Technical University of Crete

School of Electrical and Computer Engineering

Diploma Thesis

Boliotis Manousos

Asynchronous Things and Services Descriptions in the Web of Things

Supervisor: Euripides G.M. Petrakis, Professor, TUC

Examination Committee

Euripides Petrakis	Chrysi Tsinaraki	Georgios Chalkiadakis
Professor	Lab Instructor	Professor

Chania, Greece, 2025

Abstract

In IoT, people and devices can interact in a shared network which promotes data sharing and services among them. Yet, as the number of connected devices continues to increase, the coexistence of multiple protocols and standards creates considerable complexities. The Web of Things (WoT) is a layer that can be built on-top of the IoT to facilitate communication and management of heterogeneous objects in a unified way. The Web of Things (WoT) relies on standardized descriptions to enable interoperability across heterogeneous devices and services. Although the W3C Thing Description (TD) specification provides a solid framework for describing IoT resources, it offers hardly any direct support for the IoT's most common interaction pattern—one where asynchronous communications among distributed, event-driven components is the norm. In addition, if the TD is to be useful to the IoT community, it must serve not only synchronous components but also those that depend on asynchronous communications. This dissertation describes the creation of tools that connect the AsyncAPI and WoT ecosystems. The AsyncAPI tools produce WoT Thing Descriptions that work with asynchronous operations. The choice of using AsyncAPI over modifying existing W3C specifications was made for two reasons: 1. AsyncAPI is a mature specification that industry uses to describe asynchronous services. 2. Using AsyncAPI offers a path to add support for asynchronous operations in WoT without modifying its existing specifications meaning that instead of changing the core WoT specifications to accommodate asynchronous operations, you can use AsyncAPI as an extension or complement . We validated the tools through practical case studies involving IoT systems with asynchronous communication needs. This work contributes to the WoT community because it provides a practical solution for integrating asynchronous WoT devices into standardized WoT ecosystems.

Keywords: WoT, IoT, AsyncAPI, Thing Description, W3C, Asynchronous Communication

Acknowledgements

My first and foremost gratitude goes to my advisor, Professor Euripidis Petrakis, to whom I am greatly indebted for giving me guidance, support, and encouragement throughout my research. I also thank all members of our laboratory for their collaborations, discussions, and support. A huge thanks to my family and friends, whose support, patience, and encouragement have been my strength throughout this journey. Their belief in me has been a constant source of motivation, and I am truly grateful for their presence in my life.

Contents

1. Introduction

1.1 Background

1.1.1 Introduce the Web of Things (WoT)

The Web of Things (WoT)[1] is a set of standards aimed at creating a unified framework so that Internet of Things (IoT) devices can more easily and quickly communicate with users and services. Things Description is an integral part of an WoT architecture which also comprises Binding Template, Scripting APIs, and Security and Privacy Guidelines. Next, we will analyze the purpose of these building blocks of the W3C initiative.

Thing Description is the most essential tool for creating a common ground among various IoT device manufacturers. It describes the device in a JSON format[18] that is readable by both humans and machines. It includes information about the device, such as titles, versions, and descriptions, as well as details about the servers it connects to and the message schemas it uses. It also defines the security mechanisms that are used by the device.

Binding Templates: IoT devices use a multitude of communication protocols since there isn't a single protocol that fits all uses. Therefore, there is a need for certain guidelines and well defined schemas that are common and can be reused across different Thing Descriptions. Manufacturers can develop device-specific interaction mechanisms based on these prototypes.

The WoT scripting API offers the option to use an API based on the ECMA standard[8], providing a foundation for building applications that communicate with each IoT device. This ensures that applications are portable across devices, making the implementation of device logic easier.

Security and Privacy Guidelines are very important as every aspect of the IoT device data exchange and usage is determined by very specific policies. These policies may concern terms of use, communication, metadata usage, device software applications, and more.

1.1.2 Thing Description

The Thing Description plays an important role in the IoT ecosystem as it provides a strictly defined way for describing the capabilities of (physical or virtual) devices and the dependencies that must be met for the device to function correctly. This ensures that all necessary information about the device is clearly and uniformly presented. Additionally, it provides mechanisms by which a device can become discoverable, thus making it easier to introduce new devices into an already existing network. More specifically, it defines the mechanisms through which we can read the properties, activate actions, and observe the triggering of events. It also specifies the mechanisms used for the secure transmission of information (messages) over the network. Thus, through the Thing Description (TD), consistency is achieved in the way manufacturers describe IoT devices.

1.1.3 OpenAPI Thing Description

In the preceding section, we addressed the description utilizing the W3C standard. At this juncture, it is imperative to reference the study conducted by Aimilios Tzavaras concerning the description of Things with the use of OpenAPI standard[2]. This inclusion is significant, as his approach served as the inspiration for the methodology we intend to employ in describing asynchronous services and Things. With OpenAPI, it is feasible to describe Things that communicate using synchronous protocols but, in the world of IoT, the percentage of devices that communicate synchronously is low.

1.1.4 Asynchronous Services and Devices

The "asynchronous" term refers to operations that function independently, allowing other tasks to continue without waiting for the initial operation to finish. This methodology boosts efficiency by permitting multiple tasks to be performed simultaneously. Asynchronous services in software are designed to perform tasks independently of the main application flow. Unlike synchronous services, which can cause the main process to block until a task is

completed, asynchronous services enable the primary application to continue its operations uninterrupted. This non-blocking behavior facilitates multitasking, as the system does not have to wait for one task to finish before starting another. In the hardware domain, asynchronous devices, such as sensors, actuators or both, operate independently. For example, a temperature sensor might take a reading and then send the result to a central server(broker). The entity that requires the data does not request it directly but instead, it receives a notification when the data is available. This is a publisher-subscriber model. The entity that generates the data (publisher) and the entity that needs the data (subscriber) operate independently(not in a request-response model). By choosing asynchronous methods, systems can achieve greater efficiency and responsiveness, as tasks are decoupled and can proceed without waiting for others to complete.

1.1.5 AsyncAPI

AsyncAPI[14] is an open source specification that provides a framework to define asynchronous APIs (the way OpenAPI[4] defines REST APIs). Its objective is to standardize the way messaging architectures and event-driven systems are documented. This will help developers to create and maintain those systems. This specification is focused on messaging systems protocols like kafka, mqtt, websockets and more. The documentation created using AsynAPI is machine readable so this allows tools to generate code, documentation, and even test cases automatically[14]. Furthermore, there is a growing tooling ecosystem like code generators, validators and more. AsyncAPI full structure and use will be analyzed more in the next chapters.

1.2 Problem Statement

1.2.1 Definition

As things stand, IoT devices are part of our daily lives. Additionally, most manufacturers consider it advantageous to use asynchronous communication methods. At present, despite the W3C's[17] efforts, there is no standard that is both widely used and generally accepted for describing IoT devices that operate using asynchronous communication methods.

Therefore, it is of utmost importance that there be a generally accepted standard that can be applied to describe these devices.

1.2.2 Solution

This specific study set out to address this shortcoming by creating a standardized methodology. The well-known and established AsyncAPI specification[14] was selected to act as the cornerstone for the template tool that was later developed. The main purpose of this tool is to offer an organized and consistent way to describe asynchronously operating IoT devices. AsyncAPI was chosen because of its natural ability to describe devices that operate similarly to asynchronous services. It provides a descriptive framework that is similar to the OpenAPI specification which is well known for describing RESTful APIs. AsyncAPI was the obvious and suitable choice for this project due to its established and demonstrated ability to describe asynchronous services.

1.3 Research Objectives

The primary goal of this thesis is to design and develop tools that will generate Web of Things (WoT) descriptions . This will facilitate the smooth integration and interaction of physical devices and sensors with the web. Once these tools are developed, the next phase is to assess their performance. The objective is to examine whether the mechanisms that are created can generate better results and are more user-friendly than the ones that already exist[19] offering valuable insights for further refinement and enhancement.

1.4 Contributions

To meet the objectives, a set of tools is created. In this chapter a brief outline will be given and there will be a more in depth explanation of them in the later chapters. This is so that the reader understands better the upcoming chapters.

1.4.1 Template

The Template represents all the necessary information and data that must exist so that a Thing Description (TD) can come to life. This is where it defines what parts of the AsyncAPI specification[14] will be used and how. It is actually the pillar of this research.

1.4.2 Generator

The Generator is a tool that is created with the objective of creating Thing Descriptions (TDs) based on the Template. Its significance lies in the fact that given a minimum input with the necessary info about the Thing a valid description will be generated.

The input of the generator is in json format[18], the actual structure of this input will be described in depth in the next chapters. Json is selected because it is the standard way of data transmission on the web. This means that the generator can be used as an endpoint in a RESTful service. It also means that there can be a user interface to make the process of Thing Description creation more user-friendly.

The output of the generator is also in json format[18]. The structure of the output is based on the AsyncAPI specification[14], so the generated Thing Description is a valid AsyncAPI document that can be used with all the available tools that are provided from the AsyncAPI ecosystem .

1.4.3 Thing API Tool

The Thing API Tool is a playground used to demonstrate the capabilities of the AsyncAPI Thing Description (TD). The user can define the set of messages a sensor is capable of sending. The result will be a valid TD and also an interface that simulates the sensor functionality.

1.5 Thesis Structure

Below is presented the way this thesis is structured with a brief explanation on what will be discussed in each chapter.

Chapter two, literature review summarizes the existing research on WoT, Thing Descriptions, and related tools also Identify existing gaps.

Chapter three, methodology of creating the Template and also the tools that accompany it.

Chapter four, Implementation. In depth technical information about the implementation of the tools.

Chapter five, Results and Discussion. Present the results of this thesis and comparison with existing solutions.

Chapter six, conclusions are presented and future work is suggested.

2. Literature Review

2.1 Web of Things

The Web of Things (WoT) initiative aims to unify the world of interconnected devices over the Web. WoT suggests that Things should expose their identity and properties on the Web so that they can be discovered like web applications.

According to this concept, Things join the web and can speak with one another using established asynchronous protocols like COAP MQTT KAFKA and others. Data-interchange formats like JSON, XML and others can be used to describe the capabilities of the Things but each respected asynchronous communication mechanism is used to implement their functionality. The W3C[17] recommends the Web of Things Architecture which offers an abstract framework for the Internet of Things. Based on interoperable modular building blocks the WoT Architecture establishes a model to explain how a consumer interacts with Things. These foundational elements include the Security and Privacy Guidelines, Scripting API, Binding Templates and Thing Description. In order for other Things or clients to interact with a Thing, its metadata must be exposed on the Web using Thing Description which offers a machine-readable (and human-readable) data format for characterizing a Things properties. The goal of Binding templates is to specify network interfaces for Internet of Things protocols and ecosystems. An optional block called a Scripting API makes it possible to implement a Things application logic. Lastly recommendations for the secure setup and deployment of Things are given in the Security and Privacy Guidelines. This architecture recommendation is abstract and does not describe a specific implementation, application or communication protocol.

2.2 W3C Thing Description

The Thing Description is the most important part of the Web of Things initiative. “The WoT Thing Description (TD) can be considered as the entry point of a Thing” (W3C Recommendation, https://www.w3.org/TR/wot-thing-description11/). Therefore, it is crucial to present the structure early in this thesis. The TD[1] consists of five sectors. Those

sectors are Textual Metadata about the Thing, the Interaction Affordances that define how the Thing can be used, Schemas for the message exchange that are machine-readable, Security Definitions that provide metadata for the security mechanisms and Web Links that can be used for external documentation and relations with other Things. An overview can be seen in Fig.1 where a logical grouping of the TD fields is shown.

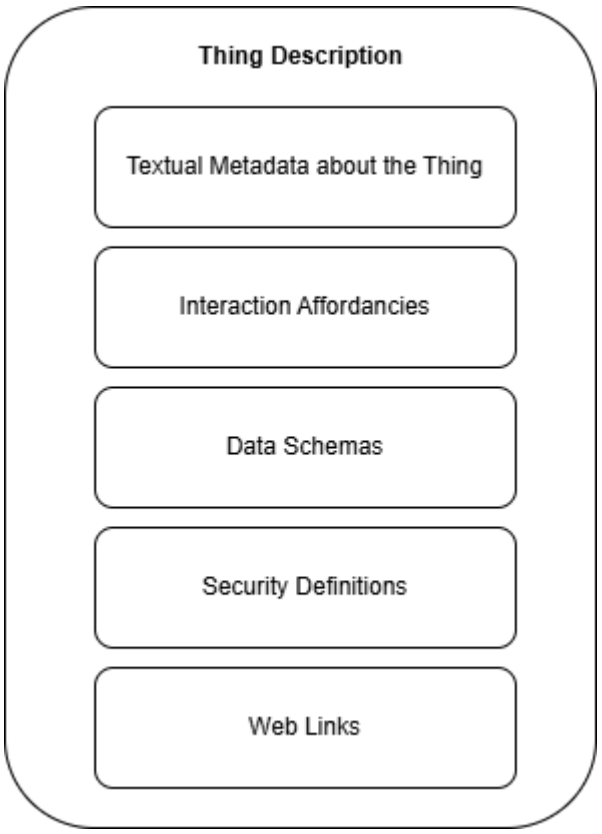


Figure1: W3C Thing Description Sectors

2.2.1 Textual Metadata

This sector is defined by twelve fields. Those fields are:

1. @context: Define shortcuts for names.
2. @type: Semantic data about the Thing.
3. id: Thing identifier.
4. title: Human readable title.
5. titles: Title alternatives for different languages.
6. description: Human readable description for the thing.

7. descriptions: Description alternatives in different languages.
8. version: TD version
9. created: Date of creation
10. modified: Date of modification
11. support: Maintainer uri
12. base: A uri based on which relative references are resolved.

From the above fields only the @context, and title are mandatory defined by the specification[1], the rest can be omitted. In Fig.2 there is an example of the Textual Sector as given from W3C documentation[1].

```
"@context": [
    "https://www.w3.org/2022/wot/td/v1.1",
    { "saref": "https://w3id.org/saref#" }
],
"id": "urn:uuid:300f4c4b-ca6b-484a-88cf-fd5224a9a61d",
"title": "MyLampThing",
"@type": "saref:LightSwitch",
```

Figure2: W3C TD Textual Component

2.2.2 Interaction Affordances

Interaction Affordances describe operations on Things or data that are sent and received from the Thing. Those affordances may have three types, Property Affordance, Action Affordance and Event Affordance.

Property Affordance is used when the data are of the form of value readings, for example temperature sensor value, lumen value etc. Based on the W3C documentation[1] the property affordance assumes HTTP GET method when accessing those values thus is a synchronous interaction. Below is defined the schema of the Property Affordance.

1. @type: Semantic tags.
2. title: Human readable title.

3. titles: Title language alternatives.
4. description: Human readable description.
5. descriptions: Description language alternatives.
6. forms: Defines a uri and protocol bindings for accessing the resource.
7. uriVariables: Define uri variables.
8. observable: Specify if the system shall allow clients to observe changes on a property.

Only the forms field is mandatory[1]. In Fig.3 there is an example of the Properties Affordance as presented from the documentation.

```
"properties": {
    "status": {
        "@type": "saref:OnOffState",
        "type": "string",
        "forms": [{
            "href": "https://mylamp.example.com/status"
        }]
    }
},
```

Figure3: W3C TD Properties Affordance

Action Affordance is used when the data are of the form of actions that the thing must perform, for example turn off the lights, correct a camera angle etc. Based on the W3C documentation the action affordance assumes HTTP POST method when accessing those actions thus is a synchronous interaction. Below is defined the schema of the Action Affordance.

1. @type: Semantic tags.
2. title: Human readable title.
3. titles: Title language alternatives.
4. description: Human readable description.
5. descriptions: Description language alternatives.
6. forms: Defines a uri and protocol bindings for accessing the resource.
7. uriVariables: Define uri variables.
8. input: Define input data schema.

9. output: Define output data schema.
10. safe: Used to define internal state change when invoked.
11. idempotent: Define if the action can be performed repeatedly with the same result based on the same input.
12. synchronous: Define if the action is synchronous or not. If it is asynchronous further querying must be performed to get the result.

Only the forms field is mandatory[1].

Based on the Action Affordance schema that defines synchronous actions and the documentation[1] that states that HTTP POST method is used for performing the action, it is obvious that when an asynchronous action is defined, to get the results, a polling mechanism shall apply. In Fig.4 we can see an example of Action Affordance as given from the official documentation [1].

```
"actions": {
    "toggle": {
        "@type": "saref:ToggleCommand",
        "forms": [{
            "href": "https://mylamp.example.com/toggle"
        }]
    }
},
```

Figure4: W3C TD Action Affordance

Event Affordance is used when the Thing must notify for the triggering of an event, for example when the temperature exceeds a set value. This communication format is asynchronous and has the following schema.

1. @type: Semantic tags.
2. title: Human readable title.
3. titles: Title language alternatives.
4. description: Human readable description.
5. descriptions: Description language alternatives.

6. forms: Defines a uri and protocol bindings for accessing the resource.
7. uriVariables: Define uri variables.
8. subscription: Data passed upon subscription for setup.
9. data: Defines the data schema for the messages the Thing will publish.
10. dataResponse: Defines the schema of the data in the response message. The response message is a message sent by the consumer of the event.
11. cancellation: Data to cancel a subscription.

Only the forms field is mandatory[1]. In Fig.5 we can see an example of Event Affordance as given from the documentation[1].

```
"events": {
    "overheating": {
        "data": {"type": "string"},
        "forms": [{
            "href": "https://mylamp.example.com/oh",
            "subprotocol": "longpoll"
        }]
    }
}
```

Figure5: W3C TD Event Affordance

2.2.3 Data Schemas

Data schemas are used to define the format of the data. The field is named *schemaDefinitions* and is a map[8]. This map will match the name of the schema to a specific schema defined by the DataSchema Class. The DataSchema class is a parent class to all available data types that can exist so a plethora of child classes exist. For the purposes of this thesis the extended presentation of the DataSchema class can be left out but for the reader to have a better understanding the subclasses will be named in Fig.6 below.

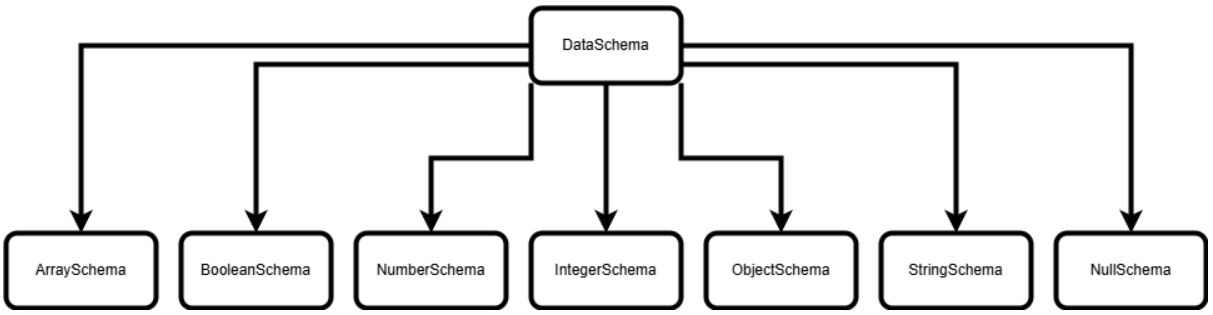


Figure6: W3C TD DataSchema Class and SubClasses

2.2.4 Security Definitions

Security definitions are a mandatory part of the W3C Thing Description[1]. In this sector the supported security mechanisms are defined. Those definitions will apply when communicating with the server. This sector consists of two fields, *securityDefinitions* and *security*. In the first field all the available security mechanisms are defined and in the second field is defined the security mechanism that is actually applied, example in Fig.7.

```
"securityDefinitions": {
    "basic_sc": {"scheme": "basic", "in": "header"}
},
"security": "basic_sc",
```

Figure7: W3C TD Security and Security Definitions

2.2.5 Web Links Sector

Finally, in this sector URIs are defined to external resources that relate to the current Thing Description. The field name is *links* and is an array of Link Class objects. The knowledge of the specific class structure is not mandatory for the purpose of the thesis so it will be omitted, nevertheless the reader can find this info at the references[1].

2.2.6 Final Schema

Now that the whole structure of the W3C Thing Description is presented it would be wise and useful to have it all in one place. In the table below the unified schema can be shown in Table1.

The *Field Name* is as defined in the TD, the *existence* shows if the field is mandatory as specified by the standard[1] and the *Type* is based on the types that are defined by the W3C.

Field Name	Existence	Type
@context	mandatory	anyURI / Array
@type	optional	string / Array of strings
id	optional	anyURI
titles	optional	Map of MultiLanguage
description	optional	string
descriptions	optional	Map of MultiLanguage
version	optional	VersionInfo
created	optional	dateTime
modified	optional	dateTime
support	optional	anyURI
base	optional	anyURI
properties	optional	Map of PropertyAffordance
actions	optional	Map of ActionAffordance
events	optional	Map of EventAffordance
links	optional	Array of Link
forms	optional	Array of Form
profile	optional	anyURI / Array of anyURI
schemaDefinitions	optional	Map of DataSchema
uriVariables	optional	Map of DataSchema
title	mandatory	string
security	mandatory	string / Array of string

Field Name	Existence	Type
@context	mandatory	anyURI / Array
@type	optional	string / Array of strings
id	optional	anyURI
securityDefinitions	mandatory	Map of SecurityScheme

Table1: W3C TD Schema

2.3 OpenAPI Thing Description

2.3.1 OpenAPI

OpenAPI[4] is a specification for defining interfaces of HTTP services in the form of OpenAPI Descriptions (OAD). It is language agnostic and the generated descriptions are both human and machine readable. This means that it can be used by tools for code generation, documentation generation, display API endpoints and its requirements and more.

The OpenAPI Description (OAD)[2] can be in a JSON or YAML format and has a specific structure that makes it easy for tools and humans to process it. Next, the structure of the description will be presented concisely, providing the reader with a comprehensive overview of the subject matter.

Field	Description
openapi	Defines the version of the OpenAPI specification.
info	Contains metadata about the API.
jsonSchemaDialect	Defines the JSON schema version in the form of URI.
servers	Contain the available servers that the endpoints are available.
paths	The API paths.
webhooks	Defines webhooks that may be implemented.

components	Contain reusable data.
security	Defines security mechanisms of the API.
tags	Contains the available tags that can be used in the description.
externalDocs	Defines whereabouts of external documentation.

Table2: OAD Structure

2.3.2 OpenAPI Thing Description for WoT

In this paragraph, we will analyze how OpenAPI[4] was utilized for the creation of Thing Descriptions. We will examine the structure of the respective fields and their significance. Subsequently, we will discuss the limitations related to the descriptions of things that are built using asynchronous communication methods.

Briefly, the fields include info, external documentation, servers, security, tags, paths, and components. Initially, the info field contains metadata that describes the Thing and is fully aligned with the info field of OpenAPI. Next, there is the external documentation field, which includes a URL providing additional documentation about the Thing. Following that, there is the security field, which specifies the security mechanisms that the Thing can support. The servers field lists the available servers through which the Thing is accessible. In contrast to the tags field in OpenAPI, the Thing Description supports only four available tags: Web Thing Resource Tag, Properties Resource Tag, Actions Resource Tag and Subscriptions Resource Tag. Similarly, the paths field has only four available paths: Web Thing Resource Operation, Properties Resource Operation, Actions Resource Operation and Subscriptions Resource Operation. Finally, the components field includes default values for schemas. Additionally, it contains reusable data that can be utilized within the Thing Description. Below is an image(Fig.8) illustrating the structure of the OpenAPI Thing Description as presented in the referenced research [2]. The colors are not of concern for the purposes of this thesis, but a comprehensive explanation can be found in the aforementioned paper.

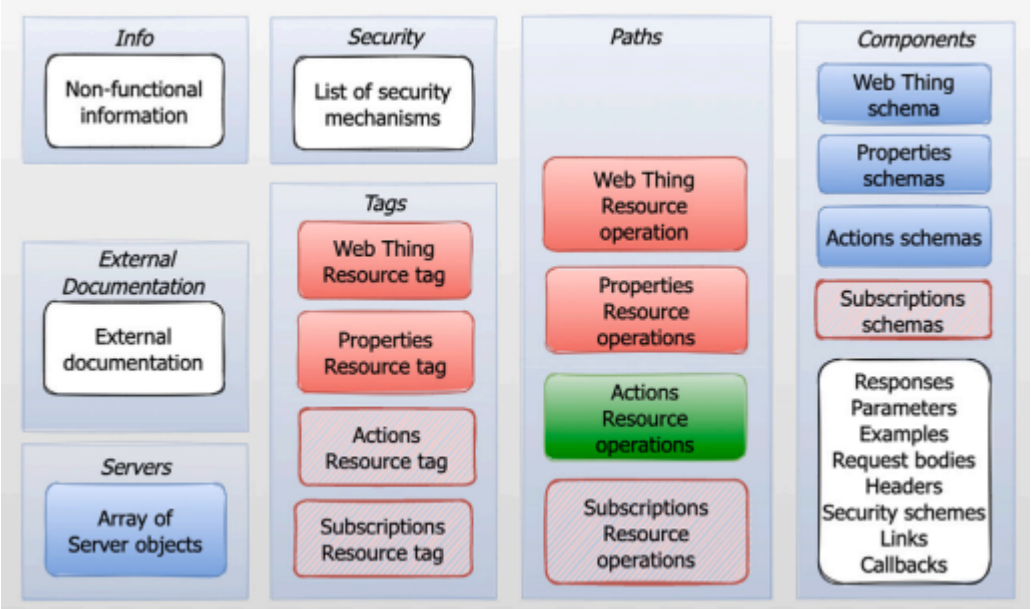


Figure8: OpenAPI TD Template

2.3.3 OpenAPI TD Asynchronous Limitations

As is already evident from the definition of OpenAPI, there are certain limitations regarding the support for asynchronous communication protocols. OpenAPI was designed with the goal of describing RESTful APIs, and similarly, the Thing Description created based on it treats all IoT devices as REST services. Thus, there is a need for the development of a standard capable of describing asynchronous protocols as well. In the following section, asynchronous communication and their structure will be thoroughly explained to provide a deeper understanding of this necessity.

2.4 Asynchronous Communication

2.4.1 Definition

Until recently, the most common communication method was synchronous, following the request-response model. However, there has been a shift towards asynchronous protocols such as Kafka, MQTT, and others. This transition is driven by their lightweight and efficient nature. Asynchronous protocols consume less power, as devices operating asynchronously can enter a sleep mode and wake up only when necessary depending on the protocol in use.

They also offer better scalability, as there is no need for synchronization with other network devices. Additionally, they reduce latency, allowing data to be sent and received as soon as they are ready, without blocking operations or requiring polling.

2.4.2 Brokers

In asynchronous communication, servers act as brokers. They define channels where messages are sent by producers. Subsequently, consumers have access to these messages. Below is an image(Fig.9 and Fig.10) illustrating the communication structure with brokers, producers, and consumers in a highly simplified form.

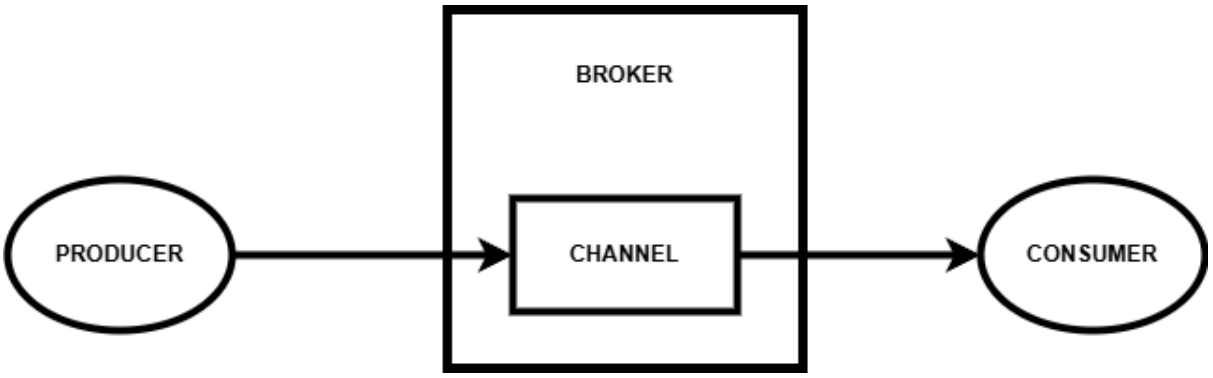


Figure9: Simple Broker Example.

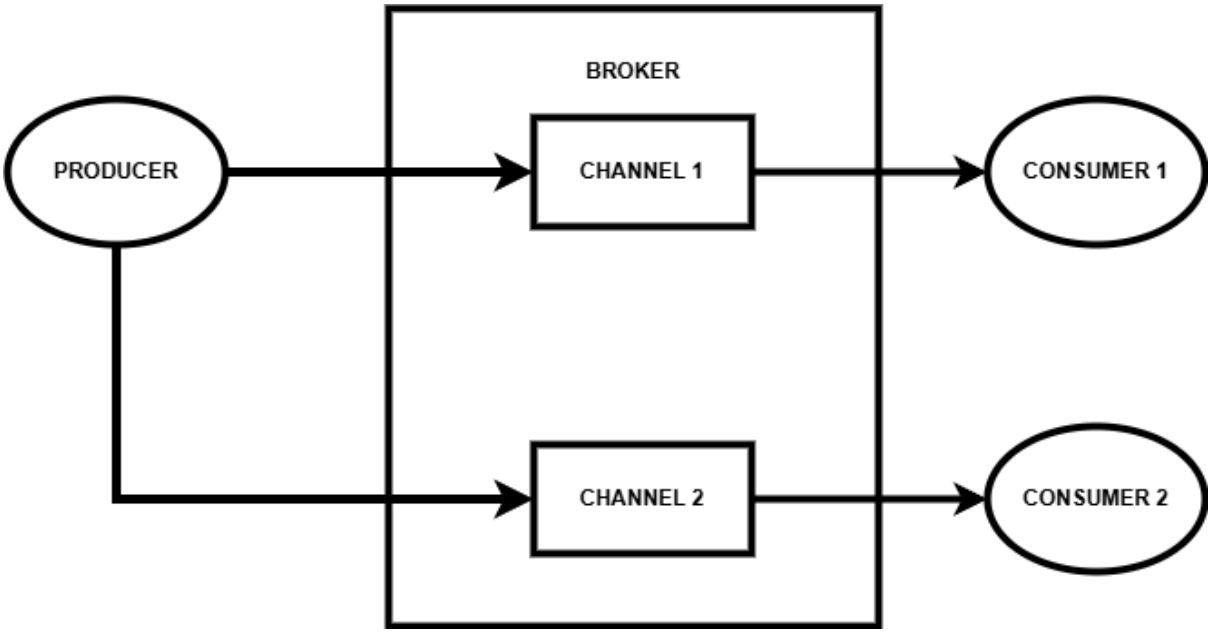


Figure10: More Complicated Communication Model.

2.4.3 Producers and Consumers

In this paragraph, producers and consumers will be analyzed. They have been examined together rather than in separate paragraphs because, in the complex world of the Web of Things (WoT), a producer can often act as a consumer and vice versa.

Producers are entities (applications or hardware) that observe events and publish them in the form of messages. For example, such an event could be a temperature increase in an industrial (or not) application.

The consumer is any entity (software or hardware) that subscribes to and monitors a specific event. It then retrieves and processes the data as defined.

In Fig.11 we can see an example of Producers and Consumers and how they communicate with the utilization of brokers through channels. There are cases where the Producer is also a Consumer and vice versa.

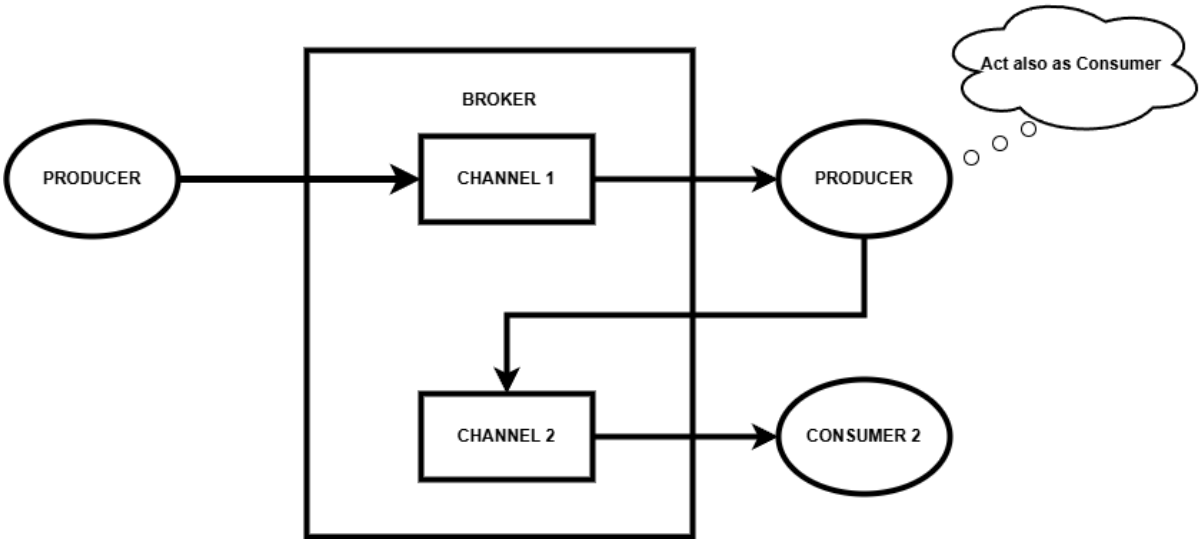


Figure11: Producer and Consumer Example.

2.4.4 Channels

A crucial aspect of the concepts discussed above is channels. Channels serve as mechanisms for organizing messages and are closely linked to the specific protocol being used. Depending on the context, they may also be referred to as topics, paths, routing keys, and other similar terms.

2.5 AsyncAPI

2.5.1 Definition

As mentioned earlier in the introduction, AsyncAPI is an open-source specification aimed at describing event-driven (asynchronous) services. The AsyncAPI document, which takes the form of JSON or YAML is used to describe event-driven services. In the following sections, the structure of this document will be thoroughly analyzed, as it will serve as the foundation for developing tools that can describe WoT systems using asynchronous protocols.

2.5.2 Structure

The structure of the AsyncAPI document is highly specific and consists of strictly defined json objects[18] that describe all the information needed for each service. At this point, a brief overview of the fields in the AsyncAPI JSON will be provided, giving the reader a general understanding. This will be followed by a more detailed analysis. The structure can be shown at the Table3 below.

Field	Type	Description
asyncapi	string	Contain the versions. Current 3.0.0
id	string	Application Identifier in the form of URI.
info	Info Object	Metadata about the service.
servers	Servers Object	Connection details about the brokers.
defaultContentType	string	Content type when message payload is not defined.
channels	Channels Object	Brokers channels.

operations	Operations Object	Defines what messages shall be sent or received.
components	Components Object	Reusable data across the document.

Table3: AsyncAPI Document Structure

2.5.2.1 Field: asyncapi

This field is required for a valid AsyncAPI document. It contains information about the version of the AsyncAPI specification being used, in the format *major.minor.patch*. This information is crucial for tools, as the specification is not backwards compatible.

2.5.2.2 Field: id

This field is not required. Provides an identification for the application. It must be unique and follows the URI format as defined in RFC3986[20]. An example is shown below in Fig.12 and Fig.13, as provided in the official documentation.

```
{
  "id": "https://github.com/smartylighting/streetlights-server"
}
```

Figure12: URL Example of id.

```
{
  "id": "urn:example:com:smartylighting:streetlights:server"
}
```

Figure13: URN Example of id.

2.5.2.3 Field: info

This field is required. It contains information about the service. It is a JSON object[18] with the structure shown in the Table4 below.

Field	Required	Type	Description
title	✓	string	Application title.
version	✓	string	Application Version.
description	✗	string	Application description.
termsOfService	✗	string	URL to Terms of Service.
contact	✗	Contact Object	Contact information.
license	✗	License Object	License Information for the application.
tags	✗	Tags Object	Tags for logical grouping.
externalDocs	✗	External Documentation Object | Reference Object	External documentation.

Table4: Info Object

For better readability the Contact Object, License Object, External Documentation Object and Reference Object structures can be found in Chapter 8. In the image below (Fig.14) an example of Info Object as provided from the official documentation is shown.

Figure14: Info Object Example.

2.5.2.4 Field: servers

The field *servers* is not mandatory for an AsyncAPI document[14] to be valid. Its purpose is to store information about the potential brokers to which the service can connect. Essentially, it is an object that contains objects of type server. Each key in *servers* object maps to a *server* object. Below (Table5), we will analyze the structure of this *server* object.

Field	Required	Type	Description
host	✓	string	Server Host name. Support variables in braces {}.
protocol	✓	string	The supported protocol by the server.
protocolVersion	✗	string	The protocol version.
pathname	✗	string	The path to the resource in the server. Support variables in braces {}.
description	✗	string	Server description.
title	✗	string	Human readable title.
summary	✗	string	A short summary.
variables	✗	Map <string, Server Variable Object | Reference Object >	Variables for host and pathname.
security	✗	List [Security Scheme Object | Reference Object]	Supported security mechanisms.
tags	✗	Tags Object	Tags for logical grouping.
externalDocs	✗	External Documentation Object | Reference Object	External documentation.
bindings	✗	Server Bindings Object | Reference Object	Protocol specific bindings.

Table5: Server Object

Server Variable Object, Security Scheme Object and Server Bindings Object definitions can be found in Appendices (Chapter 8). In Fig.15 below we can see an example of Servers Object.

Figure15: Servers Object Example.

2.5.2.5 Field: defaultContentType

This field is not mandatory. Its purpose is to provide a default content type if no specific one is defined in the message declaration. Example in Fig.16.

Figure16: Default Content Type Example.

2.5.2.6 Field: channels

This field is not mandatory[14]. It includes information about the available communication channels at the broker between the producers and consumers of the service. It is an object in which each key represents the id of a channel and corresponds to an object of type Channel Object. The structure of the Channel Object will be provided in Table6.

Field	Required	Type	Description
address	✗	string | null	Topic or Path (routing key) if any.
messages	✗	Messages Object	Available messages that can be sent to this channel.
title	✗	string	Human readable title.
summary	✗	string	Short summary.
description	✗	string	Channel description.
servers	✗	List [Reference Object]	Servers in which these channels are available.
parameters	✗	Parameters Object	Path parameters.
tags	✗	Tags Object	Tags for logical grouping.
externalDocs	✗	External Documentation Object | Reference Object	External documentation.
bindings	✗	Channel Bindings Object | Reference Object	Protocol specific definitions.

Table6: Channels Object

In Table7, the structure of the Messages Object will also be presented, as it is a highly important concept. Subsequently, we will provide examples of both the Channels and Messages Objects.

The Messages Object is a map[8] that maps messageIDs to Message Objects. Messages and Message Objects are two different things so we shall address this before continuing. Messages Object is an object in the root document while Message Object is an object that "lives" inside Messages Object and contains data about a specific message. So in this manner, a Message Object can be empty because for an AsyncAPI document to be valid we only need its messageID. Given that, in Table7, the structure of the Message object is presented, notice that all the fields are not required (can be empty) for a Message Object to be valid. Certain parts are beyond the scope of this thesis therefore, the reader may refer to the AsyncAPI documentation[14] for further information if needed.

Field	Required	Type	Description
headers	✗	Multi Format Schema Object | Schema Object | Reference Object	A schema of the message headers. This must not be protocol headers.
payload	✗	Multi Format Schema Object | Schema Object | Reference Object	Message payload schema.
correlationId	✗	Correlation ID Object | Reference Object	Message tracing.
contentType	✗	string	Message content type.
name	✗	string	Machine-readable name.
title	✗	string	Human readable title.
summary	✗	string	A short message summary.
description	✗	string	A verbose explanation of the message.
tags	✗	Tags Object	Tags for logical grouping.
externalDocs	✗	External Documentation Object | Reference Object	External documentation.
bindings	✗	Message Bindings Object | Reference Object	Protocol specific definitions.
examples	✗	List [Message Example Object]	A list with examples.

traits	✗	List [Message Trait Object | Reference Object]	A list of message traits.

Table7: Message Object

In Fig.17, Fig.18 and Fig.19 we present examples of Channels, Messages and Message Objects respectively as given from the official documentation of the AsyncAPI[14].
In Channels Object(Fig.17) we define the available messages, in this case userSignedUp, and we can see that for the body is used a Reference Object. In Fig.18 we see an example of Messages Object that contains two Message Objects, userSignedUp and userCompletedOrder. In Fig.19 we can see the actual info about the Message Object with messageID equal to userSignedUp.

```
{
  "userSignedUp": {
    "address": "user.signedup",
    "messages": {
      "userSignedUp": {
        "$ref": "#/components/messages/userSignedUp"
      }
    }
  }
}
```

Figure17: Channels Object Example.

```
{
  "userSignedUp": {
    "$ref": "#/components/messages/userSignedUp"
  },
  "userCompletedOrder": {
    "$ref": "#/components/messages/userCompletedOrder"
  }
}
```

Figure18: Messages Object Example.

Figure19: Message Object Example.

2.5.2.7 Field: operations

This field is not mandatory. It includes all the operations that the application can perform. At this point, it is defined which messages will be sent and which messages will be received. The basic structure of this object is referenced in the Table9 below and an example can be seen in Fig.20. For further information, the reader may refer to the official AsyncAPI documentation[14].

Field	Required	Type	Description
action	✓	"send" | "receive"	Use send when it's expected that the application will send a message to the given channel, and receive when the application should expect receiving messages from the given channel.
channel	✓	Reference Object	The channel this operation will perform in.
title	✗	string	Human friendly title.
summary	✗	string	Short summary.
description	✗	string	Description of the operation.
security	✗	List [Security Scheme Object | Reference Object]	Available security mechanisms.
messages	✗	List [Reference Object]	The messages this operation has access through the channel.
tags	✗	Tags Object	Logical grouping tags.
externalDocs	✗	External Documentation Object | Reference Object	External documentation.
bindings	✗	Operation Bindings Object | Reference Object	Protocol specific definitions.
reply	✗	Operation Reply Object | Reference Object	Definition for request-reply needs.
traits	✗	List [Operation Trait Object | Reference Object]	Operations traits.

Table9: Operation Object

```
{
  "title": "User sign up",
  "summary": "Action to sign a user up.",
  "description": "A longer description",
  "channel": {
    "$ref": "#/channels/userSignup"
  },
  "action": "send",
  "security": [
    {
     "petstore_auth": [
       "write:pets",
       "read:pets"
     ]
    }
  ],
  "tags": [
    { "name": "user" },
    { "name": "signup" },
    { "name": "register" }
  ],
  "bindings": {
    "amqp": {
      "ack": false
    }
  },
  "traits": [
    { "$ref": "#/components/operationTraits/kafka" }
  ],
  "messages": [
    { "$ref": "/components/messages/userSignedUp" }
  ],
  "reply": {
    "address": {
      "location": "$message.header#/replyTo"
    },
    "channel": {
      "$ref": "#/channels/userSignupReply"
    },
    "messages": [
      { "$ref": "/components/messages/userSignedUpReply" }
    ],
  }
}
```


Figure20: Operation Object Example.

2.5.2.8 Field: components

This field is not mandatory. Its purpose is to store information that is reusable throughout the document. A good practice is to keep message schemas, tags, external documentation, bindings, traits, security schemas etc., in this section of the document. In CodeSnippet1 we can see a part of a Components Object that contains Channels and Servers Objects.

```
"components": {
        "channels": {
                "signedup": {
                        “address”: "user/signedup",
                         "messages": {
                                "$ref": "#/components/messages/userSignUp"
                         }
                }
        },
        “servers”:{
                “development”:{
                        “host”: ”some/url/”,
                        “protocol” : “mqtt”
                }
        }
}
```

CodeSnippet1: Components Object Example.

3. Methodology

AsyncAPI[14] is a very useful tool with which we can describe asynchronous services. In this chapter, we will explore how we can adapt this specification to meet the needs of WoT. We will analyze which fields are required and must be used regardless of device type, i.e which should be included in the description and which must be modified to fit these requirements. Initially, we will create a template with which we will describe the devices. Then, we will construct two tools, one that will generate the Thing Descriptions (TDs) given a minimum user input and another that we will use to simulate sensor devices and will function as a playground for testing generation of Thing Descriptions (TDs) for sensor devices.

3.1 Template

The template is the most important stage in creating a Thing Description, as it determines how the capabilities of the Thing will be properly exposed. Initially, it should include all the necessary information required for the device, but it shouldn't be overly complicated to the point that the user gets confused. The structure must be clear and easily understandable. An essential aspect is that the users of the description are divided into two categories. The first category is the manufacturer, who defines the tangible capabilities of the device, and the second category is the end user, who needs to integrate the device into an already existing network, if applicable. Therefore, when creating the Thing Description, we must keep in mind the different needs of these two categories, and it should be strictly defined what each can modify within the description. For this reason, each section of the template is assigned a different color, so that the reader can easily understand which parts are the responsibility of the manufacturer and which are the responsibility of the user.

3.1.1 Template Fields

In this section, we will discuss in detail the fields supported by the template. We will examine how, starting from the general AsyncAPI structure, we have arrived at a more strict structure that can describe any WoT device. As expected, certain fields are mandatory, first to ensure the validity of the AsyncAPI document and second to give meaningful purpose to the creation of the description (an empty description would be pointless).

3.1.1.1 Field: asyncapi

This field is mandatory. The entire study of this thesis has been conducted based on version 3.0.0 of AsyncAPI specification. Therefore, the value of this field is fixed and does not change to ensure that the Thing Description can be integrated with existing tools. The field type is string, and its value is always set to 3.0.0.

3.1.1.2 Field: info

This field is mandatory and is set by the manufacturer of the Thing. It has been used as defined in the AsyncAPI specification[14], with one key difference. At this point, the user must declare the Tags and External Documentation if present.

3.1.1.3 Field: id

This field is not mandatory and follows the AsyncAPI specification[14]. The user (either the manufacturer or the end user) may declare an identifier for the Thing for practical purposes.

3.1.1.4 Field: servers

This field is important as it includes information about the servers/brokers with which the Thing can communicate. It follows the structure of the AsyncAPI specification[14]. Typically, this field is filled out by the end user who wishes to integrate the device into a network, as only they are aware of the topological characteristics of the infrastructure. It is rarely filled out by the manufacturer. More specifically, the user here declares the host, protocols, available security mechanisms, etc. Subsequently, all this information will be used by the channels and operations.

3.1.1.5 Field: channels

In this field, we will observe a significant difference compared to the AsyncAPI specification[14]. Channels can be defined either by manufacturer. Used to declare the available messages that the Thing can send or receive. Based on the research conducted, we concluded that only three channels are necessary. These channels can be seen below.
Web Thing Resource Channel, whose purpose is to allow the Thing to send messages containing metadata about it.

Properties Resource Channel, whose purpose is to send measurement values. It is used by sensors and is mandatory. It is mandatory because even an actuator must somehow communicate its state.

Actions Resource Channel, which is responsible for transferring messages to trigger an available action of the Thing. This channel is mandatory when the Thing is an actuator.

3.1.1.6 Field: operations

In a Thing Description, this is the point where it is declared which messages are sent and which are received. This field is set by the manufacturer. Just as in the case of channels, the number of operations has been limited to three, which are in absolute correspondence with the channels. Below, these operations are listed.

First, we have the Web Thing Resource Operation, which is responsible for sending messages to the broker through the corresponding channel and pertains to the metadata of the Thing.

Next, we have the Properties Resource Operation, which is responsible for sending, through the corresponding channel, messages related to sensor measurements or, more generally, the state of the Thing.

Finally, there is the Actions Resource Operation, which is responsible, through the corresponding channel, for receiving messages that affect the current state of the Thing. A simple example would be a message to turn on the lights in a room.

3.1.1.7 components

The last field we will analyze is components. Here, as in the AsyncAPI specification[14], the user can declare reusable data. Additionally, at this point, there are some schemas that are used as defaults in case the user does not provide a value. Finally, the user has the ability to extend the description by using the x- prefix for any user-defined fields.

3.1.2 Template Colors

The structure of the template is divided into color-coded sections, which define both the importance of each field and the user who is allowed to modify and edit it. The users are the manufacturer and the end user. However, there is also a virtual user, which is the template itself, containing certain default values to facilitate the process of creating a Thing Description. Next, we will explain these colors and provide an image of the template.

3.1.2.1 White Color

The white fields of the template usually refer to information that is not critical for the operation of the Thing and may be omitted from the description. However, some white fields may still contain essential information, such as the info field, where the user is required to specify the title and version. White fields are typically defined by the manufacturer, with the exception of certain bindings, which may be determined by the user integrating the device into their network.

3.1.2.2 Purple Color

The purple color indicates that this field is strictly defined by the template and cannot be modified by either the manufacturer or the end user. This restriction is in place because the template adheres to version 3.0.0 and follows its established structure. Any modification to this field would render the AsyncAPI document invalid, preventing its compatibility with the tools within the AsyncAPI ecosystem.

3.1.2.3 Blue Color

The blue areas indicate fields that are mandatory for the operation of the Thing. These fields are typically defined by the user who integrates the device into their network; however, the manufacturer may also define some fields, as this is not prohibited. For example, certain default message schemas or default messages may be specified. This does not, however, prevent the end user from modifying and adjusting them according to their needs. Furthermore, with regard to default message schemas and default messages, the template itself provides some to facilitate the process of creating Thing Descriptions.

3.1.2.4 Red Color

The red color signifies areas that are mandatory for the proper functioning of the Thing. These areas typically refer to the state of the Thing and may include measurements from potential sensors. This field is defined by the manufacturer, as only they are aware of the specific capabilities of each device. Additionally, it is a mandatory field, as even actuators have a certain state.

3.1.2.5 Green Color

The green color designates areas that are conditionally mandatory. These fields must be present if the Thing is a type of actuator. As with the red areas, these fields are defined by the manufacturer, who is aware of the device's capabilities.

3.1.2.6 Conclusion

Regarding the template and the colors that define certain properties, these are an effort to make the structure more strictly organized and compatible with the tools developed during this research. However, they simply serve as a guide for creating a Thing Description. Therefore, the final description that results may be more complex, as the template itself is by no means restrictive. On the contrary, its purpose is to assist anyone who wishes to describe a Thing in such a way that it is easily integrable into a network. In Fig.21 we can see an overview of the Template with the colors specified above.

Figure21: AsyncAPI Template.

3.2 Generator

The generator is a tool designed to create Thing Descriptions (TDs) based on the template. It achieves this by accepting a JSON[18] file from the user, which contains the essential information required to describe the Thing. The goal is to produce a complete description with minimal input. Next will explain the process by which the generator analyzes the input and generates the Thing Description. The input to the generator will be analyzed in Chapter 4, which deals with the implementation. For now, we will focus solely on the process followed by the generator. The process described below can be seen in Fig.22.

3.2.1 Initial Objects

Initially, the generator will scan the input to check if the basic objects are present. These objects are the asyncapi version, which is very important and must always be equal to 3.0.0.

Then, it will check that the info object is present, which contains two mandatory fields (title and version). Continuing, it will check that the servers with which the Thing can communicate have been provided, and finally, it will attempt to extract, if present, the Default Content Type, External Documentation, and Identifier.

3.2.2 Web Thing Resource

Continuing, the generator will check if the Web Thing Resource exists. This means that if it exists, it will need to generate the corresponding channels and operations in the asyncapi document and may also need to insert the corresponding tag. Additionally, it will need to check if the user has provided any specific message schemas or if the defaults will be used.

3.2.3 Properties Resource

This point is one of the most critical after the extraction of the basic objects. Regardless of whether the Web Thing Resource exists or not, the Properties Resource must be present. At this point, the user defines the messages that the thing can use to communicate its status, and may also define the schemas for these messages. If no schemas are defined, the defaults will be used. If the Properties Resource is not defined, the generator will produce an error.

3.2.4 Actions Resource

If the Thing we want to describe is an actuator, this is the point where the user will declare the messages with which we can intervene and change the state of the Thing. Additionally, the user may define the type of messages if they do not want to use the defaults. The generator will take this information and create the appropriate channels and operations in the asyncapi.

3.2.5 Additional Information

The user has the option to define additional information in the components section of the input JSON, which is not mandatory. The generator will scan and extract this information into the corresponding components field of the asyncapi.

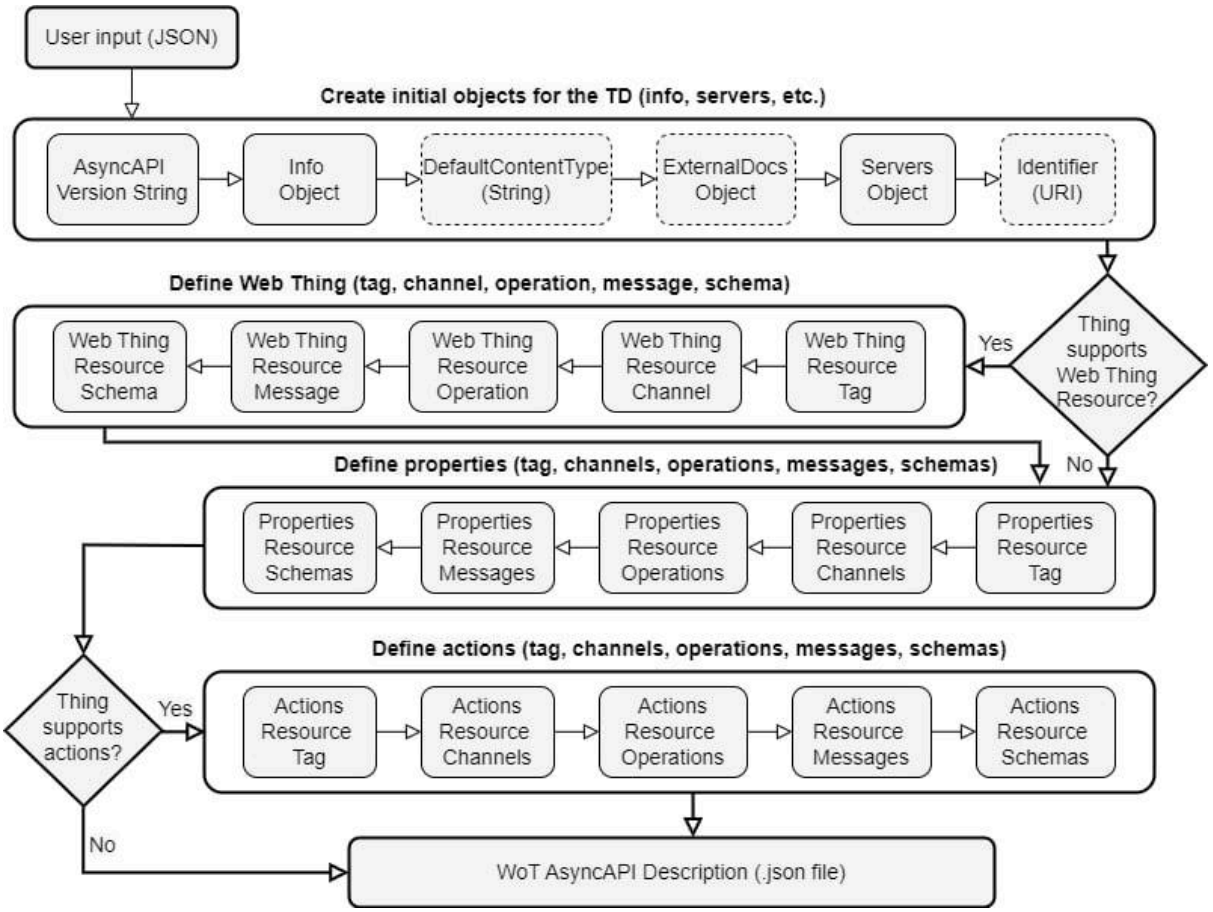


Figure22: Thing Description Generator.

3.3 Thing API Tool

Subsequently, having created the template and the generator, we will develop a tool that will generate the API of a sensor. Additionally, it will create a Thing Description for this sensor. It will have an HTTP interface so that it can connect to an actual device, allowing us to observe the messages it sends.

3.3.1 Basic Idea

The Thing API Tool will work as a playground for simulating sensors, so from now on the user implies the one using the tool - the one who wants to create a sensor. The user specifies, through a user interface, the available messages that the sensor can send. Additionally, they select the protocol supported by the device. Subsequently, the tool will generate the Thing Description in AsyncAPI and an interface in the HTTP protocol. Finally, the user will be able to use the description with all compatible tools of the AsyncAPI ecosystem, as well as utilize

the API either to simulate sending messages themselves (e.g., using tools like Postman) or to connect a device that will send the messages through the API. Finally, the user is provided with a UI where they can observe the messages being sent by the device.

3.3.2 Current State

For the Thing API Tool to work there must be implemented the core parts of the specified protocol, for example for KAFKA we need a Producer and Consumer that send and receive KAFKA messages and also a KAFKA broker. Currently, only the KAFKA requirements are met but in the future, implementations of other protocols can be added thus the functionality will be extended. In Chapter 4, we will analyze how the tool operates and provide examples.

4. Implementation

In this chapter, we will analyze the tools and technologies used within the framework of this thesis. Additionally, we will provide examples of the functionality of the tools that were developed to demonstrate their operation.

4.1 Tools and Technologies Utilized in Development

4.1.1 Python

Python[5] is a high-level, general-purpose programming language. It was chosen because it is easier to use due to its higher level of abstraction, as well as being one of the most common languages for developing web applications (others may include Java, PHP, and more). Additionally, it offers robust frameworks such as Flask[6] and FastAPI[7], which have been tested in real-world environments and have proven to be highly reliable and flexible. Therefore, we used the Flask and FastAPI frameworks for the backend of the tools we developed, as we wanted these tools to function as web applications.

4.1.1.1 Flask

Flask[6] is a lightweight and flexible Python framework designed to make the creation of web applications and REST APIs easier. It was chosen for its minimal setup requirements.

4.1.1.2 FastAPI

FastAPI[7] is a modern framework used for creating APIs. It is easy to use and fast. It includes data validation and automatic generation of documentation for the APIs built with it, which is the reason it was chosen.

4.1.2 Typescript/Javascript

JavaScript[8] is a high-level programming language primarily used for creating websites. It enables the programmer to create interactive websites by dynamically manipulating HTML and CSS. TypeScript[8] is a superset of JavaScript that enforces stricter rules regarding code structure. It was created by Microsoft and compiles into JavaScript.

4.1.2.1 ReactJS

ReactJS[10] is a Typescript/Javascript library for creating user interfaces. It was developed by the company Meta (Facebook) and makes the creation of frontend applications much easier, as the developer does not need to use HTML to build the website. Additionally, it can produce responsive UIs more easily. The main purpose of ReactJS is to create single-page applications, so it must be combined with NextJS[11] to enable routing in our application.

4.1.2.2 NextJS

Since an application typically requires more than one page to be more organized, there is a need for a form of routing. NextJS[11] is a framework built on ReactJS that provides routing capabilities. Additionally, it offers server-side rendering, which reduces response times and loading times for the application. Additionally, it has modules such as TailwindCSS[21], which allow us to create beautiful UIs with less effort.

4.1.3 NodeJS

JavaScript, and consequently TypeScript, run in the browser. To use them on the server (ReactJS, NextJS), another tool called NodeJS[9] is required. NodeJS is a runtime environment that allows us to run JavaScript like any other interpreted programming language.

4.1.4 Docker

We wanted the tools we created to be easy to deploy, so we decided to use Docker[12]. Docker is a tool that allows developers to build containers containing their applications. These containers are lightweight virtual machines that include only the necessary dependencies for the application to function. This makes deployment and portability of applications easier, as they are now independent of the machine they run on.

4.2 Developed Tools and Their Functionality

4.2.1 Generator

The implementation of the generator is divided into two parts. In the first part, we have the backend, whose role is to generate a Thing Description based on the given input. In the second part, we have a user interface to make the process more user-friendly.

4.2.1.1 Generator Input File

First, we will analyze the JSON file that serves as the input to the generator. The JSON contains thirteen fields, some of which are mandatory while others are optional. This distinction arises from the rules we have established in the template, and these rules represent the minimum input required to produce a valid AsyncAPI Thing Description. Notice that this is the input of the generator and not the Thing Description itself.

4.2.1.1.1 Field: info

This field contains metadata regarding the Thing and must be provided from the manufacturer. The presence of this field is mandatory, and in the Fig.23 below we can see the schema of this field.

Figure23: User Input JSON Field:info schema.

4.2.1.1.2 Field: servers

This field pertains to the available brokers to which the Thing can send messages. It is an array of server objects as defined in the AsyncAPI specification[14] with an extra field for defining the channel, and its presence is mandatory. Usually specified by the end user. In the CodeSnippet2 the schema of the Servers Object is presented.

```
"servers": {
        "type": "array",
        "items": {
           "additionalProperties": false,
           "required": [ "name", "host", "protocol"  ],
           "type": "object",
           "properties": {
              "for": {
                 "description": "Define the channel that this server  works with.",
                 "type": "string",
                 "enum": [
                    "all",
                    "webthing_resource_channel",
                    "properties_resource_channel",
                    "actions_resource_channel",
                    "webthing_resource_channel_&_properties_resource_channel",
                    "webthing_resource_channel_&_actions_resource_channel",
                    "properties_resource_channel_&_actions_resource_channel"
                 ]
              },
              "name": {
                 "description": "This is the server ID.",
                 "type": "string"
              },
              "host": {
                 "type": "string"
              },
              "protocol": {
                 "type": "string"
              },
```

```
    "pathname": {
      "type": "string"
    },
    "description": {
      "type": "string"
    },
    "title": {
      "type": "string"
    },
    "summary": {
      "type": "string"
    },
    "variables": {
      "description": "In this object the keys MUST be the variables in the path.",
      "type": "object",
      "properties": {
        "enum": {
          "type": "array",
          "item": {
            "type": "string"
          }
        },
        "default": {
          "type": "string"
        },
        "description": {
          "type": "string"
        },
        "examples": {
          "type": "array",
          "item": {
            "type": "string"
          }
        }
      }
    }
  },
```

```
        "security": {
          "description": "The supported security schemas of the server.
                          If this property exist then the root   property: serverSecuritySchemas must be set",
          "type": "array",
          "items": {
            "type": "string",
            "enum": [
              "userPassword",
              "apiKey",
              "X509",
              "symmetricEncryption",
              "asymmetricEncryption",
              "httpApiKey",
              "http",
              "oauth2",
              "openIdConnect",
              "plain",
              "scramSha256",
              "scramSha512",
              "gssapi"
            ]
          }
        },
        "bindings": {
          "description": "Server Bindings as defined in Asyncapi specification 3.0.0",
          "type": "object"
        }
      }
    }
```

CodeSnippet2: Servers Object Schema.

4.2.1.1.3 Field: tags

This field is not mandatory, can be defined by either manufacturer or end user and includes available tags for logical grouping. Inside the AsyncAPI TD will be located in the info object. In Fig.24 the schema of the field is presented.

Figure24: User Input JSON Field: tags schema.

4.2.1.1.4 Field: externalDocs

This field is not mandatory. It provides information about external documentation for the Thing and may be provided by either the manufacturer or the end user. In the AsyncAPI Thing Description, it will be embedded within the info object. The schema can be seen in Fig.25.

```
"externalDocs": {
    "type": "object",
    "properties": {
        "description": {
            "type": "string"
        },
        "url": {
            "type": "string"
        }
    }
},
```

Figure25: User Input JSON Field: externalDocs schema.

4.2.1.1.5 Field: web_thing_resource

In this field, the manufacturer defines the messages of the Thing, which include metadata about it. The presence of this field is not mandatory, and it is typically used in special cases. If this field exists, when provided as input to the generator, it will trigger the creation of a channel and an operation through which the Thing can communicate these messages. Schema provided in Fig.26.

```
"web_thing_resource": {
    "description": "Metadata about the thing",
    "type": "array",
    "items": {
        "type": "string"
    }
},
```

Figure26: User Input JSON Field: web_thing_resource schema.

4.2.1.1.6 Field: web_thing_state

The presence of this field is mandatory and is provided by the manufacturer. It contains messages that include the state of the device; if the device is a sensor, it will contain some measurements, and if it is an actuator, it will describe the current state of the Thing. This field triggers the creation of a channel and an operation, which are used for communication with the server/broker. The schema can be seen in Fig.27.

```
"web_thing_state": {
    "description": "Messages about the Thing state (a.k.a. sensor measurements or actuator status)",
    "type": "array",
    "items": {
        "type": "string"
    }
},
```

Figure27: User Input JSON Field: web_thing_state schema.

4.2.1.1.7 Field: web_thing_actions

This field is mandatory and provided by the manufacturer if the Thing is an actuator. It contains messages that allow us to change the state of the device (for example, turn off). As with the two previously mentioned fields, its presence triggers the creation of a channel and an operation for communicating the necessary messages. Schema can be seen in Fig.28.

```
"web_thing_actions": {
    "description": "Available actions that the thing can perform",
    "type": "array",
    "items": {
        "type": "string"
    }
},
```

Figure28: User Input JSON Field: web_thing_actions schema.

4.2.1.1.8 Field: deviceType

This field is not mandatory and can be provided by either manufacturer or end user, as this information can also be derived from the declaration of the messages. However, it is good to include it for documentation reasons and potential future expansion of the generator. The

values it can take are *sensor, actuator, or sensor&actuator*. We can see its structure in Fig.29 below.

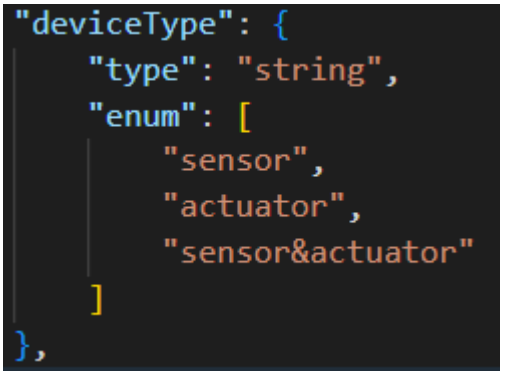


Figure29: User Input JSON Field: deviceType schema.

4.2.1.1.9 Field: serverSecuritySchemas

This field is mandatory only if we have defined a security mechanism in the server object. This field must be set by the end user that will define the servers. Its structure strictly follows the AsyncAPI specification[14] and the structure is provided in Fig.30.

Figure30: User Input JSON Field: serverSecuritySchemas schema.

4.2.1.1.10 Field: messagesSchemas

This field is not mandatory. It allows the manufacturer to define the structure of the messages so that the default value provided by the generator is not used. In any case, since each device is unique, it would be better if the user specified the structure of the messages. The schema of this field is presented in Fig.31.

Figure31: User Input JSON Field: messagesSchemas schema.

4.2.1.1.11 Field: id

The presence of this field is not mandatory and allows the manufacturer to create a unique ID for the Thing. Schema in Fig.32.

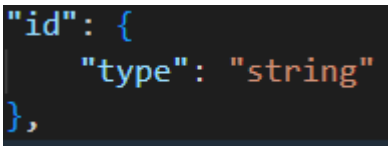


Figure32: User Input JSON Field: id schema.

4.2.1.1.12 Field: defaultContentType

This field is not mandatory, and the manufacturer can use it if the majority of the messages characterizing the Thing have the same content type, so that they do not need to declare all the headers in the messageSchemas. The schema can be seen below in Fig.33

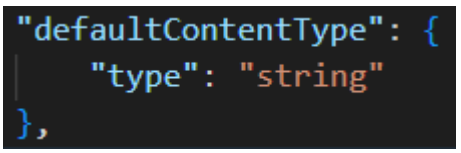


Figure33: User Input JSON Field: defaultContentType schema.

4.2.1.1.13 Field: components

This field is not mandatory and follows the structure of the components object in AsyncAPI specification [14]. The manufacturer or the end user can declare reusable objects that will be used in the input file or extend the final description, as any field starting with *x-* will be transferred as-is to the AsyncAPI TD. The component field must not contain any of the previously mentioned fields, if anything must be moved into components object at the final AsyncAPI TD, the generator will place it in the correct position. The schema can be seen below in Fig.34.

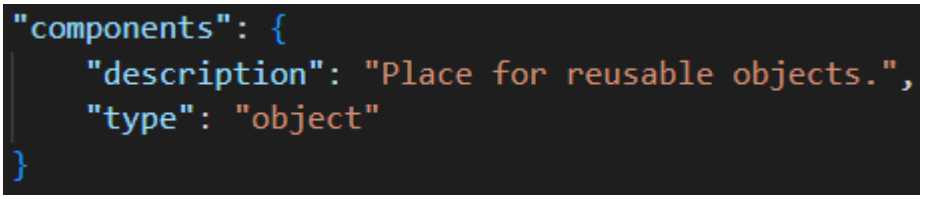


Figure34: User Input JSON Field: components schema.

4.2.1.1.14 Input Example

In CodeSnippet3 an example is presented of a Panasonic Air Conditioner. This example is based on the W3C TDs repository[15]. We have defined the basic info, title and version also we have given an id. In servers object we set the name, the host, the protocol, and a pathname in the server. This server does not support any security schemas. In addition we define the *for* field as this server is used for the properties channel. We added the available messages in *web_thing_state* and lastly we defined the schemas of these messages in *messagesSchemas*. That is all the information needed to generate the AsyncAPI Description.

```
{
  "info": {
    "title": "PanasonicAirConditionerP1",
    "version": "1.0"
  },
  "id": "urn:uuid:ed0392cc-3109-48d0-bfd2-3818e2528c78",
  "servers": [
    {
      "for": "properties_resource_channel",
      "name": "base",
      "host": "w3c.p-wot.com",
      "protocol": "mqtt",
      "pathname": "/wot-ver2/things/airconditioner/1/",
      "security": []
    }
  ],
  "web_thing_state": [
    "operationStatus",
    "operationMode",
    "desiredTemp",
    "windVolumeLevel",
    "change"
```

```
],
"messagesSchemas": [
  {
    "messageID": "change",
    "channelID": "properties_resource_channel",
    "headers": {
      "type":"object",
      "properties":{
        "contentType": {
          "type": "string",
          "const": "application/json"
        }
      }

    },
    "payload": {
      "type": "object",
      "properties": {
        "operationStatus": {
          "type": "boolean"
        },
        "operationMode": {
          "type": "string",
          "enum": [
            "Auto",
            "Cooling",
            "Heating",
            "Dehumidifying",
            "Blast"
          ]
        },
```

```
            "desiredTemp": {
               "type": "number",
               "minimum": 16,
               "maximum": 30
            },
            "windVolumeLevel": {
               "type": "number",
               "minimum": 0,
               "maximum": 8
            }
         }
      }
   },
   {
      "messageID": "windVolumeLevel",
      "channelID": "properties_resource_channel",
      "headers": {
         "type":"object",
         "properties":{
            "contentType": {
               "type": "string",
               "const": "application/json"
            }
         }
      },
      "payload": {
         "type": "number",
         "minimum": 0,
         "maximum": 8
      }
   },
```

```
{
   "messageID": "desiredTemp",
   "channelID": "properties_resource_channel",
   "headers": {
      "type":"object",
      "properties":{
         "contentType": {
            "type": "string",
            "const": "application/json"
         }
      }
   },
   "payload": {
      "type": "number",
      "minimum": 16,
      "maximum": 30
   }
},
{
   "messageID": "operationMode",
   "channelID": "properties_resource_channel",
   "headers": {
      "type":"object",
      "properties":{
         "contentType": {
            "type": "string",
            "const": "application/json"
         }
      }
   },
   "payload": {
```

```
            "type": "string",
            "enum": [
              "Auto",
              "Cooling",
              "Heating",
              "Dehumidifying",
              "Blast"
            ]
          }
        },
        {
          "messageID": "operationStatus",
          "channelID": "properties_resource_channel",
          "headers": {
            "type":"object",
            "properties":{
              "contentType": {
                "type": "string",
                "const": "application/json"
              }
            }
          },
          "payload": {
            "type": "boolean"
          }
        }
    ]
}
```

CodeSnippet3: Generator Input Example.

In the CodeSnippet4 we can see the AsyncAPI Thing Description that is created from the Generator given the input as shown in CodeSnippet3. Notice that the Generator puts everything in the correct position so the user is not concerned when providing the input. Because we only have sensory data the channels have once entry, the properties_resource_channel and the operations have one entry properties_resource_operations. Messages are stored in components with their respective schemas.

```
{
  "asyncapi": "3.0.0",
  "info": {
    "title": "PanasonicAirConditionerP1",
    "version": "1.0"
  },
  "servers": {
    "base": {
      "host": "w3c.p-wot.com",
      "protocol": "mqtt",
      "pathname": "/wot-ver2/things/airconditioner/1/",
      "description": "",
      "title": "",
      "summary": "",
      "variables": {},
      "security": [],
      "bindings": {}
    }
  },
  "channels": {
    "properties_resource_channel": {
      "messages": {
        "operationStatus": {
```

```
            "$ref": "#/components/messages/operationStatus"
          },
          "operationMode": {
            "$ref": "#/components/messages/operationMode"
          },
          "desiredTemp": {
            "$ref": "#/components/messages/desiredTemp"
          },
          "windVolumeLevel": {
            "$ref": "#/components/messages/windVolumeLevel"
          },
          "change": {
            "$ref": "#/components/messages/change"
          }
        },
        "servers": [
          {
            "$ref": "#/servers/base"
          }
        ]
      }
  },
  "operations": {
    "properties_resource_operation": {
      "messages": [
        {
          "$ref": "#/channels/properties_resource_channel/messages/operationStatus"
        },
        {
          "$ref": "#/channels/properties_resource_channel/messages/operationMode"
        },
```

```
        {
          "$ref": "#/channels/properties_resource_channel/messages/desiredTemp"
        },
        {
          "$ref": "#/channels/properties_resource_channel/messages/windVolumeLevel"
        },
        {
          "$ref": "#/channels/properties_resource_channel/messages/change"
        }
      ],
      "action": "send",
      "channel": {
        "$ref": "#/channels/properties_resource_channel"
      }
    }
  },
  "components": {
    "messages": {
      "operationStatus": {
        "headers": {
          "$ref": "#/components/schemas/operationStatus_headers"
        },
        "payload": {
          "$ref": "#/components/schemas/operationStatus_payload"
        }
      },
      "operationMode": {
        "headers": {
          "$ref": "#/components/schemas/operationMode_headers"
        },
        "payload": {
```

```
            "$ref": "#/components/schemas/operationMode_payload"
          }
        },
        "desiredTemp": {
          "headers": {
            "$ref": "#/components/schemas/desiredTemp_headers"
          },
          "payload": {
            "$ref": "#/components/schemas/desiredTemp_payload"
          }
        },
        "windVolumeLevel": {
          "headers": {
            "$ref": "#/components/schemas/windVolumeLevel_headers"
          },
          "payload": {
            "$ref": "#/components/schemas/windVolumeLevel_payload"
          }
        },
        "change": {
          "headers": {
            "$ref": "#/components/schemas/change_headers"
          },
          "payload": {
            "$ref": "#/components/schemas/change_payload"
          }
        }
      },
      "schemas": {
        "change_headers": {
          "type": "object",
```

```
    "properties": {
      "contentType": {
        "type": "string",
        "const": "application/json"
      }
    }
  },
  "change_payload": {
    "type": "object",
    "properties": {
      "operationStatus": {
        "type": "boolean"
      },
      "operationMode": {
        "type": "string",
        "enum": [  "Auto", "Cooling",  "Heating",  "Dehumidifying",  "Blast"  ]
      },
      "desiredTemp": {
        "type": "number",
        "minimum": 16,
        "maximum": 30
      },
      "windVolumeLevel": {
        "type": "number",
        "minimum": 0,
        "maximum": 8
      }
    }
  },
  "windVolumeLevel_headers": {
    "type": "object",
```

```
    "properties": {
      "contentType": {
        "type": "string",
        "const": "application/json"
      }
    }
  },
  "windVolumeLevel_payload": {
    "type": "number",
    "minimum": 0,
    "maximum": 8
  },
  "desiredTemp_headers": {
    "type": "object",
    "properties": {
      "contentType": {
        "type": "string",
        "const": "application/json"
      }
    }
  },
  "desiredTemp_payload": {
    "type": "number",
    "minimum": 16,
    "maximum": 30
  },
  "operationMode_headers": {
    "type": "object",
    "properties": {
      "contentType": {
        "type": "string",
```

```
                "const": "application/json"
              }
            }
          },
          "operationMode_payload": {
            "type": "string",
            "enum": ["Auto",  "Cooling",  "Heating",  "Dehumidifying", "Blast"
            ]
          },
          "operationStatus_headers": {
            "type": "object",
            "properties": {
              "contentType": {
                "type": "string",
                "const": "application/json"
              }
            }
          },
          "operationStatus_payload": {
            "type": "boolean"
          }
        }
      },
      "id": "urn:uuid:ed0392cc-3109-48d0-bfd2-3818e2528c78"
}
```


CodeSnippet4: Generator Result TD.

4.2.1.2 Generator Backend

In this paragraph, we will analyze, given the input, how the generator processes it to create an AsyncAPI Thing Description. To help the reader better understand, there will be code snippets, explanations of the classes that were created, and possibly explanations of some libraries.

The core idea is that we wanted the generator to be a web service, so the Flask library was used to make the process easier. The primary step is to create the endpoint where the user sends the input JSON. Therefore, we create a blueprint(Fig.35) to better organize the code. Continuing, we use this blueprint to create the endpoint(Fig.36).

```
asyncapidoc = Blueprint('asyncapi_doc', __name__)
```

Figure35: Flask Blueprint creation.

```
@asyncapidoc.route('/singleinput/<docID>', methods=['PUT'])
```

Figure36: Endpoint creation.

In Fig.36, we see that we use a Python decorator (in other languages, it is also known as a macro) which first specifies the path and then the HTTP method it responds to. The path has two parts, the first is simply the route, while the second (<docID>) is particularly significant as it represents the name of the description. Next, we need to provide the implementation of the endpoint(Fig.37), i.e., how it should extract and process any data provided by the user. This is done by providing a function, or method, immediately after the decorator.

```python
@asyncapidoc.route('/singleinput/<docID>', methods=['PUT'])
def createDocumentFromSingleInput(docID):
    try:
        usrInput = request.get_json() # Get user input as dictionary
        usrInputPrs = uiParser.UserInputParser(usrInput) # Extract all the objects
        newAsyncApiDoc = usrInputPrs.convert2Asyncapi() # Convert to Asyncapi object
        if len(newAsyncApiDoc) == 0:
            return jsonify({
                "msg":"fail",
                "description":"Failed to generate the asyncapi document. Check the input and try again."
                })
        dbe = DatabaseSim()
        dbe.StoreData(docID, newAsyncApiDoc)
        return jsonify({"msg":"success", "data":newAsyncApiDoc})
    except Exception as e:
        print(e)
        traceback.print_exc()
        return jsonify({'msg': 'fail'})
```

Figure37: Endpoint functionality.

We will now analyze this function. Initially, with the *get_json()* function, we retrieve the user's input. Next, we have constructed a parser that will process the input. Therefore, at this point, it would be appropriate to explain how the parser works for completeness.

Initially, the parser(Fig.38) is a class where, when creating an instance, we must provide the constructor with a Python dictionary that follows the structure of the user input. Additionally, in the constructor, some other values are initialized, which are used later to create a valid AsyncAPI description.

```python
def __init__(self, theInput):
    self.user_input = theInput      You, 2 months ago • Jan 8 2025 …
    self.defaultServer = self.user_input["servers"][0]["name"]
    self.serverToChannelMap = {"webthing_resource_channel":[],
                               "properties_resource_channel":[],
                               "actions_resource_channel":[]
                               }
    self.asyncapi_output = {"asyncapi":"3.0.0",
                            "info":{},
                            "servers":{},
                            "channels":{},
                            "operations":{},
                            "components":{}
                            }
```

Figure38: Parser Constructor.

Next, we have the *convert2Asyncapi()* method (Fig.39). This method will validate and extract the input data and return an AsyncAPI description. This is achieved by calling another method where all the core work is done, this method is *validateAndExtract()*.

```
def convert2Asyncapi(self):
    if self.validateAndExtract() :
        return self.asyncapi_output
    else:
        return {}       You, 2 months
```

Figure39: Parser AsyncAPI creation.

If the validateAndExtract() method (CodeSnippet5) does not detect any errors, it will create and store the AsyncAPI Description in the appropriate variable of the class. Then, it will return the description so that the endpoint's operation can continue. Next, we will briefly analyze how the aforementioned function works to make its significance more understandable.

The first step is to check that the user has indeed defined the dictionary representing the input. This is done because, in Python[5], there are no private variables, so even after the constructor is called, the programmer can intervene and change the value. Therefore, this is a critical point where, if the user's input is missing, the validation will fail, and the description will not be created. Next, we will first extract the info object, which we can take as-is since it follows the AsyncAPI specification. Additionally, we extract the servers, which require some minor processing before being included in the dictionary representing the Thing Description. This is done by calling self.extractServers(self.user_input["servers"]). Next, we will create the channels. For this purpose, a method has been developed that accepts as arguments the name of the channel and the corresponding array of available messages, channelGenerator(). According to this, it follows that this function will be called three times for the arguments: properties_resource_channel, actions_resource_channel, and webthing_resource_channel. Similarly, we have also created a function/method that will generate the necessary operations. This function is called operationGenerator() and takes as arguments the names of the operations and the array with the corresponding messages. Therefore, we understand that this function will also be called three times for the following

arguments: properties_resource_operation, actions_resource_operation, and webthing_resource_operation. Next, we extract the defaultContentType, id, and externalDoc as provided by the input, since they follow the AsyncAPI specification and do not require any processing. For the tags, messagesSchemas, and components, the necessary methods have been developed to extract and store the input data correctly. Therefore, we have the following functions respectively: *extractTags()*, E*xtractMessagesSchemas()*, and e*xtractNonMandatoryComponents()*. The name of the last one is slightly misleading, but it means that the objects it extracts are not among those that have already been extracted.

```
def validateAndExtract(self):
    if not self.inputIsSet():
        return False
    isValid = True
    try:
        if "info" not in self.user_input:
            return False
        self.asyncapi_output["info"] = self.user_input["info"]
        if "servers" not in self.user_input:
            return False
        self.asyncapi_output["servers"] = self.extractServers(self.user_input["servers"])
        # Generate the channels
        if "web_thing_state" in self.user_input:
            self.channelGenerator("properties_resource_channel",
                                  self.user_input["web_thing_state"])
        if "web_thing_actions" in self.user_input:
            self.channelGenerator("actions_resource_channel",
                                  self.user_input["web_thing_actions"])
        if "web_thing_resource" in self.user_input:
            self.channelGenerator("webthing_resource_channel",
                                  self.user_input["web_thing_resource"])
```

```
        # Generate the operations
    if "web_thing_state" in self.user_input:
        self.operationGenerator("properties_resource_operation",
                                self.user_input["web_thing_state"])
    if "web_thing_actions" in self.user_input:
        self.operationGenerator("actions_resource_operation",
                                self.user_input["web_thing_actions"])
    if "web_thing_resource" in self.user_input:
        self.operationGenerator("webthing_resource_operation",
                                self.user_input["web_thing_resource"])
    if "defaultContentType" in self.user_input:
        self.asyncapi_output["defaultContentType"] = self.user_input["defaultContentType"]
    if "id" in self.user_input:
        self.asyncapi_output["id"] = self.user_input["id"]
    if "externalDocs" in self.user_input:
        self.asyncapi_output["components"]["externalDocs"] =
        {"externalDocs": self.user_input["externalDocs"]}

    if "tags" in self.user_input:
        self.asyncapi_output["components"]["tags"] = self.extractTags(self.user_input["tags"])
    if "messagesSchemas" in self.user_input:
        self.ExtractMessagesSchemas()
    if "components" in self.user_input:
        self.extractNonMandatoryComponents()
except Exception as e:
    isValid = False
    print(e)
    traceback.print_exc()
return isValid
```

CodeSnippet5: validateAndExtract() method.

Continuing now with the functionality of the endpoint, when the validation is complete and we have a valid AsyncAPI object, we will use a class called DatabaseSim, whose purpose is to simulate interaction with a database. This is done to allow for potential future expansion using a relational or non-relational database. For now, it stores the descriptions in simple JSON files. Once this step is also completed, the endpoint will return a JSON with an AsyncAPI Thing Description. Below (Fig.40), we can see a diagram illustrating the operation of the endpoint, where the two red parallelograms represent the two possible final states.

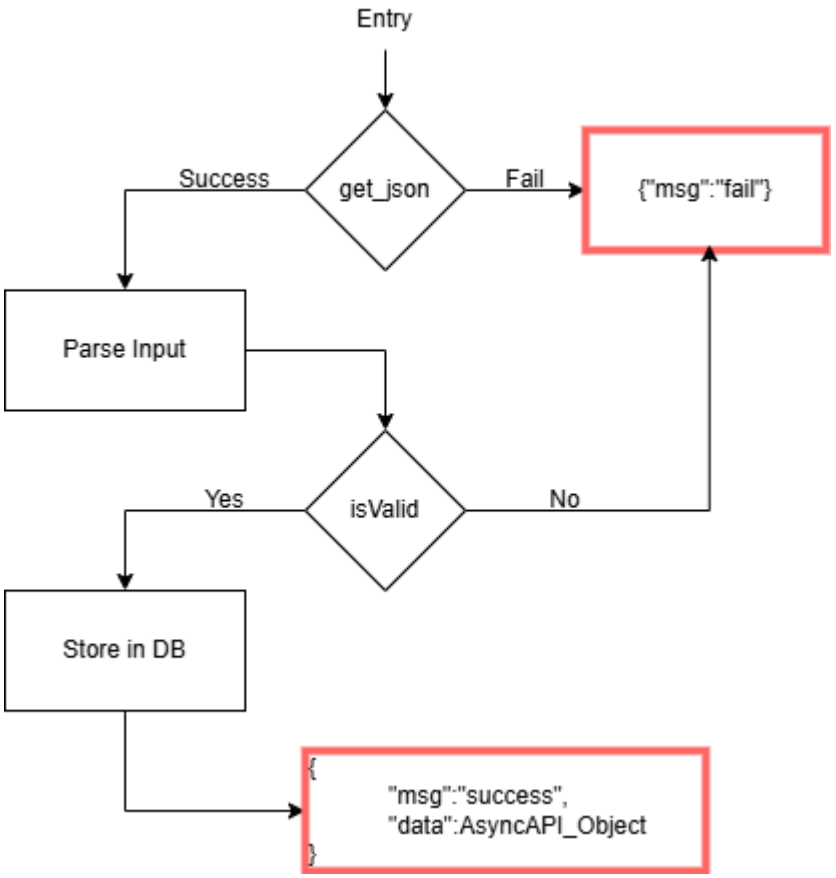


Figure40: Endpoint Functionality Diagram.

4.2.1.3 Generator Frontend

To make the process more user-friendly, we also developed a user interface (UI) where the user can create Thing Descriptions. The user has two options, either to give generator input JSON[18] for processing, or to start creating a description from scratch.

For the needs of creating the UI, React was used. This was done because it allows us to use only TypeScript/JavaScript[8] without the need for HTML and also provides the ability to better organize the code. Initially, when the user opens the application, they will be prompted to choose how to proceed, with a JSON file or from scratch. Then, if they choose a file, they will need to provide as input a JSON file that follows the same rules as the generator's input file. Once they input the file, they will be redirected to the page where they can process the data and ultimately create an AsyncAPI Thing Description (Fig.41).

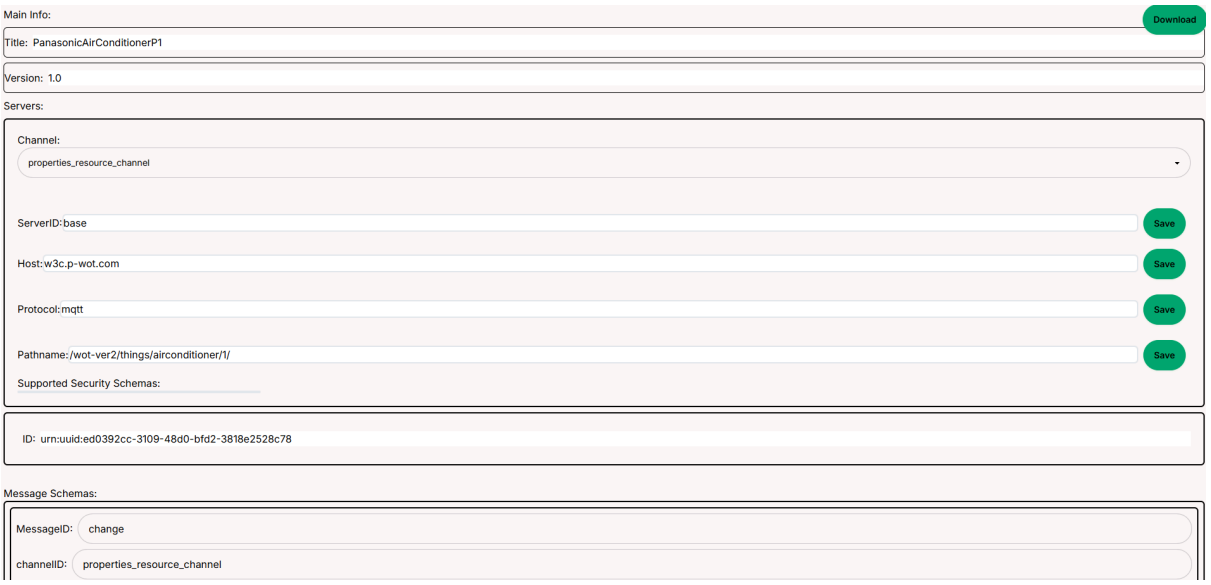


Figure41: UI: User Selecter to Edit existing File.

Next, by selecting the download option at the top right (Fig.41), the creation of a Description will be triggered, and it will be downloaded to the default directory where the browser saves downloads. Now we will see what happens if the user chooses to create a description manually.

By selecting the manually option, the user will be redirected to a page with an empty form (Fig.42) where they can input the device's data. Some fields are mandatory and are marked with a red asterisk. When the user believes they have completed the description and have filled in all the mandatory fields, they can click the "Generate Thing Description" button at

the bottom left. This will trigger the creation of an AsyncAPI description, which will be saved to the local disk in the browser's default directory.

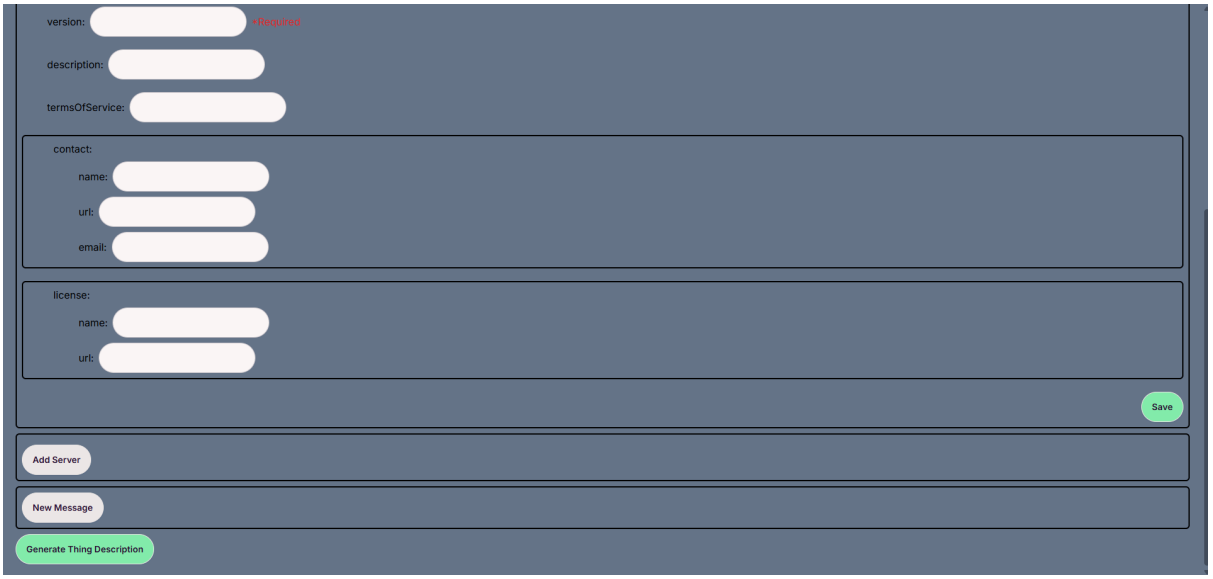


Figure42: UI: User Selected to create a TD from scratch.

In Fig.43 a diagram is presented that shows the ways the user can interact with the Generator through the User Interface.

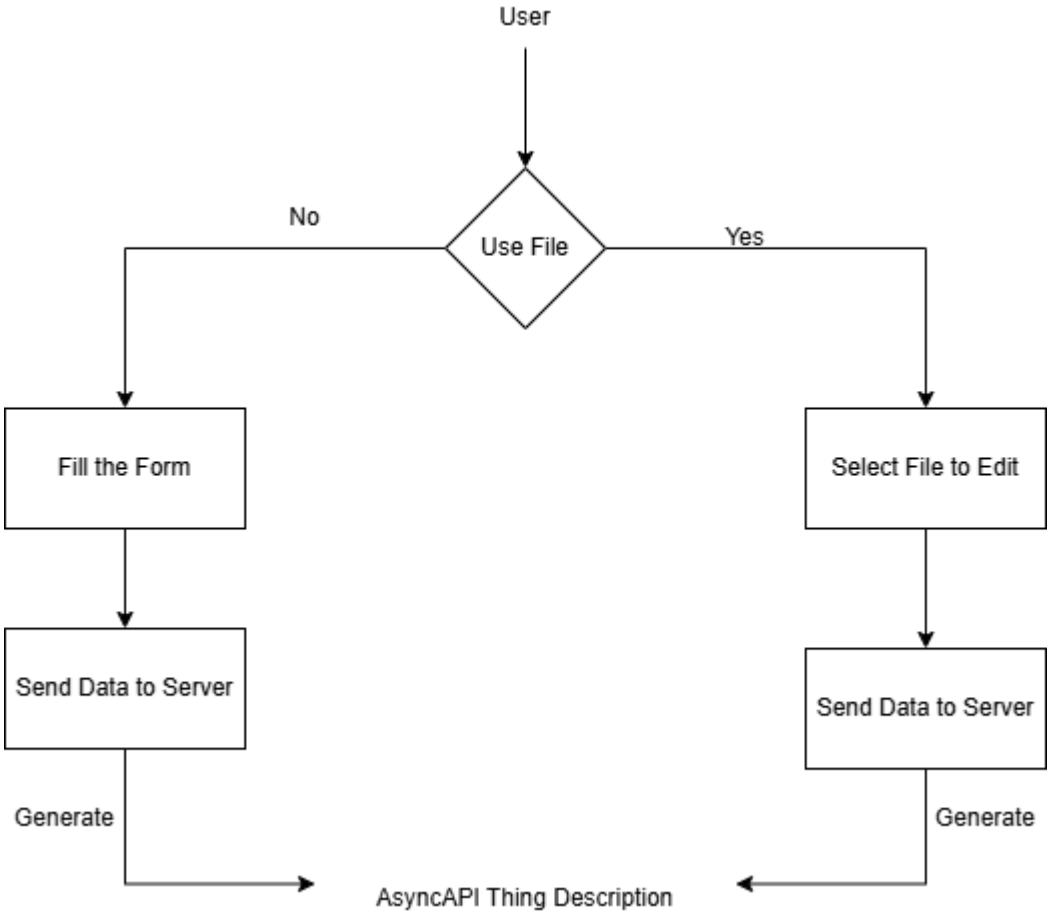


Figure43: Web App UI Use-Flow.

Finally, it should be mentioned that Docker[12] was used for the deployment of the generator and its frontend to make the process easier and more efficient.

4.2.2 Thing API Tool

For the creation of the Thing API Tool, FastAPI[7] was used for the backend and React[10] for the frontend. It allows the user to create the API of a sensor, which, with the help of Docker[12], can then operate as a service. Either a physical sensor that sends messages through http or a tool like Postman can connect to it and send messages to a KAFKA broker. This tool simultaneously generates both the description and the API.

4.2.2.1 Backend

In its current state, the tool only supports the KAFKA protocol, but it can be expanded in the future. We used an existing tool[16] on GitHub that creates a KAFKA broker, a KAFKA producer, a KAFKA consumer and a KAFDrop[22]. Through the KAFKA producer, the sensor (or a Postman-like tool) can send messages and through the KAFKA consumer we can have access to them. At this point, we need to explain the format of input that the backend receives to generate the API. This tool is used to simulate message sending of sensors. For this reason the user does not need to provide every information for the sensor, only the available messages and their schemas. Thus we can not use an already existing Thing Description but in contrast after the user provides the messages and their schemas the AsyncAPI Thing Description will be generated.

4.2.2.1.1 Input Data Schema

The input consists of a JSON with two fields: messages and protocol. The messages field is an array of objects, the structure of which we will explain below, while the protocol field is a string that specifies the protocol the device will support (for the purposes of this thesis, we assume it can only take the value KAFKA). Next, we will present the structure of the messages field. We will follow a process where we explain from the lower layers to the higher layers, declaring some objects as we have some levels of encapsulation.

So, we have the SingleMessage object(Table10), which has a mandatory field messageID that is a string and an arbitrary number of other fields defined by the user. These fields are pairs of values that include the message sent by the device and the type of the message. To make this clear, we will provide an example of a SingleMessage object: { "messageID" : "ID", "lumen":"int"}

Field	Type
messageID	string
Other Fields	int, float, string, bool, bytes, dict, List, complex combination of the above like List[dict].

Table10: SingleMessage Object.

Next, we have the MessagesPerTopic object, the structure of which is shown in the Table11 below.

Field	Type
topic	string
messages	Array <SingleMessage >

Table11: MessagesPerTopic Object.

Finally, we have the complete schema of the input, which looks like the Table12 below.

Field	Type
protocol	string
messages	Array <MessagePerTopic >

Table12: Top Level Object.

In Fig.44 we can see an example of the Input JSON. We define the protocol to be KAFKA and then we define two topics, *topic1* and *topic2*. In every topic we define a message, in this case we have *msg1* and *msg2*.

Figure44: Input Example.

4.2.2.1.2 Functionality

In this paragraph, the operation of the backend will be explained given the input. Initially, once the endpoint receives the input, analyzes it to identify the topics and the associated messages. Once it identifies them, it needs to edit the code of the tool we acquired through Github[16] (from now on will be referred as the Github Tool) and create classes based on the Pydantic library[23], which is used for data validation. These classes represent the available messages and their topics. This process is very important, and if it fails, the creation of the Thing API is not possible. When the aforementioned classes are ready, we create a new thread which, using Docker, will "spin up" the containers that constitute the Thing's API. The code can be seen in Fig.45.

```python
@router.post('/messages/create')
def CreateMessages(usrInput:m.UserInput):

    try:
        if usrInput.protocol == "KAFKA":
            CreateKafkaMessages(usrInput.messages)
            messageQueue = queue.Queue()
            dockerStart = threading.Thread (target=KAFKADockerStart, args=(messageQueue,) )
            dockerStart.start()
            dockerStart.join()
            res = messageQueue.get()
            print("Error Message:")
            print(res)
            return {
                "error":"false",
                "protocol":"KAFKA",
                "msg": "Messages created successfully",
                "url":{
                    "send":responses.kafkaURLs["send"],
                    "receive":responses.kafkaURLs["receive"]
                }
            }
```

Figure45: Endpoint Functionality.

If everything goes well, we will have two new endpoints, as shown in the return statement in Fig.45. The “send” endpoint is used to send messages to the KAFKA broker, and the “receive” endpoint provides us with an interface to access the messages being sent. In our case, we will use the Kafdrop to observe the device's communication.

4.2.2.2 Frontend

To make the process more user-friendly, a user interface was also developed from which the user can create topics and messages. Additionally, from the frontend, the user can directly obtain an AsyncAPI Thing Description of the device they have created. For the creation of the UI, the React library[10] was used, as mentioned in the previous chapters. More specifically, the user declares topics, then creates the messages that can be sent to each topic. In declaring the messages, the user essentially creates the fields that each message will contain, as well as the type of each field. When the user is ready, they click the "send" button at the bottom left, and the description is generated. Subsequently, after the submission, the backend takes over to generate the API, and the user is redirected to a page that provides the send and receive endpoints.

A use case example can be seen in Fig.46. where we define two topics: Topic1 and Topic2. In Topic1 we define the Message1 which has a field named “lumen” and is of type “int”. In Topic2 we define the Message2 with a field named “temperature” and is of type “float”.

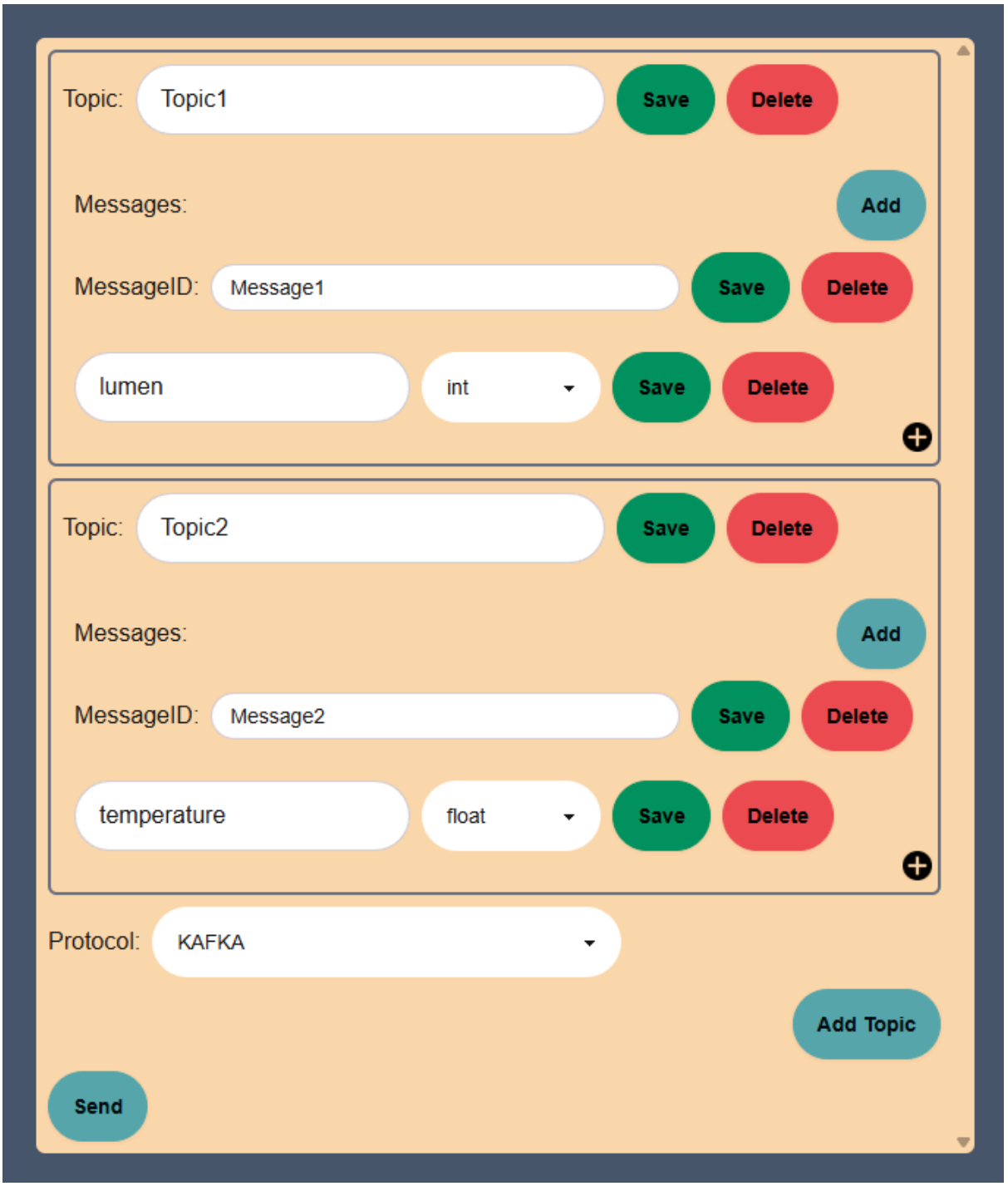


Figure46: Topic and Message Creation.

In continuation of Fig.46 when we press “send” the defined Topics, Messages and Protocol will be sent to the backend. After processing the data we provided, we will be redirected to a page (Fig.47) where we can see the endpoints for sending messages and also buttons to watch the messages that are sent (“Watch” buttons) and access to message documentation (“Docs” button).

Figure47: Redirected page.

Next, using the endpoints provided we will demonstrate a message sent. First we use Postman (Fig.48) to send a message to the KAFKA broker. Finally we can press the “Watch” button of the endpoint and watch the message that was sent with the help of Kafdrop (Fig.49)

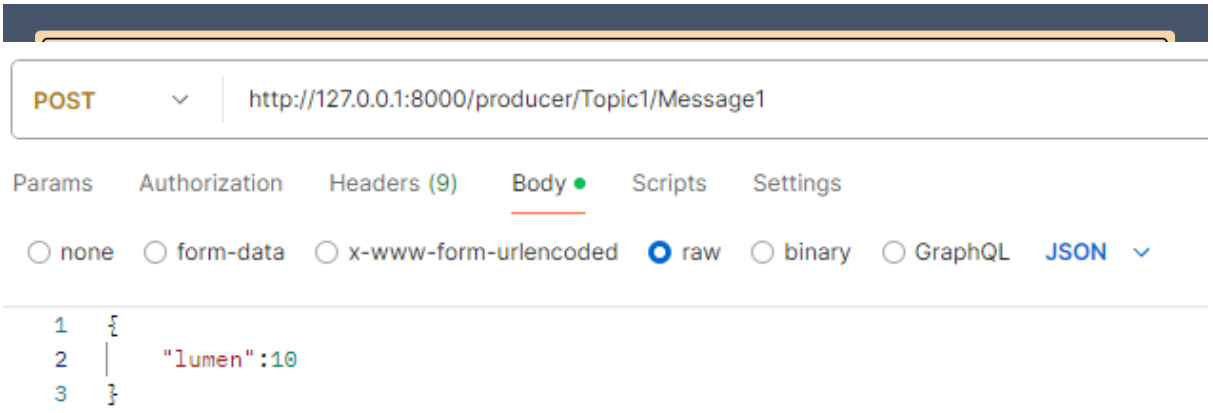


Figure48: Utilizing postman to simulate message sending.

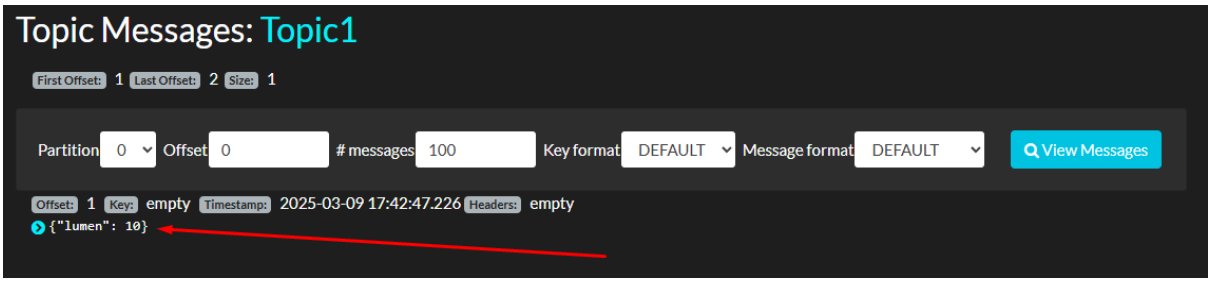


Figure49: Using the watch button to inspect the messages.

In the Fig.50 the generated AsyncAPI Thing Description can be seen. The description is generated with the logic that the sensor communicates directly with the KAFKA broker so the http interface is not part of the TD.

(a)

(b)

Figure50: Generated AsyncAPI Thing Description.

4.2.2.3 Summary

In summary, the user uses the frontend to create topics and messages, which results in the Thing Description. Subsequently, using the backend, we create the Thing's API (Fig.51), and with Docker, the deployment is carried out. Then, again from the frontend, we can observe the messages being sent (Fig.52).

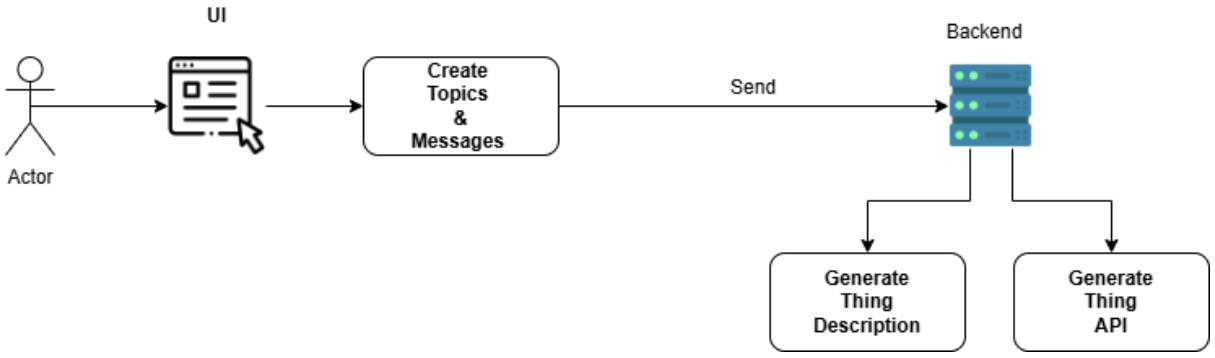


Figure51: API & Description Generation.

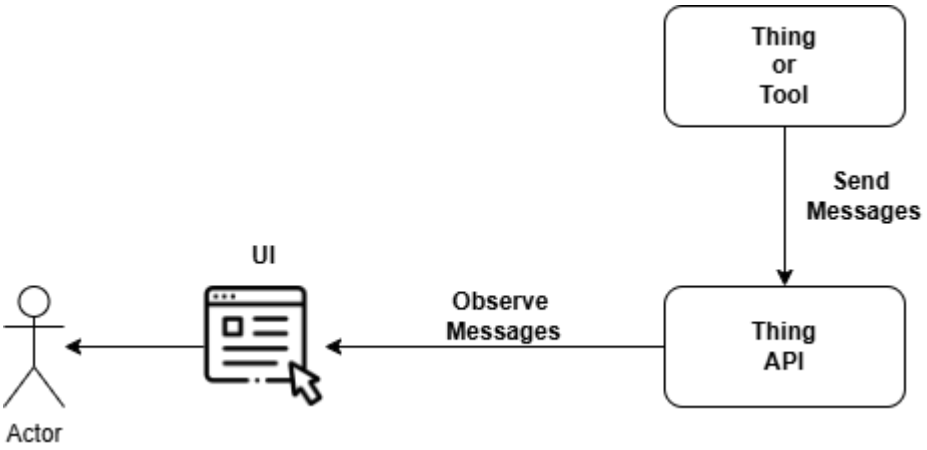


Figure52: Interaction with the Thing API.

5. Results and Discussion

In this chapter, we would like to discuss the findings of the thesis and make a comparison with existing technologies. We will discuss the contribution of the tools to the description of asynchronous Web of Things and compare the existing toolset[19] provided from the W3C for creating Thing Descriptions. We must be cautious in this comparison, as AsyncAPI has recently begun to expand into event-driven IoT and therefore does not have the years of development that the W3C standard[1] has.

5.1 Usability and Efficiency

The tools, generator and Thing API, developed within the framework of this thesis, aim to make the process of describing asynchronous WoT easier. With the generator, the user can create descriptions very easily since a UI is provided to guide them, thus reducing the need for expertise in the subject. However, through AsyncAPI[14], more experienced users can create more complex device descriptions. The W3C standard[1] served as the impetus for studying this subject. Therefore, we consider that the efficiency of the generator should be evaluated by comparing the generated descriptions with existing descriptions in the W3C TD format. Consequently, we will present some examples of descriptions available on the W3C GitHub repository and compare them with the descriptions produced by the generator.

There is no industry mechanism that produces the same results as the Thing API Tool; therefore, the comparison must be based on the findings we have generated. The Thing API Tool is a system that enables devices and applications to send messages using the KAFKA protocol while simultaneously generating the AsyncAPI Thing description. This tool has multiple applications. It can be used for simulation since, by utilizing tools like Postman, we can interact with the API. Additionally, we can connect a device (synchronous or asynchronous) and enable it to communicate via the KAFKA protocol. Since the interface operates over the HTTP protocol, even synchronous devices that utilize HTTP can be transformed into asynchronous ones that leverage the protocols offered by the Thing API.

5.2 Comparison

This section will be more technical, as it will focus on comparing descriptions between the generator and W3C[17], incorporating more images. We will aim for a one-to-one comparison to clearly demonstrate that, first, the AsyncAPI, based on the template we have designed, is not lacking in descriptive capabilities, and second, that the generator is capable of producing these descriptions.

5.2.1 Example

For our example, we will use a W3C description[1] of an LED from Fujitsu(CodeSnippet6). We will present the description available in the W3C repository[15], showcase the generator's input, and then display the corresponding description in AsyncAPI. Since the descriptions are lengthy, some parts may be truncated to maintain the structure of this document.

```
{
  "security": [
    "__auto_added_sc__"
  ],
  "id": "urn:com:fujitsu:echonet-lite:000e7b137c10029001",
  "title": "GeneralLighting-000e7b137c10029001",
  "securityDefinitions": {
    "__auto_added_sc__": {
      "scheme": "basic"
    },
    "nosec_sc": {
      "scheme": "nosec"
    }
  },
  "@context": "https://www.w3.org/2019/wot/td/v1",
  "actions": {},
  "properties": {
    "FF": {
      "forms": [
        {
          "contentType": "text/plain",
```

```
        "href": "property/FF"
      }
    ]
  },
  "SetList": {
    "readOnly": true,
    "type": "string",
    "forms": [
      {
        "contentType": "text/plain",
        "href": "property/SetList"
      }
    ]
  },
  "ProductCode": {
    "readOnly": true,
    "type": "string",
    "forms": [
      {
        "contentType": "text/plain",
        "href": "property/ProductCode"
      }
    ]
  },
  "ManufactureDate": {
    "readOnly": true,
    "type": "string",
    "forms": [
      {
        "contentType": "text/plain",
        "href": "property/ManufactureDate"
      }
    ]
  },
  "IlluminancePercentage": {
    "enums": [  "0", "1", "2",  "3", "4", "5", "6", "7",  "8", "9", "10", "11", "12", "13",
                "14",  "15",  "16", "17", "18", "19", "20", "21",  "22",  "23", "24","25",
               "26",  "27", "28",  "29", "30",  "31",   "32", "33","34", "35", "36", "37",
```

```
                "38", "39", "40",  "41", "42",  "43", "44",  "45", "46",  "47",  "48",  "49",
                 "50","51",  "52",  "53",  "54", "55", "56", "57","58",  "59", "60","61",
                       "62",  "63",  "64",  "65", "66",  "67", "68",  "69", "70", "71",  "72", "73",
           "74",  "75", "76","77","78","79", "80", "81","82","83","84","85", "86",
                 "87", "88",  "89","90","91","92", "93","94","95","96", "97","98", "99",
                "100",   "unalterable"
    ],
      "type": "string",
    "forms": [
      {
        "contentType": "text/plain",
        "href": "property/IlluminancePercentage"
      }
    ]
  },
  "CheckIPStatus": {
    "readOnly": true,
    "type": "string",
    "forms": [
      {
        "contentType": "text/plain",
        "href": "property/CheckIPStatus"
      }
    ]
  },
  "SpecVersionInfoCode": {
    "readOnly": true,
    "type": "string",
    "forms": [
      {
        "contentType": "text/plain",
        "href": "property/SpecVersionInfoCode"
      }
    ]
  },
  "GetList": {
    "readOnly": true,
    "type": "string",
```

```
    "forms": [
      {
        "contentType": "text/plain",
        "href": "property/GetList"
      }
    ]
  },
  "DeviceConnectStatus": {
    "readOnly": true,
    "type": "string",
    "forms": [
      {
        "contentType": "text/plain",
        "href": "property/DeviceConnectStatus"
      }
    ]
  },
  "MakerCode": {
    "readOnly": true,
    "type": "string",
    "forms": [
      {
        "contentType": "text/plain",
        "href": "property/MakerCode"
      }
    ]
  },
  "FaultStatus": {
    "enums": [
      "Fault",
      "Normal"
    ],
    "readOnly": true,
    "type": "string",
    "forms": [
      {
        "contentType": "text/plain",
        "href": "property/FaultStatus"
```

```
      }
    ]
  },
  "DeviceRunningStatus": {
    "readOnly": true,
    "type": "string",
    "forms": [
      {
        "contentType": "text/plain",
        "href": "property/DeviceRunningStatus"
      }
    ]
  },
  "OperationStatus": {
    "enums": [
      "ON",
      "OFF"
    ],
    "type": "string",
    "forms": [
      {
        "contentType": "text/plain",
        "href": "property/OperationStatus"
      }
    ]
  },
  "NotifyList": {
    "readOnly": true,
    "type": "string",
    "forms": [
      {
        "contentType": "text/plain",
        "href": "property/NotifyList"
      }
    ]
  },
  "PlaceOfBusinessCode": {
    "readOnly": true,
```

```
        "type": "string",
        "forms": [
          {
            "contentType": "text/plain",
            "href": "property/PlaceOfBusinessCode"
          }
        ]
      }
    },
    "events": {},
    "base": "mqtt://wot.f-ncs.ad.jp/Things/urn%3Acom%3Afujitsu%3Aechonet-lite%3A000e7b137c10029001/"
}
```

CodeSnippet6: W3C TD of Fujitsu Light.

Next, we will examine how to generate the same description in AsyncAPI. First, we will create the input for the generator (CodeSnippet7).

```
{
    "info": {
      "title": "GeneralLighting-000e7b137c10029001",
      "version": "1.0.0",
      "description": "LED light"
    },
    "servers": [
      {
        "for": "properties_resource_channel",
        "name": "base",
        "host": "wot.f-ncs.ad.jp/Things/urn%3Acom%3Afujitsu%3Aechonet-lite%3A000e7b137c10029001/",
        "protocol": "mqtt",
        "pathname": "property/",
        "security": [
          "plain"
        ]
      }
    ],
```

```
"serverSecuritySchemas": [
  {
    "type": "plain"
  }
],
"web_thing_state": [
  "FF",
  "SetList",
  "ProductCode",
  "ManufactureDate",
  "IlluminancePercentage",
  "CheckIPStatus",
  "SpecVersionInfoCode",
  "GetList",
  "DeviceConnectStatus",
  "MakerCode",
  "FaultStatus",
  "DeviceRunningStatus",
  "OperationStatus",
  "NotifyList",
  "PlaceOfBusinessCode"

],
"messagesSchemas": [
  {
    "messageID": "PlaceOfBusinessCode",
    "channelID": "properties_resource_channel",
    "headers": {},
    "payload": {
      "type": "string"
    }
  },
  {
    "messageID": "NotifyList",
    "channelID": "properties_resource_channel",
    "headers": {},
    "payload": {
      "type": "string"
```

```
    }
  },
  {
    "messageID": "OperationStatus",
    "channelID": "properties_resource_channel",
    "headers": {},
    "payload": {
      "type": "string",
      "enum":[
        "ON",
        "OFF"
      ]
    }
  },
  {
    "messageID": "DeviceRunningStatus",
    "channelID": "properties_resource_channel",
    "headers": {},
    "payload": {
      "type": "string"
    }
  },
  {
    "messageID": "FaultStatus",
    "channelID": "properties_resource_channel",
    "headers": {},
    "payload": {
      "type": "string"
    }
  },
  {
    "messageID": "MakerCode",
    "channelID": "properties_resource_channel",
    "headers": {},
    "payload": {
      "type": "string"
    }
  },
```

```
{
   "messageID": "DeviceConnectStatus",
   "channelID": "properties_resource_channel",
   "headers": {},
   "payload": {
      "type": "string"
   }
},
{
   "messageID": "GetList",
   "channelID": "properties_resource_channel",
   "headers": {},
   "payload": {
      "type": "string"
   }
},
{
   "messageID": "SpecVersionInfoCode",
   "channelID": "properties_resource_channel",
   "headers": {},
   "payload": {
      "type": "string"
   }
},
{
   "messageID": "CheckIPStatus",
   "channelID": "properties_resource_channel",
   "headers": {},
   "payload": {
      "type": "string"
   }
},
{
   "messageID": "FF",
   "channelID": "properties_resource_channel",
   "headers": {},
   "payload": {
      "type": "string"
```

```
    }
},
{
    "messageID": "SetList",
    "channelID": "properties_resource_channel",
    "headers": {},
    "payload": {
        "type": "string"
    }
},
{
    "messageID": "ProductCode",
    "channelID": "properties_resource_channel",
    "headers": {},
    "payload": {
        "type": "string"
    }
},
{
    "messageID": "ManufactureDate",
    "channelID": "properties_resource_channel",
    "headers": {},
    "payload": {
        "type": "string"
    }
},
{
    "messageID": "IlluminancePercentage",
    "channelID": "properties_resource_channel",
    "headers": {},
    "payload": {
        "type": "string",
        "enum": [ "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "10", "11", "12", "13",
                  "14", "15", "16", "17", "18", "19", "20", "21", "22", "23", "24","25",
                 "26", "27", "28", "29", "30", "31", "32", "33","34", "35", "36", "37",
              "38", "39", "40", "41", "42", "43", "44", "45", "46", "47", "48", "49",
               "50","51", "52", "53", "54", "55", "56", "57","58", "59", "60","61",
                    "62", "63", "64", "65", "66", "67", "68", "69", "70", "71", "72", "73",
```

```
                "74",  "75", "76","77","78","79", "80", "81","82","83","84","85", "86",
                    "87", "88",  "89","90","91","92", "93","94","95","96", "97","98", "99",
                   "100",   "unalterable"
        ],
        }
      }
  ]
}
```

CodeSnippet7: Generator Input.

Now we have the input of the generator (CodeSnippet7) we can use it to create the actual AsyncAPI Thing Description (CodeSnippet8)

```
{
  "asyncapi": "3.0.0",
  "info": {
    "title": "GeneralLighting-000e7b137c10029001",
    "version": "1.0.0",
    "description": "LED light"
  },
  "servers": {
    "base": {
      "host": "wot.f-ncs.ad.jp/Things/urn%3Acom%3Afujitsu%3Aechonet-lite%3A000e7b137c10029001/",
      "protocol": "mqtt",
      "pathname": "property/",
      "description": "",
      "title": "",
      "summary": "",
      "variables": {},
      "security": [
        {
          "type": "plain"
        }
```

```
      ],
      "bindings": {}
    }
  },
  "channels": {
    "properties_resource_channel": {
      "messages": {
        "FF": {
          "$ref": "#/components/messages/FF"
        },
        "SetList": {
          "$ref": "#/components/messages/SetList"
        },
        "ProductCode": {
          "$ref": "#/components/messages/ProductCode"
        },
        "ManufactureDate": {
          "$ref": "#/components/messages/ManufactureDate"
        },
        "IlluminancePercentage": {
          "$ref": "#/components/messages/IlluminancePercentage"
        },
        "CheckIPStatus": {
          "$ref": "#/components/messages/CheckIPStatus"
        },
        "SpecVersionInfoCode": {
          "$ref": "#/components/messages/SpecVersionInfoCode"
        },
        "GetList": {
          "$ref": "#/components/messages/GetList"
        },
        "DeviceConnectStatus": {
          "$ref": "#/components/messages/DeviceConnectStatus"
        },
        "MakerCode": {
          "$ref": "#/components/messages/MakerCode"
        },
        "FaultStatus": {
```

```
            "$ref": "#/components/messages/FaultStatus"
          },
          "DeviceRunningStatus": {
            "$ref": "#/components/messages/DeviceRunningStatus"
          },
          "OperationStatus": {
            "$ref": "#/components/messages/OperationStatus"
          },
          "NotifyList": {
            "$ref": "#/components/messages/NotifyList"
          },
          "PlaceOfBusinessCode": {
            "$ref": "#/components/messages/PlaceOfBusinessCode"
          }
        },
        "servers": [
          {
            "$ref": "#/servers/base"
          }
        ]
      }
    },
    "operations": {
      "properties_resource_operation": {
        "messages": [
          {
            "$ref": "#/channels/properties_resource_channel/messages/FF"
          },
          {
            "$ref": "#/channels/properties_resource_channel/messages/SetList"
          },
          {
            "$ref": "#/channels/properties_resource_channel/messages/ProductCode"
          },
          {
            "$ref": "#/channels/properties_resource_channel/messages/ManufactureDate"
          },
          {
```

```
        "$ref": "#/channels/properties_resource_channel/messages/IlluminancePercentage"
      },
      {
        "$ref": "#/channels/properties_resource_channel/messages/CheckIPStatus"
      },
      {
        "$ref": "#/channels/properties_resource_channel/messages/SpecVersionInfoCode"
      },
      {
        "$ref": "#/channels/properties_resource_channel/messages/GetList"
      },
      {
        "$ref": "#/channels/properties_resource_channel/messages/DeviceConnectStatus"
      },
      {
        "$ref": "#/channels/properties_resource_channel/messages/MakerCode"
      },
      {
        "$ref": "#/channels/properties_resource_channel/messages/FaultStatus"
      },
      {
        "$ref": "#/channels/properties_resource_channel/messages/DeviceRunningStatus"
      },
      {
        "$ref": "#/channels/properties_resource_channel/messages/OperationStatus"
      },
      {
        "$ref": "#/channels/properties_resource_channel/messages/NotifyList"
      },
      {
        "$ref": "#/channels/properties_resource_channel/messages/PlaceOfBusinessCode"
      }
    ],
    "action": "receive",
    "channel": {
      "$ref": "#/channels/properties_resource_channel"
    }
}
```

```
},
"components": {
  "messages": {
    "FF": {
      "headers": {
        "$ref": "#/components/schemas/FF_headers"
      },
      "payload": {
        "$ref": "#/components/schemas/FF_payload"
      }
    },
    "SetList": {
      "headers": {
        "$ref": "#/components/schemas/SetList_headers"
      },
      "payload": {
        "$ref": "#/components/schemas/SetList_payload"
      }
    },
    "ProductCode": {
      "headers": {
        "$ref": "#/components/schemas/ProductCode_headers"
      },
      "payload": {
        "$ref": "#/components/schemas/ProductCode_payload"
      }
    },
    "ManufactureDate": {
      "headers": {
        "$ref": "#/components/schemas/ManufactureDate_headers"
      },
      "payload": {
        "$ref": "#/components/schemas/ManufactureDate_payload"
      }
    },
    "IlluminancePercentage": {
      "headers": {
        "$ref": "#/components/schemas/IlluminancePercentage_headers"
```

```
    },
    "payload": {
      "$ref": "#/components/schemas/IlluminancePercentage_payload"
    }
  },
  "CheckIPStatus": {
    "headers": {
      "$ref": "#/components/schemas/CheckIPStatus_headers"
    },
    "payload": {
      "$ref": "#/components/schemas/CheckIPStatus_payload"
    }
  },
  "SpecVersionInfoCode": {
    "headers": {
      "$ref": "#/components/schemas/SpecVersionInfoCode_headers"
    },
    "payload": {
      "$ref": "#/components/schemas/SpecVersionInfoCode_payload"
    }
  },
  "GetList": {
    "headers": {
      "$ref": "#/components/schemas/GetList_headers"
    },
    "payload": {
      "$ref": "#/components/schemas/GetList_payload"
    }
  },
  "DeviceConnectStatus": {
    "headers": {
      "$ref": "#/components/schemas/DeviceConnectStatus_headers"
    },
    "payload": {
      "$ref": "#/components/schemas/DeviceConnectStatus_payload"
    }
  },
  "MakerCode": {
```

```
    "headers": {
      "$ref": "#/components/schemas/MakerCode_headers"
    },
    "payload": {
      "$ref": "#/components/schemas/MakerCode_payload"
    }
  },
  "FaultStatus": {
    "headers": {
      "$ref": "#/components/schemas/FaultStatus_headers"
    },
    "payload": {
      "$ref": "#/components/schemas/FaultStatus_payload"
    }
  },
  "DeviceRunningStatus": {
    "headers": {
      "$ref": "#/components/schemas/DeviceRunningStatus_headers"
    },
    "payload": {
      "$ref": "#/components/schemas/DeviceRunningStatus_payload"
    }
  },
  "OperationStatus": {
    "headers": {
      "$ref": "#/components/schemas/OperationStatus_headers"
    },
    "payload": {
      "$ref": "#/components/schemas/OperationStatus_payload"
    }
  },
  "NotifyList": {
    "headers": {
      "$ref": "#/components/schemas/NotifyList_headers"
    },
    "payload": {
      "$ref": "#/components/schemas/NotifyList_payload"
    }
```

```
    },
    "PlaceOfBusinessCode": {
      "headers": {
        "$ref": "#/components/schemas/PlaceOfBusinessCode_headers"
      },
      "payload": {
        "$ref": "#/components/schemas/PlaceOfBusinessCode_payload"
      }
    }
  },
  "schemas": {
    "PlaceOfBusinessCode_headers": {},
    "PlaceOfBusinessCode_payload": {
      "type": "string"
    },
    "NotifyList_headers": {},
    "NotifyList_payload": {
      "type": "string"
    },
    "OperationStatus_headers": {},
    "OperationStatus_payload": {
      "type": "string",
      "enum": [
        "ON",
        "OFF"
      ]
    },
    "DeviceRunningStatus_headers": {},
    "DeviceRunningStatus_payload": {
      "type": "string"
    },
    "FaultStatus_headers": {},
    "FaultStatus_payload": {
      "type": "string"
    },
    "MakerCode_headers": {},
    "MakerCode_payload": {
      "type": "string"
```

```
},
"DeviceConnectStatus_headers": {},
"DeviceConnectStatus_payload": {
   "type": "string"
},
"GetList_headers": {},
"GetList_payload": {
   "type": "string"
},
"SpecVersionInfoCode_headers": {},
"SpecVersionInfoCode_payload": {
   "type": "string"
},
"CheckIPStatus_headers": {},
"CheckIPStatus_payload": {
   "type": "string"
},
"FF_headers": {},
"FF_payload": {
   "type": "string"
},
"SetList_headers": {},
"SetList_payload": {
   "type": "string"
},
"ProductCode_headers": {},
"ProductCode_payload": {
   "type": "string"
},
"ManufactureDate_headers": {},
"ManufactureDate_payload": {
   "type": "string"
},
"IlluminancePercentage_headers": {},
"IlluminancePercentage_payload": {
   "type": "string",
   "enum": [  "0", "1", "2",  "3", "4", "5", "6", "7",  "8", "9", "10", "11", "12", "13",
                  "14",  "15",  "16", "17", "18", "19", "20", "21",  "22",  "23", "24","25",
```

```
                    "26",  "27", "28",  "29", "30",  "31",   "32", "33","34", "35", "36", "37",
                "38", "39", "40",  "41", "42",  "43", "44",  "45", "46",  "47",  "48",  "49",
                 "50","51",  "52",  "53",  "54", "55", "56", "57","58",  "59", "60","61",
                      "62",  "63",  "64",  "65", "66",  "67", "68",  "69", "70", "71",  "72", "73",
            "74",  "75", "76","77","78","79", "80", "81","82","83","84","85", "86",
                "87", "88",  "89","90","91","92", "93","94","95","96", "97","98", "99",
                "100",   "unalterable"
        ],
        }
      }
   }
}
```

CodeSnippet8: AsyncAPI Thing Description of Fujitsu Lights

As demonstrated from the above example,with the use of the generator we can create AsyncAPI TD (CodeSnippet8) of the same quality as the W3C TD (CodeSnippet6). The objective of this comparison is not to prove that the AsyncAPI TD is better than the W3C TD as this would be an arrogant statement. The objective is to demonstrate that the AsyncAPI specification[14], which is created to describe event-driven services, can describe asynchronous Things at the same level of quality as the W3C standard[1], which its core objective is to do that. In the Appendices, the reader can find additional AsyncAPI Thing Descriptions.

5.3 Implications

At this point, we should also mention some implications. While both AsyncAPI[14] and W3C[1] are widely used in the industry, the community surrounding them is small and highly specialized, meaning there are not many available resources and tools for comparison. There are some tools available for W3C[19], but for AsyncAPI, since the IoT integration is at early stages, it does not yet have a large toolset. This thesis highlights the need for the creation of more tools that are user-friendly and accessible to everyone.

6. Conclusion

In conclusion, we will provide an overview of what has been covered in this thesis and how the study conducted can contribute to the field of asynchronous device description.

6.1 Thesis Summary

In this research, we studied the possibility of creating Thing Descriptions for event-driven or asynchronous Things. To this end, we approached the issue from the perspective of the AsyncAPI standard[14], which has been specifically designed to describe asynchronous services.

We studied the standard and created a template aimed at retaining only the essential components of AsyncAPI, removing the rest to avoid unnecessary information. We concluded that we can limit the channels and operations to just three of each, ensuring compatibility with the W3C standard[1]. Therefore, we have three channels: Properties, Actions, and Web Thing, and similarly for the operations. These are sufficient to describe most devices. We kept the remaining components of AsyncAPI as they are, ensuring that our descriptions remain compatible with tools from the AsyncAPI ecosystem.

Using the template, we developed the generator, which, by providing input with the minimum required fields, can generate a description. Additionally, we created a user interface to make the process easier and more user-friendly.

Finally, the Thing API Tool was developed, where the user specifies the available messages by topic, i.e., defining the fields and their types. The tool will then generate a description (TD) and create the Thing's API. This API can be used by tools and devices to send messages based on the declared protocol (currently, KAFKA is available). Additionally, it features an interface that allows us to monitor these messages, which we utilized to track them via a UI in the browser.

6.2 Future Work

This thesis was conducted with great care and thorough study of existing standards, aiming to be as comprehensive as possible. However, within the scope of a thesis, time is limited. In this paragraph, we will outline potential improvements and future developments for the research conducted so far.

Initially, regarding the AsyncAPI standard, our research was based on the existing W3C Web of Things description technology[1] and OpenAPI TD[2]. This may have introduced a limitation in our approach, preventing us from fully utilizing the capabilities of AsyncAPI. Specifically, it would be useful in the future to explore the possibility of having more than three channels and operations, allowing for greater flexibility.

Furthermore, regarding the generator, it would be worthwhile to explore the possibility of generating descriptions based on other standards, such as OpenAPI and W3C. This would allow it to become a more comprehensive solution for users.

Finally, it is important to highlight the significance of the Thing API Tool. Currently, it only supports the KAFKA protocol. Therefore, it should be expanded to support additional protocols, enhancing its versatility and applicability.

7. References

[1] *W3C Thing Description 1.1. URL: https://www.w3.org/TR/wot-thing-description11/*

[2] *Aimilios Tzavaras, N. Mainas, F. Bouraimis, E. Petrakis "OpenAPI Thing Descriptions for the Web of Things" In: 2021, IEEE International Conference on Tools with Artificial Intelligence*

[3] *Aimilios Tzavaras, Chrisa Tsinaraki, E. Petrakis "Integrated Framework for Device and Service Descriptions in the Web of Things" In: 2024, IEEE International Conference on Tools with Artificial Intelligence*

[4] *OpenAPI Specification URL: https://swagger.io/specification/*

[5] *Python Documentation URL: https://docs.python.org/3.13/*

[6] *Flask Documentation URL: https://flask.palletsprojects.com/en/stable/*

[7] *FastAPI Documentation URL: https://fastapi.tiangolo.com/*

[8] *Typescript Documentation URL: https://www.typescriptlang.org/docs/*

[9] *NodeJS Documentation URL: https://nodejs.org/docs/latest/api/*

[10] *ReactJS Documentation URL: https://react.dev/*

[11] *NextJS Documentation URL: https://nextjs.org/docs*

[12] *Docker Documentation URL: https://docs.docker.com/*

[13] *Postman Documentation*
URL: https://learning.postman.com/docs/introduction/overview/

[14] *AsyncAPI Specification V3*
URL: https://www.asyncapi.com/docs/reference/specification/v3.0.0

[15] *W3C Thing Descriptions Repository*
URL: https://github.com/w3c/wot/tree/main/workshop/ws2/Demos/TDs

[16] *KAFKA Tool Repository URL: python-kafka-docker-main*

[17] *The World Wide Web Consortium URL: https://www.w3.org/about/*

[18] *JSON URL: https://www.json.org/json-en.html*

[19] *W3C WoT Tools URL: https://www.w3.org/WoT/developers/*

[20] *RFC3986 URL: https://datatracker.ietf.org/doc/html/rfc3986*

[21] Tailwindcss URL: https://tailwindcss.com/

[22] Kafdrop URL: https://github.com/obsidiandynamics/kafdrop

[23] Pydantic URL: https://docs.pydantic.dev/latest/

[24] Festo URL: https://www.festo.com/gr/en/

8. Appendices

8.1 AsyncAPI Objects

The tables below present the structure of the Objects according to the AsyncAPI Specification. If the reader needs more clarification shall visit the official documentation[14].

8.1.1 Contact Object

Field	Required	Type	Description
name	✗	string	Name of person or organization.
url	✗	string	URL to contact information.
email	✗	string	Email address of contact person or organization.

Table13: Contact Object

8.1.2 License Object

Field	Required	Type	Description
name	✓	string	License name of the application.
url	✗	string	License URL.

Table14: License Object

8.1.3 Tags Object

Tags object is a list of Tag objects [18].

8.1.4 Tag Object

Field	Required	Type	Description
name	✓	string	Tag name.
description	✗	string	Tag description.
externalDocs	✗	External Documentation Object | Reference Object	Additional documentation.

Table15: Tag Object

8.1.5 External Documentation Object

Field	Required	Type	Description
url	✓	string	Documentation URL.
description	✗	string	Short documentation description.

Table16: External Documentation Object

8.1.6 Reference Object

Field	Required	Type	Description
$ref	✓	string	In-Document reference string.

Table17: Reference Object

8.1.7 Server Variable Object

Field	Required	Type	Description
enum	✗	List [string]	Available values.
default	✗	string	The default value.
description	✗	string	A short description.
examples	✗	List [string]	Usage examples.

Table18: Server Variable Object

8.1.8 Security Scheme Object

The required fields are required only when used with the specified security mechanism orelse it is not a valid AsyncAPI document. If *type* matches the *when* column then shall be provided.

Field	Required	When	Type	Description
type	✓	any	string	Security mechanism type. Valid values are "userPassword", "apiKey", "X509", "symmetricEncryption", "asymmetricEncryption", "httpApiKey", "http", "oauth2", "openIdConnect", "plain", "scramSha256", "scramSha512", and "gssapi"
description	✗	any	string	Short description.
name	✓	httpApiKey	string	The name of the header, query or cookie parameter to be used.
in	✓	apiKey | httpApiKey	string	The location of the API key. Valid values are "user" and "password" for apiKey and "query", "header" or "cookie" for httpApiKey.
scheme	✓	http	string	HTTP authorization scheme.
bearerFormat	✗	http ("bearer")	string	An example of the bearer format.

flows	✓	oauth2	OAuth Flows Object	Configuration information for oauth2.
openIdConnectUrl	✓	openIdConnect	string	OpenId Connect URL to discover OAuth2 configuration values. This MUST be in the form of an absolute URL.
scopes	✗	oauth2 | openIdConnect	List [string]	List of the needed scope names. An empty array means no scopes are needed.

Table19: Security Scheme Object

8.1.9 OAuth Flows Object

Field	Required	Type	Description
implicit	✗	OAuth Flow Object	Configuration Information.
password	✗	OAuth Flow Object	Configuration Information.
clientCredentials	✗	OAuth Flow Object	Configuration Information.
authorizationCode	✗	OAuth Flow Object	Configuration Information.

Table20: OAuth Flows Object

8.1.10 OAuth Flow Object

Field	Required	Type	When	Description
authorizationUrl	✓	string	oauth2 ("implicit", "authorizationCode")	The authorization URL to be used for this flow. This MUST be in the form of an absolute URL.
tokenUrl	✗	string	oauth2 ("password", "clientCredentials", "authorizationCode")	The token URL to be used for this flow. This MUST be in the form of an absolute URL.
refreshUrl	✓	string	oauth2	The URL to be used for obtaining refresh tokens. This MUST be in the form of an absolute URL.
availableScopes	✓	Map <string, string >	oauth2	The available scopes for the OAuth2 security scheme. A map between the scope name and a short description for it.

Table21: OAuth Flows Object

8.1.11 Server Bindings Object

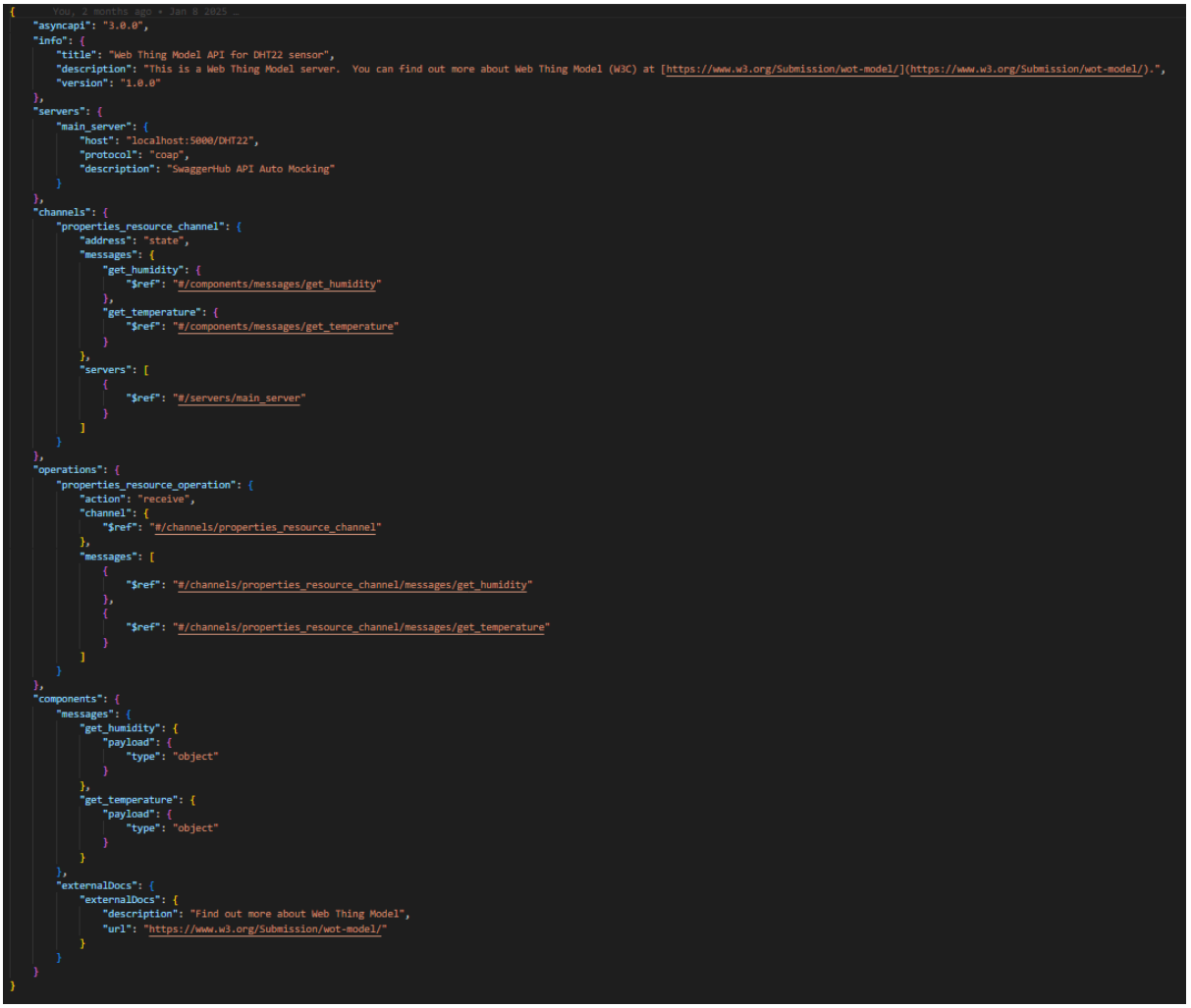
Since the definition of this object goes beyond the scope of this thesis, the reader who wishes to learn about its structure can find it at the AscynAPI Dcomentation[14].

8.2 AsyncAPI Thing Descriptions

Below, we provide some descriptions generated by the generator.

8.2.1 DHT22 Sensor AsyncAPI TD

In Fig.53 is presented the AsyncAPI Thing Description of a DHT22 sensor.



```
{
    "asyncapi": "3.0.0",
    "info": {
        "title": "Web Thing Model API for DHT22 sensor",
        "description": "This is a Web Thing Model server.  You can find out more about Web Thing Model (W3C) at [https://www.w3.org/Submission/wot-model/](https://www.w3.org/Submission/wot-model/).",
        "version": "1.0.0"
    },
    "servers": {
        "main_server": {
            "host": "localhost:5000/DHT22",
            "protocol": "coap",
            "description": "SwaggerHub API Auto Mocking"
        }
    },
    "channels": {
        "properties_resource_channel": {
            "address": "state",
            "messages": {
                "get_humidity": {
                    "$ref": "#/components/messages/get_humidity"
                },
                "get_temperature": {
                    "$ref": "#/components/messages/get_temperature"
                }
            },
            "servers": [
                {
                    "$ref": "#/servers/main_server"
                }
            ]
        }
    },
    "operations": {
        "properties_resource_operation": {
            "action": "receive",
            "channel": {
                "$ref": "#/channels/properties_resource_channel"
            },
            "messages": [
                {
                    "$ref": "#/channels/properties_resource_channel/messages/get_humidity"
                },
                {
                    "$ref": "#/channels/properties_resource_channel/messages/get_temperature"
                }
            ]
        }
    },
    "components": {
        "messages": {
            "get_humidity": {
                "payload": {
                    "type": "object"
                }
            },
            "get_temperature": {
                "payload": {
                    "type": "object"
                }
            }
        },
        "externalDocs": {
            "externalDocs": {
                "description": "Find out more about Web Thing Model",
                "url": "https://www.w3.org/Submission/wot-model/"
            }
        }
    }
}
```


Figure53

8.2.2 Festo Discrete Plant AsyncAPI TD

In Fig.54 we can see the AsyncAPI TD of Festo Discrete Plant. Discrete Plant in this context refers to automation solutions applied to manufacturing processes. For more information the reader has to search the official documentation[24].

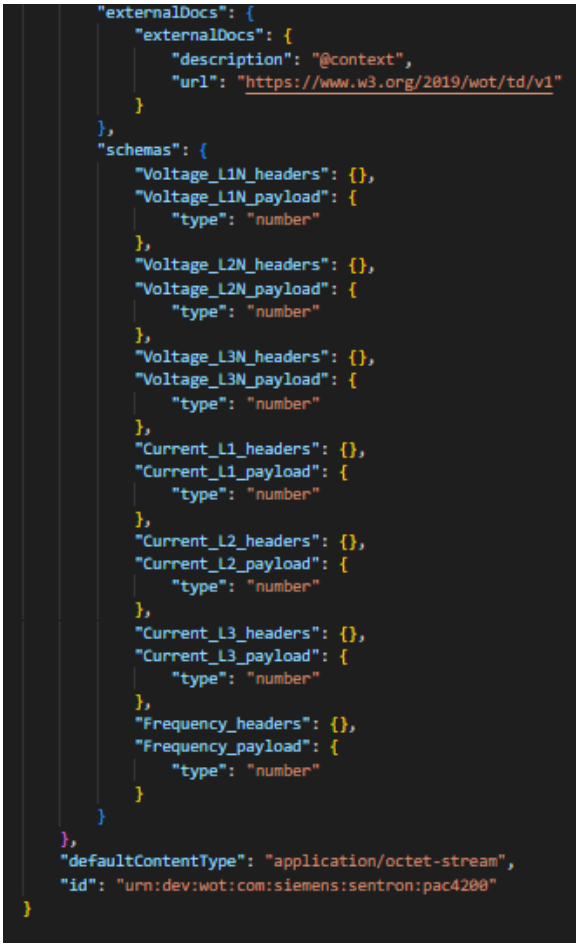
Figure54

8.2.3 Pac4200 Light AsyncAPI TD

In Fig.55 (a) and (b) is presented the TD of Pac4200 Light. In this case it is interesting that the protocol is modbus and tcp which shows the versatility of AsyncAPI capabilities.

Figure55 (a)

```
"components": {
    "messages": {
        "Voltage_L1N": {
            "headers": {
                "$ref": "#/components/schemas/Voltage_L1N_headers"
            },
            "payload": {
                "$ref": "#/components/schemas/Voltage_L1N_payload"
            }
        },
        "Voltage_L2N": {
            "headers": {
                "$ref": "#/components/schemas/Voltage_L2N_headers"
            },
            "payload": {
                "$ref": "#/components/schemas/Voltage_L2N_payload"
            }
        },
        "Voltage_L3N": {
            "headers": {
                "$ref": "#/components/schemas/Voltage_L3N_headers"
            },
            "payload": {
                "$ref": "#/components/schemas/Voltage_L3N_payload"
            }
        },
        "Current_L1": {
            "headers": {
                "$ref": "#/components/schemas/Current_L1_headers"
            },
            "payload": {
                "$ref": "#/components/schemas/Current_L1_payload"
            }
        },
        "Current_L2": {
            "headers": {
                "$ref": "#/components/schemas/Current_L2_headers"
            },
            "payload": {
                "$ref": "#/components/schemas/Current_L2_payload"
            }
        },
        "Current_L3": {
            "headers": {
                "$ref": "#/components/schemas/Current_L3_headers"
            },
            "payload": {
                "$ref": "#/components/schemas/Current_L3_payload"
            }
        },
        "Frequency": {
            "headers": {
                "$ref": "#/components/schemas/Frequency_headers"
            },
            "payload": {
                "$ref": "#/components/schemas/Frequency_payload"
            }
        }
    },
```

```
    "externalDocs": {
        "externalDocs": {
            "description": "@context",
            "url": "https://www.w3.org/2019/wot/td/v1"
        }
    },
    "schemas": {
        "Voltage_L1N_headers": {},
        "Voltage_L1N_payload": {
            "type": "number"
        },
        "Voltage_L2N_headers": {},
        "Voltage_L2N_payload": {
            "type": "number"
        },
        "Voltage_L3N_headers": {},
        "Voltage_L3N_payload": {
            "type": "number"
        },
        "Current_L1_headers": {},
        "Current_L1_payload": {
            "type": "number"
        },
        "Current_L2_headers": {},
        "Current_L2_payload": {
            "type": "number"
        },
        "Current_L3_headers": {},
        "Current_L3_payload": {
            "type": "number"
        },
        "Frequency_headers": {},
        "Frequency_payload": {
            "type": "number"
        }
    }
},
"defaultContentType": "application/octet-stream",
"id": "urn:dev:wot:com:siemens:sentron:pac4200"
}
```


Figure55 (b)

8.2.4 Smart Door AsyncAPI TD

In CodeSnippet9 the AsyncAPI TD of a Smart Door is presented.

```
{
  "asyncapi": "3.0.0",
  "info": {
    "title": "Web Thing Model API for Smart Door actuator",
    "version": "1.0.0",
    "description": "This is a Web Thing Model server."
  },
  "servers": {
    "main_server": {
      "host": "localhost:5000/DHT22",
      "protocol": "mqtt",
      "description": "SwaggerHub API Auto Mocking"
    },
    "dev_server": {
      "host": "localhost:5220/DHT22",
      "protocol": "mqtt",
      "description": "SwaggerHub API Auto Mocking"
    }
  },
  "channels": {
    "properties_resource_channel": {
      "address": "state",
      "messages": {
        "get_state": {
          "$ref": "#/components/messages/get_state"
        }
      },
      "servers": [
        {
          "$ref": "#/servers/main_server"
        }
      ]
    },
    "actions_resource_channel": {
```

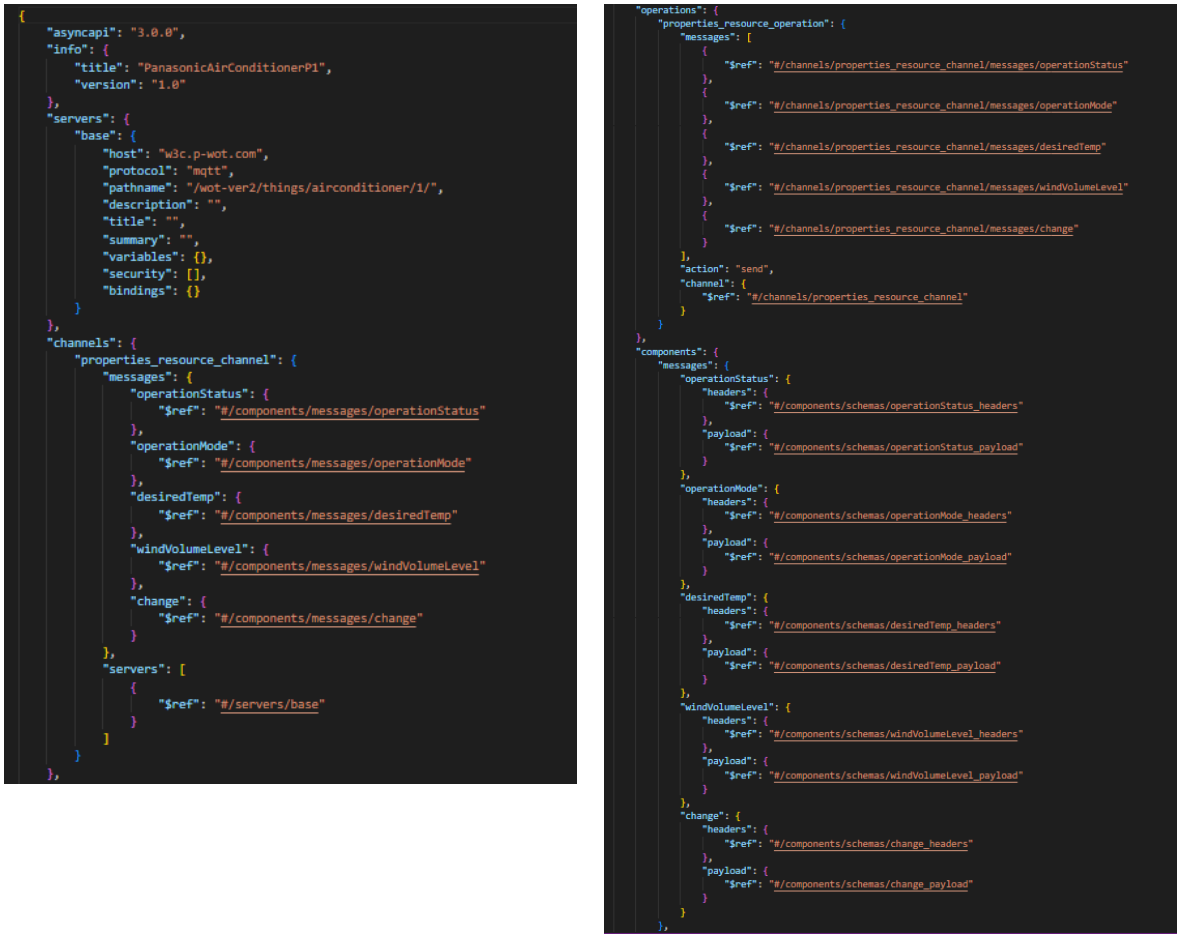
```
      "address": "actions",
      "messages": {
        "lock": {
          "$ref": "#/components/messages/lock"
        },
        "unlock": {
          "$ref": "#/components/messages/unlock"
        }
      },
      "servers": [
        {
          "$ref": "#/servers/main_server"
        }
      ]
    }
  },
  "operations": {
    "properties_resource_operation": {
      "action": "receive",
      "channel": {
        "$ref": "#/channels/properties_resource_channel"
      },
      "messages": [
        {
          "$ref": "#/channels/properties_resource_channel/messages/get_state"
        }
      ]
    },
    "actions_resource_operation": {
      "action": "send",
      "channel": {
        "$ref": "#/channels/actions_resource_channel"
      },
      "messages": [
        {
          "$ref": "#/channels/actions_resource_channel/messages/lock"
        },
        {
```

```
                "$ref": "#/channels/actions_resource_channel/messages/unlock"
              }
            ]
          }
        },
        "components": {
          "messages": {
            "get_state": {
              "payload": {
                "type": "object"
              }
            },
            "lock": {
              "payload": {
                "type": "object"
              }
            },
            "unlock": {
              "payload": {
                "type": "object"
              }
            }
          },
          "externalDocs": {
            "externalDocs": {
              "description": "Find out more about Web Thing Model",
              "url": "https://www.w3.org/Submission/wot-model/"
            }
          }
        }
      }
```

CodeSnippet9

8.2.5 Panasonic Air Conditioner AsyncAPI TD

In Fig.56 (a) and (b) we can see the TD of a Panasonic Air Conditioner. The importance of this description is that this is a real life example of an asynchronous device.



```
{
    "asyncapi": "3.0.0",
    "info": {
        "title": "PanasonicAirConditionerP1",
        "version": "1.0"
    },
    "servers": {
        "base": {
            "host": "w3c.p-wot.com",
            "protocol": "mqtt",
            "pathname": "/wot-ver2/things/airconditioner/1/",
            "description": "",
            "title": "",
            "summary": "",
            "variables": {},
            "security": [],
            "bindings": {}
        }
    },
    "channels": {
        "properties_resource_channel": {
            "messages": {
                "operationStatus": {
                    "$ref": "#/components/messages/operationStatus"
                },
                "operationMode": {
                    "$ref": "#/components/messages/operationMode"
                },
                "desiredTemp": {
                    "$ref": "#/components/messages/desiredTemp"
                },
                "windVolumeLevel": {
                    "$ref": "#/components/messages/windVolumeLevel"
                },
                "change": {
                    "$ref": "#/components/messages/change"
                }
            },
            "servers": [
                {
                    "$ref": "#/servers/base"
                }
            ]
        }
    },
```

```
    "operations": {
        "properties_resource_operation": {
            "messages": [
                {
                    "$ref": "#/channels/properties_resource_channel/messages/operationStatus"
                },
                {
                    "$ref": "#/channels/properties_resource_channel/messages/operationMode"
                },
                {
                    "$ref": "#/channels/properties_resource_channel/messages/desiredTemp"
                },
                {
                    "$ref": "#/channels/properties_resource_channel/messages/windVolumeLevel"
                },
                {
                    "$ref": "#/channels/properties_resource_channel/messages/change"
                }
            ],
            "action": "send",
            "channel": {
                "$ref": "#/channels/properties_resource_channel"
            }
        }
    },
    "components": {
        "messages": {
            "operationStatus": {
                "headers": {
                    "$ref": "#/components/schemas/operationStatus_headers"
                },
                "payload": {
                    "$ref": "#/components/schemas/operationStatus_payload"
                }
            },
            "operationMode": {
                "headers": {
                    "$ref": "#/components/schemas/operationMode_headers"
                },
                "payload": {
                    "$ref": "#/components/schemas/operationMode_payload"
                }
            },
            "desiredTemp": {
                "headers": {
                    "$ref": "#/components/schemas/desiredTemp_headers"
                },
                "payload": {
                    "$ref": "#/components/schemas/desiredTemp_payload"
                }
            },
            "windVolumeLevel": {
                "headers": {
                    "$ref": "#/components/schemas/windVolumeLevel_headers"
                },
                "payload": {
                    "$ref": "#/components/schemas/windVolumeLevel_payload"
                }
            },
            "change": {
                "headers": {
                    "$ref": "#/components/schemas/change_headers"
                },
                "payload": {
                    "$ref": "#/components/schemas/change_payload"
                }
            }
        },
```

Figure56 (a)


"schemas": {
"change_headers": {
"type": "object",
"properties": {
"contentType": {
"type": "string",
"const": "application/json"
}
}
},
"change_payload": {
"type": "object",
"properties": {
"operationStatus": {
"type": "boolean"
},
"operationMode": {
"type": "string",
"enum": [
"Auto",
"Cooling",
"Heating",
"Dehumidifying",
"Blast"
]
},
"desiredTemp": {
"type": "number",
"minimum": 16,
"maximum": 30
},
"windVolumeLevel": {
"type": "number",
"minimum": 0,
"maximum": 8
}
}
},
"windVolumeLevel_headers": {
"type": "object",
"properties": {
"contentType": {
"type": "string",
"const": "application/json"
}
}
},
"windVolumeLevel_payload": {
"type": "number",
"minimum": 0,
"maximum": 8
},


"desiredTemp_headers": {
"type": "object",
"properties": {
"contentType": {
"type": "string",
"const": "application/json"
}
}
},
"desiredTemp_payload": {
"type": "number",
"minimum": 16,
"maximum": 30
},
"operationMode_headers": {
"type": "object",
"properties": {
"contentType": {
"type": "string",
"const": "application/json"
}
}
},
"operationMode_payload": {
"type": "string",
"enum": [
"Auto",
"Cooling",
"Heating",
"Dehumidifying",
"Blast"
]
},
"operationStatus_headers": {
"type": "object",
"properties": {
"contentType": {
"type": "string",
"const": "application/json"
}
}
},
"operationStatus_payload": {
"type": "boolean"
}
}
},
"id": "urn:uuid:ed0392cc-3109-48d0-bfd2-3818e2528c78"
}

Figure56 (b)