. W3C is the leading contributor to the WoT technology, and is responsible for updating the standard and schema. As we have concluded in the evaluation chapter, there is a lot to be done on the frameworks considered for WoT before it should be used for a large scale project like Smart Ocean. In this chapter we summarize what the current state of the technology is, and present directions for future work.

8.1 Summary

In this thesis, we have created two prototypes and evaluated them based on the experiences from the development. The prototypes show how WoT can be implemented and deployed into the Smart Ocean project. There has been some interesting discoveries during development that we summarise in this section. From the evaluation of the frameworks, we can clearly see which frameworks should be used to implement the WoT technology and which should not be used based on the summary score seen in figure 8.1. This is important to address as the creators of the technology itself, W3C, has recommended these frameworks themselves.

As mentioned earlier, the score is based on the experience gathered during development. We can also see the median score from the evaluation in figure 8.2. The only difference being the Mozilla Gateway, which had such big security issues that it brought the summary score down.

Framework	Maturity	Documentation	Amount of code	Learnability	Automation	Security	Summary
Eclipse ThingWeb	2	2	4	2	3	4	3
Node-RED	3	4	N/A	5	5	3	4
Mozilla WebThings	1	1	3	1	3	2	2
Mozilla Gateway	5	4	N/A	5	5	1	3

Table 8.1: Table of the score from the evaluation chapter.

Framework	Score
Eclipse ThingWeb	2.83
Node-RED	4
Mozilla WebThings	1.83
Mozilla Gateway	4

Table 8.2: Average score of the frameworks

We can conclude that the framework which is the best to be used for the technology is the Eclipse ThingWeb framework, as both Node-RED and Mozilla Gateway uses extensions only to be available to consume the thing descriptors. Even though Eclipse ThingWeb has some flaws as we have seen in the evaluation, it is still the leading framework while looking at the web of things technology and considering frameworks investigated in this thesis. The fact that it is the leading framework with the score evaluated means the WoT technology itself may need some work before it should be used for large scale projects like Smart Ocean.

8.2 Research Questions

By going back to the research questions mentioned in chapter 1.2.1, we now have enough research and experience to answer them. Our first research question, R1, is about which frameworks and platforms that exist. While investigating the technology we identified several frameworks. This is then explained in chapter 3, where we discuss which framework that implements the WoT technology. There exist many more, but these were the ones recommended by W3C, and is the reason why they were picked. Our second research question, R2, is about how mature the frameworks are based on the W3C standard. First we had to figure out the standard. W3C emphasizes their four building blocks for defining the WoT standard, which can be found in chapter 2.2. We then evaluated the frameworks as seen in chapter 7 based on this information. Here we assessed the matureness of each framework, and gave an overall score on where the framework stands. Based on the research completed, we can conclude with the frameworks lacking matureness based on the information collected. Our third research question, R3, address how the sensors from Smart Ocean can be represented with the technology. As we have been provided with four pilot demonstrators from Smart Ocean, we replicate them by using the frameworks. In chapter 4, we showcased how the pilot demonstrators will look based on the W3C standard. We also showcased what to do to get the pilot demonstrators to work with the frameworks in chapter 6.1. In our last research question, R4, we ask to what extend the frameworks can be used to implement the case studies. This is also covered in 6.1, where we showcased how the frameworks work, and to what extent they can be used. Here we identified important information on what we can do with the frameworks.

8.3 Conclusion

As mentioned earlier, the technology itself got plenty of work by W3C. W3C defines all the building blocks and provides good documentation on how the technology works. The technology itself and the idea around the technology is well-defined, which means it should, in theory, easily be used for any device driven application without the flaws for the IoT technology. But the current state of the frameworks which is meant to implement the technology implies that the technology needs some work before it should be used. There is currently too many flaws with the frameworks, which is made for the technology. We would have to work around these flaws to get the technology to work. This means using IoT would be easier with the tools out on the internet. As seen in this thesis, WoT tries to prevent re-usage of code which comes along while creating an application with IoT. But there exists solutions to prevent this problem for the IoT technology. The WoT technology tries to create edge gateways to simplify the process of getting sensors connected to the World Wide Web, which IoT has a hard time doing.

But if we take a look at for instance Amazon Web Services solution to the IoT technology, it already has a fix to several of the problems identified in this thesis for the WoT frameworks. Their solution is called AWS Greengrass [4], and gives the user the possibility to create edge gateways connected to the internet where sensors can connect to and provide its data. This sounds a lot similar to what WoT tries to achieve. The difference is that AWS have spent many years developing and perfecting this technology, providing the user with a framework with few flaws. AWS Greengrass also makes all communication encrypted with advanced algorithms, which the WoT framework tries to do with their security building block. Because of this, one would be better of using the IoT technology, and if encountering the problems which the WoT technology tries to prevent, then look towards the usage of solutions like AWS Greengrass instead. For a project like Smart Ocean, which probably will be dealing with confidential data, they should be using more secure and well-built frameworks, instead of trying out new technologies. The current state of the technology means a developer have to spend a lot of time fixing the bugs and vulnerabilities which the current WoT frameworks provides. Here one could use less time by considering solutions like AWS Greengrass. And it is not only AWS which has tried to prevent these common problems with the IoT technology, as both Google Cloud and Microsoft Azure have their own ways to fix them.

The fact that WoT has been in development since 2009, and has only reached this stage with its development is not a good sign. In a press release in 2020 [33], W3C mentions a few frameworks which was meant to be the leading frameworks for this technology. The fact that we have evaluated three out of the six they provide and found as many flaws as we have, either means W3C does not do continuously checkups on their W3C members, or that this is where WoT currently stands with its development.

8.4 Related Works

The technology has not received a lot of attention from the web community. This means few people has evaluated the quality of the technology beforehand. Daves [29] investigated the current state of the technology in 2015. Davis explains what the priorities of W3Cs development were at the beginning of the technology, and states that W3C tries to expand the usage of more protocols and increase the security of the technology. We can see today that they have reached a long way with this. Daves also recommends using Node-RED as a way to get the WoT application to the cloud, which may be a better way of using the framework than running it locally as we did. Mathews book [20], released in 2013, discuss how the development of the technology will be. The main claim here is that the the technology has been in a constant dynamic development and the technology lacked matureness since day one. Mathew also states the importance of the security aspect of the technology, since once a thing has reached the web, it needs to be trusted to keep its privacy. As we see security is an important topic when it comes to the technology, we can also look into Sardars [31] article. Sardar concludes in 2021 that the technologies security measurements are to immature to tackle the security challenges that exists today, and is prone to multiple attacks. Something we have not discussed in this thesis is the latency of a

WoT solution. The article from Naik [23] discusses the challenges of using the HTTP/1 for a WoT application. The HTTP/1 is the standard protocol most web solutions use today, which we also use in this thesis. The article concludes with the protocol being too slow for a wide scale WoT application, and proposes to use low latency protocols like CoAP instead. Naik also states that HTTP/2 could significantly increase the latency for a WoT application, but then W3C needs to do some work to get the protocol implemented. But there is a problem with this, which Mashal [19] expresses in his article. Mashal explains that the protocol is not suitable for WoT, as it currently has to many security flaws to be used, which would be critical for a WoT application. Garcías [9] article explains how to implement the WoT technology. Garcías clearly states that the technology will decrease the development cost as the the amount of code needed to get the applications to work will decrease.

8.5 Future Work

If we decided to use the WoT technology, even with all the flaws found in this thesis, then there is some steps we can do to get the technology to work. First step would be to investigate the other frameworks which W3C recommends and see if these frameworks has reached a further stage than the frameworks evaluated in this thesis. There is a possibility that the choice of frameworks used in this thesis did not showcase the full potential of the technology. But the fact that we picked the frameworks from the biggest developers make this highly unlikely. Second thing we could do to get the technology to work, is try to fix the flaws by the frameworks before using them in an application. This step would demand some work, alongside cooperation with the developers who created the frameworks, as the documentation is not provided for this. Third thing one could do is wait and see if the technology gets more development over time before being used. As the frameworks explored in this thesis has continuously been updated during the making of this thesis means there is still hope that one day the frameworks could be more useful than what we have experienced today.

Based on the information collected, considering solutions created from the IoT technology would be more time efficient. This would create more stable solutions for the Smart Ocean project. The reason for this is that this technology has reached a better stage of matureness than what the WoT technology has. It provides better support and frameworks, and is highly sought after by big companies. There is also more developers which has experience with development with the IoT technology, making it easier to find consultants to help the development process.

List of Figures

List of Tables

Appendix A

Thing Descriptors

In the following chapter will we list some of the important thing descriptors from the different frameworks.

The next following thing descriptors are hand-made to work along the different frameworks:

```
{
   "@context":[
      "https://www.w3.org/2019/wot/td/v1",
      {
         "@language":"en"
      }
   ],
   "id":"PD1:thing",
   "title":"PD1",
   "description":"Pilot Demonstrator 1",
   "properties":{
      "CO2":{
         "type":"integer",
         "description":"current CO2 value",
         "observable":false,
         "readOnly":true,
         "writeOnly":false
      },
      "Temperature":{
         "type":"integer",
         "description":"current Temperature value in Celcius",
         "observable":false,
         "readOnly":true,
         "writeOnly":false
      },
      "pH":{
         "type":"integer",
         "description":"current pH value",
         "observable":false,
         "readOnly":true,
         "writeOnly":false
      },
      "Gravity":{
         "type":"integer",
         "description":"current G force value in m/s^2",
         "observable":false,
            "readOnly":true,
            "writeOnly":false
        },
        "Salinity":{
            "type":"integer",
            "description":"current Salinity value in parts per
                thousand",
            "observable":false,
            "readOnly":true,
            "writeOnly":false
        }
    },
    "events":{
        "GasLeakage":{
            "type":"boolean",
            "description":"Warning from gas leakage"
        }
    },
    "@type":"Thing",
    "security":[
        "nosec_sc"
    ],
    "securityDefinitions":{
        "nosec_sc":{
            "scheme":"nosec"
        }
    }
}
```

Listing A.1: Thing Descriptor for Pilot Demonstrator 1

```
{
    "@context":[
        "https://www.w3.org/2019/wot/td/v1",
        {
            "@language":"en"
        }
    ],
    "id":"PD2:thing",
    "title":"PD2",
    "description":"Pilot Demonstrator 2",
    "properties":{
        "AcousticTomopraphy":{
            "type":"integer",
            "unit":"hertz",
            "description":"current Acoustic Tomography in Hertz",
            "observable":false,
            "readOnly":true,
            "writeOnly":false
        },
        "AcousticEnvironment":{
            "type":"integer",
            "unit":"hertz",
            "description":"current Acoustic Environment in Hertz",
            "observable":false,
            "readOnly":true,
            "writeOnly":false
        },
        "GeoPositioning":{
            "type":"integer",
            "unit":"coordinates",
            "description":"current Geo-Positioning in coordinates",
            "observable":false,
            "readOnly":true,
            "writeOnly":false
        },
        "OceanGraphicPoint":{
            "type":"integer",
            "unit":"coordinates",
            "description":"current Acoustic Environment in Hertz",
            "observable":false,
            "readOnly":true,
            "writeOnly":false
        }
    },
    "@type":"Thing",
    "security":[
        "nosec_sc"
    ],
    "securityDefinitions":{
        "nosec_sc":{
            "scheme":"nosec"
        }
    }
}
```

Listing A.2: Thing Descriptor for Pilot Demonstrator 2

```
{
    "@context":[
        "https://www.w3.org/2019/wot/td/v1",
        {
            "@language":"en"
        }
    ],
    "id":"PD3:thing",
    "title":"PD3",
    "description":"Pilot Demonstrator 3",
    "properties":{
        "AcousticResonance":{
            "type":"integer",
            "unit":"hertz",
            "description":"current Acoustic Resonance in Hertz",
            "observable":false,
            "readOnly":true,
            "writeOnly":false
        },
        "GUW":{
            "type":"integer",
            "description":"current GUW measurements",
            "observable":false,
            "readOnly":true,
            "writeOnly":false
        }
    },
    "@type":"Thing",
    "security":[
        "nosec_sc"
    ],
    "securityDefinitions":{
        "nosec_sc":{
            "scheme":"nosec"
        }
    }
}
```

Listing A.3: Thing Descriptor for Pilot Demonstrator 3

```
{
    "@context":[
        "https://www.w3.org/2019/wot/td/v1",
        {
            "@language":"en"
        }
    ],
    "id":"PD4:thing",
    "title":"PD4",
    "description":"Pilot Demonstrator 4",
    "properties":{
        "PipelineVibration":{
            "type":"integer",
            "unit":"hertz",
            "description":"current Pipeline Vibrations in Hertz",
            "observable":false,
            "readOnly":true,
            "writeOnly":false
        },
        "GUW":{
            "type":"integer",
            "description":"current GUW measurements",
            "observable":false,
            "readOnly":true,
            "writeOnly":false
        }
    },
    "events":{
        "GasLeakage":{
            "type":"boolean",
            "description":"Warning from gas leakage"
        }
    },
    "@type":"Thing",
    "security":[
        "nosec_sc"
    ],
    "securityDefinitions":{
        "nosec_sc":{
            "scheme":"nosec"
        }
    }
}
```

Listing A.4: Thing Descriptor for Pilot Demonstrator 4

The next following thing descriptors are made from the ThingWeb framework:

```
{
    "@context":[
        "https://www.w3.org/2019/wot/td/v1",
        {
            "@language":"en"
        }
    ],
    "id":"PD1:thing",
    "title":"PD1",
    "description":"Pilot Demonstrator 1",
    "properties":{
      "CO2":{
        "type":"integer",
        "description":"current CO2 value",
        "observable":false,
        "readOnly":true,
        "writeOnly":false,
        "forms":[
          {
            "href":"http://192.168.1.36:8080/PD1/properties/CO2",
            "contentType":"application/json",
            "op":[
              "readproperty"
            ],
            "htv:methodName":"GET"
          }
        ]
      },
      "Temperature":{
        "type":"integer",
        "description":"current Temperature value in Celcius",
        "observable":false,
        "readOnly":true,
        "writeOnly":false,
        "forms":[
          {
            "href":"http://192.168.1.36:8080/PD1/properties/Temperature",
            "contentType":"application/json",
            "op":[
              "readproperty"
            ],
            "htv:methodName":"GET"
          }
        ]
      },
      "pH":{
        "type":"integer",
        "description":"current pH value",
        "observable":false,
        "readOnly":true,
        "writeOnly":false,
        "forms":[
          {
            "href":"http://192.168.1.36:8080/PD1/properties/pH",
            "contentType":"application/json",
            "op":[
              "readproperty"
            ],
            "htv:methodName":"GET"
          }
        ]
      },
      "Gravity":{
        "type":"integer",
        "description":"current G force value in m/s^2",
        "observable":false,
        "readOnly":true,
        "writeOnly":false,
        "forms":[
          {
               "href":"http://192.168.1.36:8080/PD1/properties/Gravity",
               "contentType":"application/json",
               "op":[
                  "readproperty"
               ],
               "htv:methodName":"GET"
            }
         ]
      },
      "Salinity":{
         "type":"integer",
         "description":"current Salinity value in parts per thousand",
         "observable":false,
         "readOnly":true,
         "writeOnly":false,
         "forms":[
            {
               "href":"http://192.168.1.36:8080/PD1/properties/Salinity",
               "contentType":"application/json",
               "op":[
                  "readproperty"
               ],
               "htv:methodName":"GET"
            }
         ]
      }
   },
   "events":{
      "GasLeakage":{
         "type":"boolean",
         "description":"Warning from gas leakage",
         "forms":[
            {
               "href":"http://192.168.1.36:8080/PD1/events/GasLeakage",
               "contentType":"application/json",
               "subprotocol":"longpoll",
               "op":[
                  "subscribeevent"
               ]
            },
            {
               "href":"ws://192.168.1.36:8081/PD1/events/GasLeakage",
               "contentType":"application/json",
               "op":"subscribeevent"
            }
         ]
      }
   },
   "@type":"Thing",
   "security":[
      "nosec_sc"
   ],
   "forms":[
      {
         "href":"http://192.168.1.36:8080/PD1/all/properties",
         "contentType":"application/json",
         "op":[
            "readallproperties",
            "readmultipleproperties",
            "writeallproperties",
            "writemultipleproperties"
         ]
      }
   ],
   "securityDefinitions":{
      "nosec_sc":{
         "scheme":"nosec"
      }
   }
}
```

Listing A.5: ThingWeb Thing Descriptor for Pilot Demonstrator 1

```
{
   "@context":[
      "https://www.w3.org/2019/wot/td/v1",
      {
         "@language":"en"
      }
   ],
   "id":"PD2:thing",
   "title":"PD2",
   "description":"Pilot Demonstrator 2",
   "properties":{
      "AcousticTomopraphy":{
         "type":"integer",
         "unit":"hertz",
         "description":"current Acoustic Tomography in Hertz",
         "observable":false,
         "readOnly":true,
         "writeOnly":false,
         "forms":[
            {
               "href":"http://192.168.1.36:8080/PD2/properties/
                   AcousticTomopraphy",
               "contentType":"application/json",
               "op":[
                  "readproperty"
               ],
               "htv:methodName":"GET"
            }
         ]
      },
      "AcousticEnvironment":{
         "type":"integer",
         "unit":"hertz",
         "description":"current Acoustic Environment in Hertz",
         "observable":false,
         "readOnly":true,
         "writeOnly":false,
         "forms":[
            {
               "href":"http://192.168.1.36:8080/PD2/properties/
                   AcousticEnvironment",
               "contentType":"application/json",
               "op":[
                  "readproperty"
               ],
               "htv:methodName":"GET"
            }
          ]
      },
      "GeoPositioning":{
          "type":"integer",
          "unit":"coordinates",
          "description":"current Geo−Positioning in coordinates",
          "observable":false,
          "readOnly":true,
          "writeOnly":false,
          "forms":[
            {
              "href":"http://192.168.1.36:8080/PD2/properties/
                  GeoPositioning",
              "contentType":"application/json",
              "op":[
                "readproperty"
              ],
              "htv:methodName":"GET"
            }
          ]
      },
      "OceanGraphicPoint":{
          "type":"integer",
          "unit":"coordinates",
          "description":"current Acoustic Environment in Hertz",
          "observable":false,
          "readOnly":true,
          "writeOnly":false,
          "forms":[
            {
              "href":"http://192.168.1.36:8080/PD2/properties/
                  OceanGraphicPoint",
              "contentType":"application/json",
              "op":[
                "readproperty"
              ],
              "htv:methodName":"GET"
            }
          ]
      }
  },
  "@type":"Thing",
  "security":[
      "nosec_sc"
  ],
  "forms":[
      {
          "href":"http://192.168.1.36:8080/PD2/all/properties",
          "contentType":"application/json",
          "op":[
            "readallproperties",
            "readmultipleproperties",
            "writeallproperties",
            "writemultipleproperties"
          ]
      }
  ],
  "securityDefinitions":{
      "nosec_sc":{
          "scheme":"nosec"
      }
```

```
    }
}
```

Listing A.6: ThingWeb Thing Descriptor for Pilot Demonstrator 2

```
{
    "@context":[
        "https://www.w3.org/2019/wot/td/v1",
        {
            "@language":"en"
        }
    ],
    "id":"PD3:thing",
    "title":"PD3",
    "description":"Pilot Demonstrator 3",
    "properties":{
        "AcousticResonance":{
            "type":"integer",
            "unit":"hertz",
            "description":"current Acoustic Resonance in Hertz",
            "observable":false,
            "readOnly":true,
            "writeOnly":false,
            "forms":[
                {
                    "href":"http://192.168.1.36:8080/PD3/properties/
                        AcousticResonance",
                    "contentType":"application/json",
                    "op":[
                        "readproperty"
                    ],
                    "htv:methodName":"GET"
                }
            ]
        },
        "GUW":{
            "type":"integer",
            "description":"current GUW measurements",
            "observable":false,
            "readOnly":true,
            "writeOnly":false,
            "forms":[
                {
                    "href":"http://192.168.1.36:8080/PD3/properties/GUW"
                        ,
                    "contentType":"application/json",
                    "op":[
                        "readproperty"
                    ],
                    "htv:methodName":"GET"
                }
            ]
        }
    },
    "@type":"Thing",
    "security":[
        "nosec_sc"
    ],
    "forms":[
        {
            "href":"http://192.168.1.36:8080/PD3/all/properties",
            "contentType":"application/json",
            "op":[
                "readallproperties",
                "readmultipleproperties",
                "writeallproperties",
                "writemultipleproperties"
            ]
        }
    ],
    "securityDefinitions":{
        "nosec_sc":{
            "scheme":"nosec"
        }
    }
}
```

Listing A.7: ThingWeb Thing Descriptor for Pilot Demonstrator 3

```
{
    "@context":[
        "https://www.w3.org/2019/wot/td/v1",
        {
            "@language":"en"
        }
    ],
    "id":"PD4:thing",
    "title":"PD4",
    "description":"Pilot Demonstrator 4",
    "properties":{
        "PipelineVibration":{
            "type":"integer",
            "unit":"hertz",
            "description":"current Pipeline Vibrations in Hertz",
            "observable":false,
            "readOnly":true,
            "writeOnly":false,
            "forms":[
                {
                    "href":"http://192.168.1.36:8080/PD4/properties/
                        PipelineVibration",
                    "contentType":"application/json",
                    "op":[
                        "readproperty"
                    ],
                    "htv:methodName":"GET"
                }
            ]
        },
        "GUW":{
            "type":"integer",
            "description":"current GUW measurements",
            "observable":false,
            "readOnly":true,
            "writeOnly":false,
            "forms":[
                {
                    "href":"http://192.168.1.36:8080/PD4/properties/GUW"
                        ,
                    "contentType":"application/json",
                    "op":[
                        "readproperty"
                    ],
                    "htv:methodName":"GET"
                }
            ]
        }
    },
    "events":{
        "GasLeakage":{
            "type":"boolean",
            "description":"Warning from gas leakage",
            "forms":[
                {
                    "href":"http://192.168.1.36:8080/PD4/events/GasLeakage",
                    "contentType":"application/json",
                    "subprotocol":"longpoll",
                    "op":[
                        "subscribeevent"
                    ]
                },
                {
                    "href":"ws://192.168.1.36:8081/PD4/events/GasLeakage",
                    "contentType":"application/json",
                    "op":"subscribeevent"
                }
            ]
        }
    },
    "@type":"Thing",
    "security":[
        "nosec_sc"
    ],
    "forms":[
        {
            "href":"http://192.168.1.36:8080/PD4/all/properties",
            "contentType":"application/json",
            "op":[
                "readallproperties",
                "readmultipleproperties",
                "writeallproperties",
                "writemultipleproperties"
            ]
        }
    ],
    "securityDefinitions":{
        "nosec_sc":{
            "scheme":"nosec"
        }
    }
}
```

Listing A.8: ThingWeb Thing Descriptor for Pilot Demonstrator 4

The next following thing descriptors are made from the WebThings framework:

```
{
    "@type":[
        "MultiLevelSensor"
    ],
    "description":"Sensor for underwater technology",
    "links":[
        {
            "rel":"properties",
            "href":"/properties"
        },
        {
            "rel":"actions",
            "href":"/actions"
        },
        {
            "rel":"events",
            "href":"/events"
        }
    ],
    "id":"PD1",
    "title":"PD1",
    "@context":"https://iot.mozilla.org/schemas",
    "actions":{

    },
    "properties":{
        "Temperature":{
            "unit":"degree celsius",
            "@type":"TemperatureProperty",
            "description":"The current temperature in celsius",
            "readOnly":true,
            "links":[
                {
                    "rel":"property",
                    "href":"/properties/Temperature"
                }
            ],
            "title":"Temperature",
            "type":"number"
        },
        "CO2":{
            "unit":"percent",
            "@type":"LevelProperty",
            "description":"The current CO2 in %",
            "maximum":100,
            "readOnly":true,
            "links":[
                {
                    "rel":"property",
                    "href":"/properties/CO2"
                }
            ],
            "title":"CO2",
            "type":"number",
            "minimum":0
        },
        "pH":{
            "@type":"LevelProperty",
            "description":"The current pH level",
            "readOnly":true,
            "links":[
                {
                    "rel":"property",
                    "href":"/properties/pH"
                }
            ],
            "title":"pH",
            "type":"number"
        },
        "Gravity":{
            "@type":"LevelProperty",
            "description":"The current Gravity in m/s^2",
            "readOnly":true,
            "links":[
                {
                    "rel":"property",
                    "href":"/properties/Gravity"
                }
            ],
            "title":"Gravity",
            "type":"number"
        },
        "Salinity":{
            "@type":"LevelProperty",
            "description":"The current Salinity in parts per thousand"
                ,
            "readOnly":true,
            "links":[
                {
                    "rel":"property",
                    "href":"/properties/Salinity"
                }
            ],
            "title":"Salinity",
            "type":"number"
        }
    },
    "events":{
        "GasLeakage":{
            "description":"Event for GasLeakage",
            "links":[
                {
                    "rel":"event",
                    "href":"/events/GasLeakage"
                }
            ],
            "type":"boolean"
        }
    }
}
```

Listing A.9: WebThing Thing Descriptor for Pilot Demonstrator 1

```
{
    "@type":[
        "MultiLevelSensor"
    ],
    "description":"Sensor for underwater technology",
    "links":[
        {
            "rel":"properties",
            "href":"/properties"
        },
        {
            "rel":"actions",
            "href":"/actions"
        },
        {
            "rel":"events",
            "href":"/events"
        }
    ],
"id":"PD2",
"title":"PD2",
"@context":"https://iot.mozilla.org/schemas",
"actions":{

},
"properties":{
   "Geo-Positioning":{
      "@type":"LevelProperty",
      "description":"The Geo-Positioning in coordinates",
      "readOnly":true,
      "links":[
         {
            "rel":"property",
            "href":"/properties/Geo-Positioning"
         }
      ],
      "title":"Geo-Positioning",
      "type":"number"
   },
   "Acoustic Environment":{
      "unit":"hertz",
      "@type":"FrequencyProperty",
      "description":"The Acoustic Environment in Hertz",
      "readOnly":true,
      "links":[
         {
            "rel":"property",
            "href":"/properties/Acoustic Environment"
         }
      ],
      "title":"Acoustic Environment",
      "type":"number"
   },
   "Acoustic Tomography":{
      "unit":"hertz",
      "@type":"FrequencyProperty",
      "description":"The current Acoustic Tomography in Hertz",
      "readOnly":true,
      "links":[
         {
            "rel":"property",
            "href":"/properties/Acoustic Tomography"
         }
      ],
      "title":"Acoustic Tomography",
      "type":"number"
   },
   "Oceanographic Point":{
      "@type":"LevelProperty",
      "description":"The Oceanographic Point in coordinates",
      "readOnly":true,
      "links":[
         {
            "rel":"property",
            "href":"/properties/Oceanographic Point"
         }
      ],
      "title":"Oceanographic Point",
      "type":"number"
   }
},
    "events":{

    }
}
```

Listing A.10: WebThing Thing Descriptor for Pilot Demonstrator 2

```
{
    "@type":[
        "MultiLevelSensor"
    ],
    "description":"Sensor for underwater technology",
    "links":[
        {
            "rel":"properties",
            "href":"/properties"
        },
        {
            "rel":"actions",
            "href":"/actions"
        },
        {
            "rel":"events",
            "href":"/events"
        }
    ],
    "id":"PD3",
    "title":"PD3",
    "@context":"https://iot.mozilla.org/schemas",
    "actions":{

    },
    "properties":{
        "Acoustic Resonance":{
            "unit":"hertz",
            "@type":"FrequencyProperty",
            "description":"The current Acoustic Resonance in Hertz",
            "readOnly":true,
            "links":[
                {
                    "rel":"property",
                    "href":"/properties/Acoustic Resonance"
                }
            ],
            "title":"Acoustic Resonance",
            "type":"number"
        },
        "GUW Measurements":{
            "@type":"LevelProperty",
            "description":"The current GUW Measurements",
            "readOnly":true,
            "links":[
                {
                    "rel":"property",
                    "href":"/properties/GUW Measurements"
                }
            ],
            "title":"GUW Measurements",
            "type":"number"
        }
    },
    "events":{
    }
}
```

Listing A.11: WebThing Thing Descriptor for Pilot Demonstrator 3

```
{
    "@type":[
        "MultiLevelSensor"
    ],
    "description":"PD4 Sensor for underwater technology",
    "links":[
        {
            "rel":"properties",
            "href":"/properties"
        },
        {
            "rel":"actions",
            "href":"/actions"
        },
        {
            "rel":"events",
            "href":"/events"
        }
    ],
    "id":"PD4",
    "title":"PD4",
    "@context":"https://iot.mozilla.org/schemas",
    "actions":{

    },
    "properties":{
        "Pipeline Vibrations (DAS)":{
            "unit":"hertz",
            "@type":"FrequencyProperty",
            "description":"The current Pipeline Vibrations in Hertz",
            "readOnly":true,
            "links":[
                {
                    "rel":"property",
                    "href":"/properties/Pipeline Vibrations (DAS)"
                }
            ],
            "title":"Pipeline Vibrations (DAS)",
            "type":"number"
        },
        "GUW Measurements":{
            "@type":"LevelProperty",
            "description":"The current GUW Measurements",
            "readOnly":true,
            "links":[
                {
                    "rel":"property",
                    "href":"/properties/GUW Measurements"
                }
            ],
            "title":"GUW Measurements",
            "type":"number"
        }
    },
    "events":{
        "GasLeakage":{
            "description":"Event for GasLeakage",
            "links":[
                {
                    "rel":"event",
                    "href":"/events/GasLeakage"
                }
            ],
            "type":"boolean"
        }
    }
}
```

Listing A.12: WebThing Thing Descriptor for Pilot Demonstrator 4

Appendix B

Source Code

The source code for the project is located in two different GitHub repositories.

The source code for the ThingWeb prototype is located on `https://github.com/181182/Master_ThingWeb`.

The source code for WebThing prototype is located on `https://github.com/181182/Master_WebThing`.

The json export for Node-RED prototype is located on `https://github.com/181182/Master_Node-RED`

Bibliography

[1] *About : Node-RED*. [Online; accessed 26. Nov. 2020]. Nov. 2020. URL: https://nodered.org/about.

[2] *About W3C*. [Online; accessed 9. Jun. 2021]. Jan. 2021. URL: https://www.w3.org/Consortium.

[3] aviviano. *Windows Network Architecture and the OSI Model - Windows drivers*. [Online; accessed 26. Nov. 2020]. Nov. 2020. URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/network/windows-network-architecture-and-the-osi-model.

[4] *AWS IoT Greengrass - Amazon Web Services*. [Online; accessed 31. May 2021]. May 2021. URL: https://aws.amazon.com/greengrass.

[5] Arshdeep Bahga and Vijay Madisetti. *Internet of Things (A Hands-on-Approach) 1st Edition*. VPT; 1st edition (August 9, 2014), 2014.

[6] A.W. Brown and K.C. Wallnau. "A framework for evaluating software technology." In: *IEEE Software* 13.5 (1996), pp. 39–49. DOI: 10.1109/52.536457.

[7] *Desigo CC — Building management systems — Siemens Global*. https://new.siemens.com/global/en/products/buildings/automation/desigo/building-management/desigo-cc.html. (Accessed on 01/23/2021).

[8] *Eclipse Thingweb*. [Online; accessed 26. Nov. 2020]. Mar. 2018. URL: https://projects.eclipse.org/proposals/eclipse-thingweb.

[9] Andrés García Mangas and Francisco José Suárez Alonso. "WOTPY: A framework for web of things applications." In: *Computer Communications* 147 (2019), pp. 235–251. ISSN: 0140-3664. DOI: https://doi.org/10.1016/j.comcom.2019.09.004. URL: https://www.sciencedirect.com/science/article/pii/S0140366419304633.

[10] *GitHub - thingweb/node-red-contrib-web-of-things: A node-red node for the Web of Things*. https://github.com/thingweb/node-red-contrib-web-of-things. (Accessed on 01/22/2021).

[11] Dominique Guinard and Vlad Trifa. *Building the Web of Things: With examples in Node.js and Raspberry Pi*. Manning Publications; 1st edition (June 18, 2016), 2016.

[12] *How much of the ocean have we explored?* https://oceanservice.noaa.gov/facts/exploration.html. (Accessed on 01/23/2021).

[13] *How to Install JDK 8 (on Windows, Mac OS, Ubuntu) and Get Started with Java Programming*. [Online; accessed 3. May 2021]. Feb. 2021. URL: https://www3.ntu.edu.sg/home/ehchua/programming/howto/JDK_Howto.html.

[14] *HTML: HyperText Markup Language.* [Online; accessed 26. Nov. 2020]. Nov. 2020. URL: https://developer.mozilla.org/en-US/docs/Web/HTML.

[15] *Introducing Mozilla WebThings – Mozilla Hacks - the Web developer blog.* [Online; accessed 27. Apr. 2021]. Apr. 2021. URL: https://hacks.mozilla.org/2019/04/introducing-mozilla-webthings.

[16] *ISO/IEC 25010:2011.* [Online; accessed 27. May 2021]. May 2021. URL: https://www.iso.org/standard/35733.html.

[17] M. Krochmal and S. Cheshire. *Multicast DNS.* [Online; accessed 15. Mar. 2021]. Feb. 2013. URL: https://tools.ietf.org/html/rfc6762.

[18] *Library - Node-RED.* [Online; accessed 10. Jun. 2021]. June 2021. URL: https://flows.nodered.org.

[19] Ibrahim Mashal et al. "Choices for interaction with things on Internet and underlying issues." In: *Ad Hoc Networks* 28 (2015), pp. 68–90. ISSN: 1570-8705. DOI: https://doi.org/10.1016/j.adhoc.2014.12.006. URL: https://www.sciencedirect.com/science/article/pii/S1570870514003138.

[20] Sujith Samuel Mathew et al. "The Web of Things - Challenges and Enabling Technologies." In: *Internet of Things and Inter-cooperative Computational Technologies for Collective Intelligence.* Ed. by Nik Bessis et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 1–23. ISBN: 978-3-642-34952-2. DOI: 10.1007/978-3-642-34952-2_1. URL: https://doi.org/10.1007/978-3-642-34952-2_1.

[21] *Mozilla IoT.* https://iot.mozilla.org/. (Accessed on 01/23/2021).

[22] *Mozilla IoT about.* [Online; accessed 26. Nov. 2020]. Nov. 2020. URL: https://iot.mozilla.org/about.

[23] Nitin Naik and Paul Jenkins. "Web protocols and challenges of Web latency in the Web of Things." In: *2016 Eighth International Conference on Ubiquitous and Future Networks (ICUFN).* 2016, pp. 845–850. DOI: 10.1109/ICUFN.2016.7537156.

[24] *Node-RED.* https://nodered.org/. (Accessed on 01/23/2021).

[25] node-red. *node-red-nodegen.* [Online; accessed 26. Nov. 2020]. Nov. 2020. URL: https://github.com/node-red/node-red-nodegen.

[26] SFI Smart Ocean. *SFI Smart Ocean - Flexible and cost-effective monitoring for management of productive and healthy ocean.* Tech. rep. SFI Smart Ocean, 2021.

[27] *OWASP Top Ten Web Application Security Risks | OWASP.* [Online; accessed 10. May 2021]. Apr. 2021. URL: https://owasp.org/www-project-top-ten.

[28] Mitch Pronschinske. "A General Software Maturity Model." In: *Dzone* (Jan. 2016). URL: https://dzone.com/articles/a-general-software-maturity-model.

[29] Dave Raggett. "The Web of Things: Challenges and Opportunities." In: *Computer* 48.5 (2015), pp. 26–32. DOI: 10.1109/MC.2015.149.

[30] *rest-discuss : Message: Re: [rest-discuss] RFC for REST?* [Online; accessed 26. Nov. 2020]. Nov. 2020. URL: https://web.archive.org/web/20091111012314/http://tech.groups.yahoo.com/group/rest-discuss/message/6757.

[31] Ruhma Sardar and Tayyaba Anees. “Web of Things: Security Challenges and Mechanisms.” In: *IEEE Access* 9 (2021), pp. 31695–31711. DOI: 10.1109/ACCESS.2021.3057655.

[32] *SFI Smart Ocean.* [Online; accessed 9. Jun. 2021]. June 2021. URL: https://sfismartocean.no.

[33] *Solution for IoT Interoperability - W3C Web of Things (WoT).* [Online; accessed 26. Apr. 2021]. Aug. 2020. URL: https://www.w3.org/2020/04/pressrelease-wot-rec.html.en.

[34] *Stack Overflow - Where Developers Learn, Share, & Build Careers.* [Online; accessed 7. Jun. 2021]. June 2021. URL: https://stackoverflow.com.

[35] *The WebSocket API (WebSockets) - Web APIs — MDN.* https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API. (Accessed on 01/22/2021).

[36] *thingweb - Exposed Thing with node-wot as Dependency.* [Online; accessed 28. Apr. 2021]. Mar. 2021. URL: https://www.thingweb.io/hands-on-exposed-thing-guide.html.

[37] *URI differences: URN, URL... – Dimitri's Wanderings.* [Online; accessed 4. May 2021]. Jan. 2016. URL: https://dimitri.janczak.net/2015/08/04/uri-differences-urn-url.

[38] w3c. *web-of-things-framework.* [Online; accessed 26. Nov. 2020]. Nov. 2020. URL: https://github.com/w3c/web-of-things-framework.

[39] *Web of Things (WoT) Architecture.* [Online; accessed 26. Nov. 2020]. Apr. 2020. URL: https://www.w3.org/TR/wot-architecture.

[40] *Web of Things (WoT) Binding Templates.* [Online; accessed 26. Nov. 2020]. Jan. 2020. URL: https://www.w3.org/TR/wot-binding-templates.

[41] *Web of Things (WoT) Scripting API.* [Online; accessed 26. Nov. 2020]. Nov. 2020. URL: https://www.w3.org/TR/wot-scripting-api.

[42] *Web of Things (WoT) Security and Privacy Guidelines.* [Online; accessed 26. Nov. 2020]. Nov. 2019. URL: https://www.w3.org/TR/wot-security.

[43] *Web of Things (WoT) Thing Description.* [Online; accessed 26. Nov. 2020]. July 2020. URL: https://www.w3.org/TR/wot-thing-description/#introduction.

[44] *WebThings Gateway by Mozilla.* [Online; accessed 27. Apr. 2021]. Apr. 2021. URL: https://iot.mozilla.org/gateway.

[45] WebThingsIO. *gateway.* [Online; accessed 3. May 2021]. May 2021. URL: https://github.com/WebThingsIO/gateway/releases/tag/1.0.0.

[46] *What is the Web of Things? – Web of Things.* https://webofthings.org/2017/04/08/what-is-the-web-of-things/. (Accessed on 01/23/2021).

[47] *Working with JSON.* [Online; accessed 26. Nov. 2020]. Nov. 2020. URL: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON.