



**Technical Report / Student Internship**

**Fortiss, Industrial IoT**

Jorge de Lima Tostes – Intern

Nisrine Bnouhanna – First Supervisor

Prof. Dr. Rute C. Sofia – Coordinator and Second Supervisor

Munich – Germany

## Acknowledgements

The work developed in this internship relates with a fortiss open-source demonstrator, TSMatch. The work has been supervised by Nisrine Bnouhanna, and by Prof. Dr. Rute C. Sofia.

The work has also been supported by the student Erkan Karabulut, who has been developing a component of TSMatch, the TSMatch client.

## Executive Summary

This document corresponds to the proposed DetNetWiFi dissemination and publication plan. The planning shall be regularly revised and updated via reports due on M12 (D1.2) and M20 (D1.3). The aim of this report is to assist in a better and coordinated dissemination of the scientific and technological results of DetNetWiFi.

## Contents

1.	Introduction .....	5
1.1.	Focus .....	5
1.2.	Proposed Activities and Schedule .....	5
1.3.	Internship's Goals .....	7
1.4.	Expected Outcome.....	7
2.	Background Work.....	8
2.1.	TSMatch Concept.....	8
2.2.	TSMatch Demonstrator.....	8
2.3.	Service Descriptions and Current Ontology in TSMatch.....	9
3.	Architecture and Implementation Aspects.....	12
3.1.	Service Description Models.....	14
3.2.	Parser Component .....	15
3.2.1.	Detailed Code Description .....	18
3.2.2.	How to Run .....	19
3.2.3.	Limitations.....	20
3.3.	External Service Interface (ESI) .....	21
3.3.1.	Detailed Code Description .....	23
3.3.2.	How to Run .....	24
3.3.3.	Dealing with changes to the TSMatch Structure and to the Taxonomy .....	25
4.	Summary and Next Steps.....	27
5.	References .....	28

## List of Tables

Table 1: parser software structure. ....	16
Table 2: External Service Interface code structure. ....	21

## List of Figures

Figure 1: Internship Gantt chart. ....	7
Figure 2: TSMATCH demonstrator. ....	8
Figure 3: Early version of the Ontology for TSMATCH inputs. ....	9
Figure 4: Code Example 1 – WSDL example. ....	11
Figure 5: WSDL’s structure. ....	11
Figure 6: Code Example 2 – WSDL structure following the ontology developed. ....	12
Figure 7: Architecture Diagram for the demonstrator’s system. The “API Connector” corresponds to the “External Service Interface” . ....	13
Figure 8: Communication Diagram involving the developed components. ....	14
Figure 9: TSMATCH client developed for Android. ....	19
Figure 10: Communication Diagram for the API Connector. ....	22

## 1. Introduction

This report is an official and technical documentation of the work developed during the six months of the internship ("Pflichtpraktikum") developed as part of the student's master program for EIGSI – La Rochelle in fortiss GmbH.

The report covers the planned activities, design choices, implementation aspects and provides information about constraints found during the project that led to the addition of new tasks.

### 1.1. Focus

The work developed served as a mandatory internship required by EIGSI as a conclusion project for the 10<sup>th</sup> semester of studies, necessary for acquiring the diploma of master's in engineering.

This work was focused on providing technical support to work being developed in the context of "Industrial IoT", regarding supporting decentralised data exchange. Specifically, the work shall assist the development of tasks in the context of the "Industrial IoT Lab" project of fortiss, namely, in tasks concerning the demonstrator "Decentralised data exchange Demonstrator", aka TSMatch. The work developed was also be integrated into a demonstration in the context of the H2020 EFPP project. The specific goal of the work was to support the automated parsing of semantic descriptions of IoT things, to allow a better matching to semantic descriptions of IoT services.

### 1.2. Proposed Activities and Schedule

Three main activities and a study phase were included as the initial goal of the work plan for this internship; moreover, a fourth activity was included during the project's development. These activities were: creation of service description models in the W3C recommended description languages; development of a software-based component that parses description files, validates their contents, and serializes requests to TSMatch; evaluation of the work developed by presenting a technical report, and provide a demonstration and a presentation. An extra task has been added, namely, the creation of an API connector that receives external service's requirements via a REST interface

The Gantt Chart is presented in Figure 1. Marked cells indicate the weeks in which they were developed.



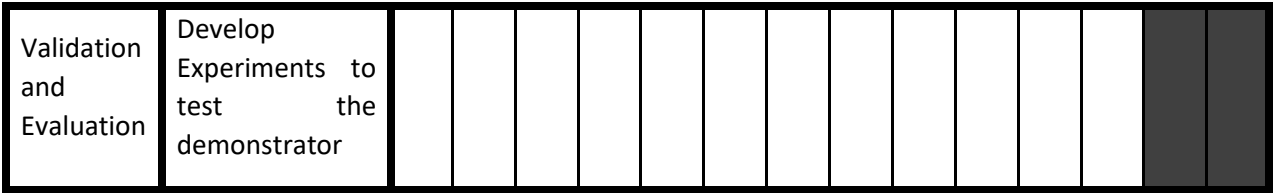


Figure 1: Internship Gantt chart.

### 1.3. Internship's Goals

The proposed goals were as follows:

- Goal 1: to develop a parser capable of supporting an improved and automated data matching between IoT Things data (attributes) and IoT services.
- Goal 2: to integrate in the parser the capability to handle a large, extended ontology.
- Goal 3: to create an interface (REST) to allow the support of external services on the demonstrator (TSMatch).
- Goal 4: to validate the developed software, based upon provided use-cases.

### 1.4. Expected Outcome

- Outcome 1, Description Language Models, available via GitLab.
- Outcome 2, parser, available via GitLab.
- Outcome 3, External service interface available via GitLab.
- Outcome 4, improved TSMatch demonstrator and demonstration.

## 2. Background Work

This section provides information about the overall TSMatch concept and the current TSMatch demonstrator, as well as background work that has been used for the development of the code during the internship.

### 2.1. TSMatch Concept

TSMatch is a software-based framework that, based on IoT Things descriptions, performs similarity matchmaking to service descriptions, to better support data aggregation. Further information of TSMatch is available via [11].

### 2.2. TSMatch Demonstrator

The TSMatch demonstrator is one of the demonstrators of the fortiss IIoT and is illustrated in Figure 2. Its server-side component, the TSMatch Engine, is installed on the fortiss IIoT gateway, an Intel NUC device that runs also other services on the Lab: Mosquitto (MQTT broker); PostGRES SQL daemon (database), etc. The client-side of TSMatch currently runs on Android. The demonstrator includes multiple sensors interconnected to 2 Raspberry Pis.

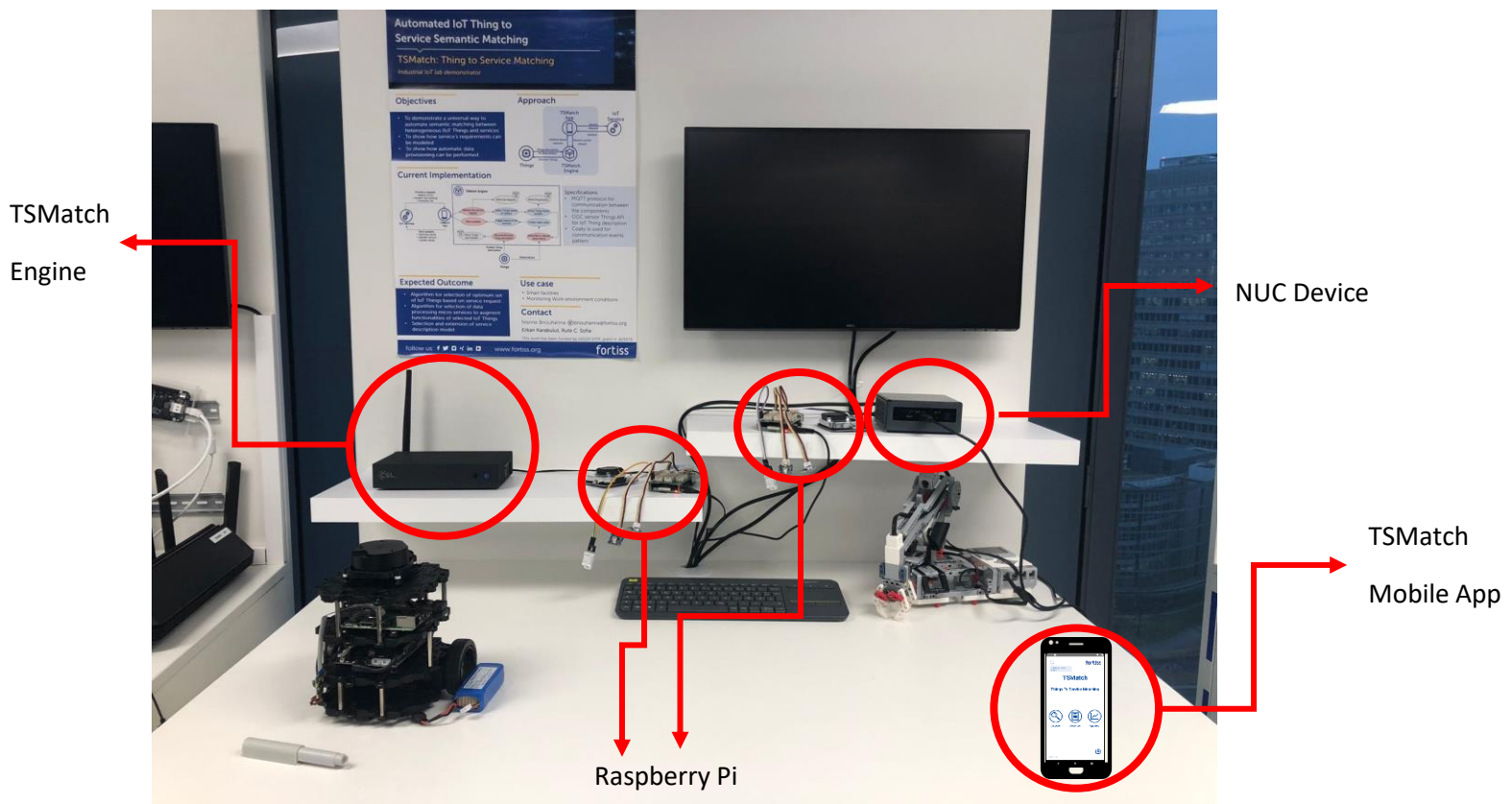


Figure 2: TSMatch demonstrator.



### 2.3. Service Descriptions and Current Ontology in TSMatch

In TSMatch, services are modelled semantically. Currently, the parser software that has been developed in this internship relies on two examples of services that have been described by recurring to WSDL 2.0 and OWL-S, both recommended by the W3C [5][9].

An aspect to consider was the capability to add new attributes (new IoT Things). For that, one can recur to different ontologies which are often developed per domain [10]. To simplify our work, we have recurred to a multiple-domain ontology currently under work in fortiss, IIoT. The ontology is being developed based on an extensive multi-domain taxonomy and follows a tree structure. An asset has been created to model this proposed [cross-domain ontology](#). Currently, the lower branches include the following attributes: Domain, Monitoring Aspect and Measurement Type; they are used to classify different properties from sensors. And the top branches are the words that can be used describe each property (such as Temperature, Humidity, Velocity and Occupancy). An early version of the cross-domain ontology is shown in Figure 3.

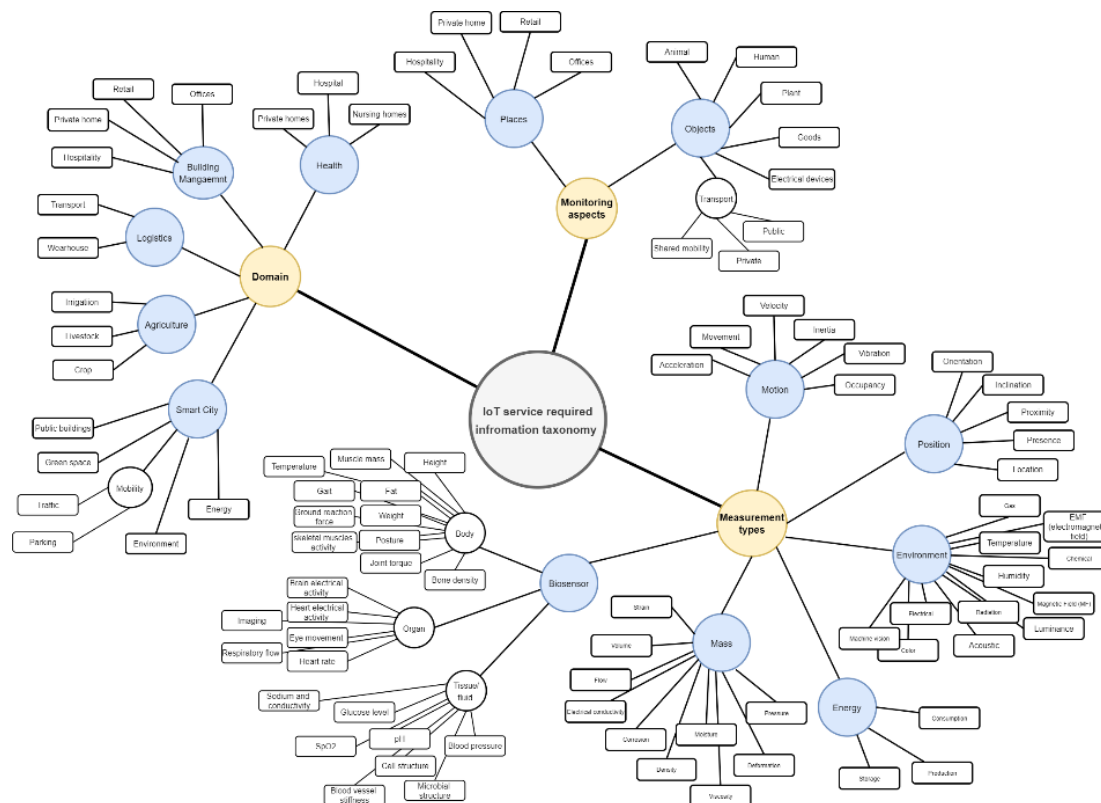


Figure 3: Early version of the Ontology for TSMatch inputs.

To understand the development process of the description models and the parser it is necessary to explain the structure of WSDL and OWL-S. The first one is an XML-based structure with a divided in four elements: types, interface, binding and service, these elements can contain parameters, such as “**name**”, “**type**”, “**element**”, and “**ref**”, which help define them and locate other components that are linked to them in the file, and are inside a parent element called “**description**” [6][7].

Types contains all the data type definitions and element declarations of the service's parameters, thus the necessary information about the inputs are contained in this tag. Interface contains the operation's inputs and outputs; they have names and references as parameters pointing to elements in the **"types"** tag that contain the description of these parameters. Therefore, the first step of the parser software is to locate all inputs from the service's **"Interface"** tag and get their definitions in the pointed elements inside the **"types"**. The latter two elements, **"binding"** and **"service"**, describe the communication protocols and structure as well as the location of the endpoint that serves that service, therefore they don't contain relevant information for the parser software and therefore have no impact in the description models for TSMATCH. An example of each of those elements is shown in bold inside the WSDL code example 1 illustrated in Figure 4 and extracted from [6].

```

<types>
  <xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://greath.example.com/2004/schemas/resSvc"
    xmlns="http://greath.example.com/2004/schemas/resSvc">
    <xs:element name="checkAvailability" type="tCheckAvailability"/>
    <xs:complexType name="tCheckAvailability">
      <xs:sequence>
        <xs:element name="checkInDate" type="xs:date"/>
        <xs:element name="checkOutDate" type="xs:date"/>
        <xs:element name="roomType" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
    <xs:element name="checkAvailabilityResponse" type="xs:double"/>
    <xs:element name="invalidDataError" type="xs:string"/>
  </xs:schema>
</types>

<interface name = "reservationInterface" >
  <fault name = "invalidDataFault"
    element = "ghns:invalidDataError"/>
  <operation name="opCheckAvailability"
    pattern="http://www.w3.org/ns/wsdli/in-out"
    style="http://www.w3.org/ns/wsdli/style/iri"
    wsdlx:safe = "true">
    <input messageLabel="In"
      element="ghns:checkAvailability" />
    <output messageLabel="Out"
      element="ghns:checkAvailabilityResponse" />
    <outfault ref="tns:invalidDataFault" messageLabel="Out"/>
  </operation>
</interface>

<binding name="reservationSOAPBinding"
  interface="tns:reservationInterface"
  type="http://www.w3.org/ns/wsdli/soap"

```

```

    wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP/">
    <fault ref="tns:invalidDataFault"
    wsoap:code="soap:Sender"/>
    <operation ref="tns:opCheckAvailability"
    wsoap:mep="http://www.w3.org/2003/05/soap/mep/soap-response"/>
  </binding>
  <service name="reservationService"
    interface="tns:reservationInterface">
    <endpoint name="reservationEndpoint"
      binding="tns:reservationSOAPBinding"
      address="http://greath.example.com/2004/reservation"/>
  </service>

```

Figure 4: Code Example 1 – WSDL example.

The structure mentioned is also depicted in Figure 5.

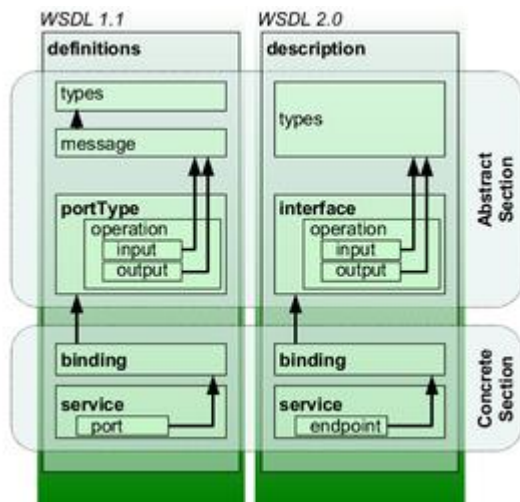


Figure 5: WSDL's structure.

The second description language, OWL-S, follows a similar pattern with changes regarding only how the software gets the inputs. Unlike WSDL, where the **"types"** element contains data types that define the parameters of the service, in OWL-S each parameter is an object that has all its properties defined inside them. Therefore, after searching for the inputs inside the atomic processes of the file, the parser afterwards looks for those inputs as individual types themselves, that contain the properties of interest for TSMATCH. The description file models that were created have the same structure of a standard description file, as mentioned above, but its inputs inside the **"types"** element follow the taxonomy. Those files were created for testing purposes and to give external services a better understanding of the requirements from the TSMATCH software and how they can build a valid description file that can be parsed by the demonstrator.

Two of the taxonomy models created were based on an Environment Monitoring Service that gives information about the quality of the working environment based on the temperature, humidity, and occupancy of a target room. That hypothetical service needs to send three requirements to TSMATCH, one for each of the measurement types listed above (temperature, humidity, and occupancy).

Therefore, the **“types”** and **“interface”** elements follow the structure shown in code example 2:

```
<types>
  <xsd:import namespace="URL_for_Schema" schemaLocation="taxonomy_2.xsd"/>
  <xsd:schema          elementFormDefault="qualified"          attributeFormDefault="qualified"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" targetNamespace="URL_for_Schema">
    <!--Start of WSDL Elements -->
    <xsd:element name="TemperatureInput">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element ref="Taxonomy">
            <xsd:element ref="Domain">
              <xsd:element ref="Building-Management">
                <xsd:element name="Offices" type="xsd:string"/>
              </xsd:element>
            </xsd:element>
          <xsd:element ref="Monitoring-Aspects">
            <xsd:element ref="Places">
              <xsd:element name="Offices" type="xsd:string"/>
            </xsd:element>
          </xsd:element>
          <xsd:element ref="Measurement-Types">
            <xsd:element ref="Environment">
              <xsd:element name="Temperature" type="xsd:string"/>
            </xsd:element>
          </xsd:element>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:schema>
</types>

<interface name="SouthboundInterface">
  <operation          name="Updates"          pattern="http://www.w3.org/ns/wsdli/in-only"
style="http://www.w3.org/ns/wsdli/style/iri">
    <input element="ghns:TemperatureInput"/>
    <input element="ghns:HumidityInput"/>
    <input element="ghns:OccupancyInput"/>
  </operation>
</interface>
```

Figure 6: Code Example 2 – WSDL structure following the ontology developed.

### 3. Architecture and Implementation Aspects

In this section, all the code details, structure of implementation and guidelines on how and where to change the software are presented. Therefore, it contains simple explanations and communication

diagrams on the components developed. Figure 7 presents the architecture diagram that provides an overview of the system, and the developed components are:

- Examples for services, based on specific descriptive language, section 4.1.
- Parser service, currently placed on the TSMatch client as illustrated, section 4.2.
- External Service Interface, described as “API Connector” in Figure 2, section 4.3.

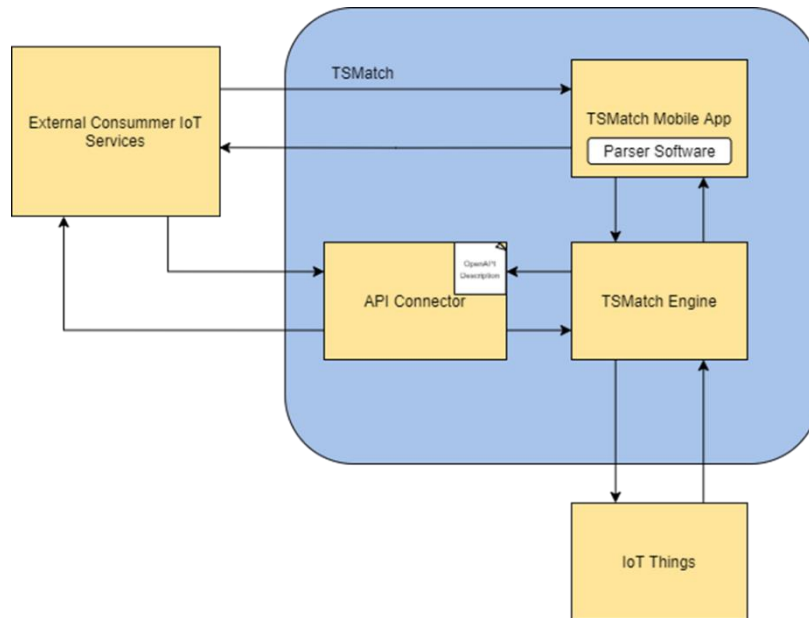


Figure 7: Architecture Diagram for the demonstrator's system. The “API Connector” corresponds to the “External Service Interface”.

Figure 8 illustrates the communication sequence between an external service and an external service orchestration entity; the TSMatch client (TSMatch Mobile App); The TSMatch Engine; the Parser Software.

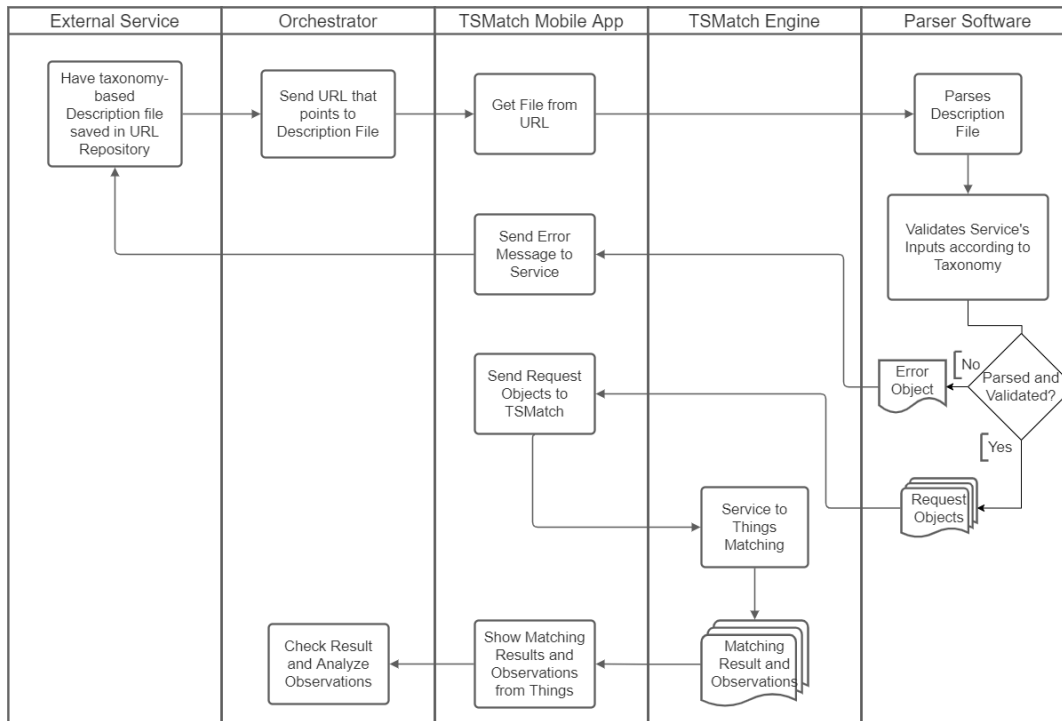


Figure 8: Communication Diagram involving the developed components.

### 3.1. Service Description Models

Since there are two description languages that this project focus on: WSDL and OWL-S, the first task of this project was to create descriptive files and schemas to validate the contents of those services in each of the sub dependencies of those languages, namely XML, RDF, XSD and OWL. These files would have a basic structure and would model a service for the description models that the demonstrator can understand.

In the XSD, there are complex type definitions for each of the branches of the taxonomy tree. And in the OWL file there are three object attributes to classify the main branches of the inputs: “**hasDomain**” for domain, “**hasMeasurementType**” for measurement type, and “**hasMonitoringAspect**” for monitoring aspect, which are necessary for the parser. On the latter case, those attributes were created to avoid the need of three inputs. It is possible, however, to understand the input objects also through the description models created in the internship.

The Service description models were built according to the ontology currently being used in TSMatch and briefly described in section 3.3. Therefore, changes to the taxonomy structure would imply in changes to the Description models and their schemas. The schemas, be it the type definitions for WSDL and OWL-S or the OpenAPI Schema, are the definition of the taxonomy’s data type, in other words, they describe the values that can be contained in the taxonomy. Any change on the taxonomy would imply in equivalent changes to the schemas. An example is the OpenAPI schema definition for “Measurement Type” shown in code example 6.

```

1. measurementType:
2.     type: string
  
```

```

3.         enum:
4.             - motion
5.             - position
6.             - environment
7.             - energy
8.             - mass
9.             - biosensor

```

#### Code Example 6 – Schema definition for the OpenAPI Specification.

If any new definitions were to be added to “Measurement Type”, they would have to be added to the Schema above. In addition, if the element name “Measurement Type” is changed, all values that contain that name in the Schema would require change. Another example of how the taxonomy is used is depicted in code example 7, in one of the WSDL description models’ “types” definition, it can be seen inside “ref” and “name” parameters.

```

1. <xsd:element ref="Taxonomy">
2. <xsd:element ref="Domain">
3.     <xsd:element ref="Building-Management">
4.         <xsd:element name="Offices" type="xsd:string"/>
5.     </xsd:element>
6. </xsd:element>
7. <xsd:element ref="Monitoring-Aspects">
8.     <xsd:element ref="Places">
9.         <xsd:element name="Offices" type="xsd:string"/>
10.    </xsd:element>
11. </xsd:element>
12. <xsd:element ref="Measurement-Types">
13.     <xsd:element ref="Motion">
14.         <xsd:element name="Occupancy" type="xsd:string"/>
15.     </xsd:element>
16. </xsd:element>
17. </xsd:element>

```

#### Code Example 7 – Use of Taxonomy inside WSDL file.

### 3.2. Parser Component

The parser had as requirement to be built on JavaScript to be portable, and independent of platform. It should parse the service description file from consumer services, extract the requirements from those services, and check if they are aligned with the active ontology(ies), and then sends that information in TSMATCH’s message structure, in other words, the built-in communication structure for messages through the MQTT broker. Please refer to the [parser code in gitlab](#). To develop the parser, we have resorted to NodeJS<sup>1</sup>, as it allows for Javascript to be run outside of its environment, e.g., browser. The development has been done in Windows, Visual Studio IDE. Currently, the parser component is a module that runs on the TSMATCH client; however, it can be run anywhere else.

The TSMATCH client currently creates (or obtains) a service description as mentioned. This semantic service description is parsed and validated based on the ontology defined. For that purpose, a JavaScript object

---

<sup>1</sup> <https://nodejs.org/en/>

created from the XML file that contains the entire ontology structure was included as an asset inside the software's structure. Further details can be found in gitlab, on the [ontology asset code](#), currently under the TSMatch client.

The Parser component has been placed currently in the TSMatch client (currently, developed as an Android App) and has two main files linked to it: "[ServiceRequestParserConfig.js](#)" and "[ServiceRequestParser.js](#)". Inside the first file there is the ontology object that is received through an XML file or by filling the object present inside this code. If the taxonomy object inside the "ServiceRequestParserConfig.js" is used, all changes to the taxonomy must be also included to the taxonomy object. The latter file contains the method that parses description files and validates the inputs according to the taxonomy.

Table 1 shows the file structure for the demonstrator containing the Parser Software. All files used by the Parser software are highlighted in bold.

Table 1: parser software structure.

TSMatchMobileApp.	
├─ broker	
├─ graphdb script	
├─ robot	
├─ TSMatch_API	
├─ TSMatch_Engine	
└─ TSMatchMobileApp	
├─ index.js	- entry file of the app with basic config
├─ src	
├─ action	- react redux action definitions
├─ asset	
├─ component	
├─ config	
├─ app.js	- initial react native component
├─ navigation.js	- react-navigation config
├─ <b>ServiceRequestParserConfig.js</b>	- <b>configuration of the service request parser</b>
├─ store.js	- react redux store config



└─ constant	
└─ reducer	- react redux reducer tasks/definitions
└─ scene	- UI and logic for each of the screen
shown	
in the app	
└─ service	
└─ AspectService.js	- common service request response
operations	
└─ DeviceService.js	- common sensor/thing related operations
└─ LocalStorageService.js	- store/restore to/from local storage
└─ MeasurementUnitService.js	- measurement units related operations
└─ MQTTClientService.js	- mqtt client
└─ ObservationService.js	- sensor observation related operations
└─ PushNotificationService.js	- push notification service
└─ ServiceRequestParser.js	- service request parser for .owl and .wsdl files

After the parsing is complete, each of the inputs is validated against the ontology JSON object, checking the first child object, which will be referred here as (I), of each of the three main components: domain, monitoring aspects and measurement type, is present in the taxonomy. In case it is, it then proceeds to check if the next element, hereby called (II), of each of the components mentioned before is contained as a child of the element (I). As an example, if the value of the child of “measurement type”, which would be an example of (I), is “environment”, the element of the next child, which would be an example of (II), must be a son of “environment” in the taxonomy. This procedure is repeated for each of the inputs present in the description file in case there are multiple. The code snippet shown next is taken from an input object that follows the cross-domain proposed ontology:

```

1. inputObject = {
2.     "domain": {
3.         "building management": "offices"
4.     },
5.     "monitoring aspect": {
6.         "places": "offices"
7.     },
8.     "measurement type": {
9.         "environment": "temperature"
10.    },

```

```
11.  };
```

If all the three attributes of the service are according to the ontology, then the parser creates a request object to be sent to the TSMATCH engine. that follows a structure of the **"INNOVINT/HAMBURG\_FACTORY1/REQUEST"** MQTT topic that is sent as the **"return"** variable of the parser module. If any incorrection is found, another object with the kind of validation error is then sent. The latter object contains 6 elements: domain, domain detail, monitoring aspect, monitoring aspect detail, measurement type and feature of interest that are shown only if there is a validation mistake related to it. It also states which of the inputs from the description file contained that mistake. In a file with n inputs, "input 1" would be the first input from top to bottom in the **"interface"** element of the file and "input n" would be the last.

As an error proofing method, there is a function that turns all the contents of the ontology asset to lowercase, be it from an external XML file or from the taxonomy object declared inside the software; and the parser software also transforms every value extracted from the description files to lowercase. Additionally, since WSDL and OWL-S files do not accept spaces in the names and parameters of its elements, but only underscores and hyphens, the parser software transforms every hyphen and underscore contained in those elements into spaces to follow TSMATCH's object structure.

Moreover, as of the composition of this report, TSMATCH only recognizes values of the taxonomy where each word begins in uppercase values. Therefore, if "domain" contains values such as "health" or "Smart city", the software would not be able to do the matching, so those values mentioned before should be sent as "Health" and "Smart City". Thus, after creating the request object that will be sent to the TSMATCH App, the parser software transforms every letter at the beginning of a word into uppercase.

### 3.2.1. Detailed Code Description

The parser component is present in the [ServiceRequestParser.js file](#). It exports a class to TSMATCH Mobile APP that Parses description files sent by external services. Its first method, **getRequestsFromURL**, takes the URL provided by the consumer service and extracts its content, which should be a file. It then checks if the file has the correct extensions. If it has a valid extension, it checks if it is OWL-S or WSDL.

In case of WSDL it checks if the file is version 1.0 or 2.0 and if the file contains prefixes or not and extracts the result, it saves the version of the language and presence or absence of prefixes in four variables called **wsdl1**, **wsdl1Prefix**, **wsdl2**, **wsdl2Prefix**, that contain values of **true** or **false** depending on the condition met. Afterwards, a variable called **version** receives the only value among **wsdl1**, **wsdl1Prefix**, **wsdl2** and **wsdl2Prefix** that has a **true** statement.

If the conditions **wsdl1** or **wsdl1Prefix** are met, the contents are sent to the **wsdl1Parsing** method. This method is divided into two parts: parsing and validation. The parsing first checks if the WSDL 1 file has prefixes or not, by the **version** variable sent from the **getRequestsFromURL** method, adds them to the parsing mechanism if necessary and extracts the file's content as explained before.

The validation process from the method gets the inputs extracted from the file and compares them to the ontology located at [ServiceRequestParserConfig.js](#) according to the 3 domain attributes that are required

by TSMatch's Ontology and by the "INNOVINT/HAMBURG\_FACTORY1/REQUEST" topic's message structure. These elements are "domain", "monitoringAspect" and "measurementType".

If the conditions **wsdl2** or **wsdl2Prefix** are met, the contents are sent to the **wsdl2Parsing** method. The difference of the **wsdl2Parsing** and **wsdl1Parsing** methods are how the files are parsed due to differences in their structure. Basically, WSDL version 1 has one extra attribute called "message", that must be referenced between "interface" and "types" attributes. The validation process is the same for both.

If the file is OWL-S no condition is met in the **getRequestsFromURL** method and the contents extracted from the URL are sent to the **owlParsing** method. Similarly to the two methods above, it first parses the file to extract the inputs according to the OWL-S structure explained in 2.2.1 and then validates them. However, for OWL-S there is a taxonomy prefix linked to the inputs that was explained in chapter 2.2 that must be checked before validation.

Currently, due to requirements of TSMatch, there is a method called **titleCase** that transforms the first letter of every word in the request sent to TSMatch Mobile App into Uppercase. This method is called after validating every input from the description files in all three methods explained above: **wsdl1Parsing**, **wsdl2Parsing** and **owlParsing**.

### 3.2.2. How to Run

The parser component is currently installed on the TSMatch client, for which a mobile App has been developed by fortiss. The GUI is illustrated in Figure 9. Currently, the client can get a service description from an URL. The file can be contained in a registry or inside the web service's URL domain.



Figure 9: TSMatch client developed for Android.

The entered URL is sent to the Parser component, that fetches the file from the URL, parses it, validates its contents according to the taxonomy and generates TSMATCH's request object in the format of the topic **"INNOVINT/HAMBURG\_FACTORY1/REQUEST"**. The request object structure is shown in code example 8.

```
1. var requestObject = {
2.   "featureOfInterest": "",
3.   "unitOfMeasurement": "unit",
4.   "threshold": "30",
5.   "creatorId": "FORTISS_IOTLAB_DESCRIPTION_PARSER",
6.   "location": "Per-room",
7.   "uuid": uuid.v4(),
8.   "domain": "",
9.   "domainDetail": "",
10.  "monitoringAspect": "",
11.  "monitoringAspectDetail": ""
12. };
```

**Code Example 8** – Request object structure for TSMATCH Request topic.

### 3.2.3. Limitations

There are three methods in the Parser software that use the message structure of the **"INNOVINT/HAMBURG\_FACTORY1/REQUEST"** topic, they are: **"wsdl1Parsing"**, **"wsdl2Parsing"**, and **"owlParsing"**. If any changes are made to that topic's message structure, the global object variable named **"requestObject"** must be changed to follow the new structure.

Moreover, there are 3 elements in the taxonomy structure that are used in the Parser, they are: **"domain"**, **"monitoring aspect"**, and **"measurement type"**. If those name definitions are changed in the taxonomy or if the ontology is extended, their respective changes and the addition of elements must be included throughout all the Validation Process of the **"wsdl1Parsing"**, **"wsdl2Parsing"** and **"owlParsing"** methods. Examples of lines of code where changes would be necessary are shown in code example 9.

```
318   inputs[i][j][0].replace(/_/-/g, ' ').toLowerCase().trim() === 'domain'
319   if (inputs[i][j][1].replace(/_/-/g, ' ').toLowerCase().trim() in config['taxonomy']['domain'])
320     request["domain"] = inputs[i][j][1].replace(/_/-/g, ' ').toLowerCase().trim();
321   let element = config['taxonomy']['domain'][request["domain"]];
```

**Code Example 9** – Example of lines of code that would need changing.

The lines of code above use the **"domain"** tag from TSMATCH's taxonomy and **"domainDetail"** from the request object structure. In addition, the equivalent lines for **"monitoring aspect"** and **"measurement type"** for the taxonomy, and **"monitoringAspectDetail"** and **"featureOfInterest"** for the request object would also need to be changed accordingly.

### 3.3. External Service Interface (ESI)

To allow for scalability and the interconnection of external services, we have developed an “[External Service Interface](#)”<sup>2</sup>, based on REST. This interface receives HTTP messages from external services containing their requirements, manages the communication between the services and the TSMATCH API, and provides an [OpenAPI YAML description](#) so services can more easily communicate with it. OpenAPI is a description language for REST interfaces created by Swagger, it follows a JSON or YAML structure, two of the most used, and it’s simpler to understand for humans when compared to WSDL or OWL-S, which makes it easier to be implemented and used. Table 2 shows the file structure of the External Service interface component.

Table 2: External Service Interface code structure.

API_Connector.		
├─ config		
├─ API_connector_parameters.js	- Parameters for the API app	
├─ mqtt_parameters.js	- Parameters for the MQTT communication	
├─ observation_parameters.js	- Parameters for the WebSocket app	
└─ taxonomy.js	- Taxonomy JavaScript object	
├─ Dockerfile	- Docker image configurations	
├─ Index.js	- Main file, starts API and WebSocket apps	
├─ models		
├─ description.js	- Validation class for requests coming from the client service	
└─ mqtt_client.js	- MQTT class for stablishing connection, publishing and deleting requests on TSMATCH	
├─ observation_server		
└─ index.js	- WebSocket class to configure the server and communication with TSMATCH and client services	
├─ routes		
└─ index.js	- HTTP app, defining POST and DELETE methods and communication with MQTT broker	

---

<sup>2</sup> The external service interface is currently bamed as API connector. However, this will be changed.

└─ specification	
└─ specification.json	- OpenAPI Specification for the API_Connector in json format
└─ specification.yaml	- OpenAPI Specification for the API_Connector in yaml format
└─ start_connector_api.sh	- API_Connector docker image runner

Moreover, Figure 10 illustrates the communication sequence between an external service and TSMatch, via this interface.

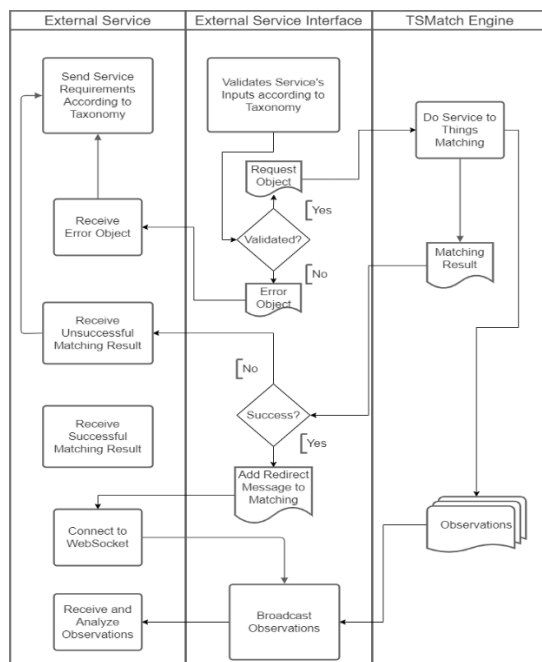


Figure 10: Communication Diagram for the API Connector.

Similarly to the parser component, the ESI needs to validate request elements according to the cross-domain ontology adopted, they should not distinguish uppercase from lowercase letters. The [ontology](#) is currently contained as an asset inside the software that can be imported from an XML file or filled as a JavaScript object. Consequently, as an error proofing method, there is a function that turns all the contents of the taxonomy asset to lowercase, be it from an external XML file or from the taxonomy object declared inside the software; and the parser software also transforms every value extracted from the service's request message to lowercase.

The [HTTP connector](#) was built using the Express library from Nodejs and was implemented in the port 3003 of the IIoT gateway. This connector has two path routes: **"/openapi/allrequests"** and **"/openapi/request"**. The first path can receive a DELETE message that deletes all previous requests made

by the API connector from the demonstrator. The latter path can receive a DELETE message that contains the uuid of a specific request as a path parameter, the request that had that uuid is then deleted from the database. Both DELETE requests mentioned above use the “**INNOVINT/HAMBURG\_FACTORY1/DELETE\_REQUEST**” TSMATCH MQTT topic for the deletion procedure.

The “**/openapi/request**” path can also receive a POST message where the POST body should be a JSON object following the structure (III) explained above, it sends that body to the previously mentioned method and, if no validation error is received, subscribes the return value in the “**INNOVINT/HAMBURG\_FACTORY1/REQUEST**”. The connector waits for a response from the MQTT broker’s “**INNOVINT/RESPONSE/NDATA/TSMATCH\_INNOVINT\_1**” topic for up to 5 seconds that is sent back to the consumer service.

In case the message received from the broker states that the matching was not possible, that message is forwarded to the external service without modifications. Nevertheless, if the message informs a successful matching, the connector adds one additional parameter to the response object, containing the path to a WebSocket inside the connector component that the external service should connect to for receiving the observations from the matching result.

This [WebSocket was implemented](#) in the port 3004 of the fortiss IIoT gateway and has the function of keeping a two-way communication between TSMATCH and the external services that are receiving observations from successful matchings. It broadcasts information from all observations received from the “**INNOVINT/OBSERVATION/NDATA/TSMATCH\_INNOVINT\_1**” topic in the WebSocket connection if there is at least one requester service connected to it.

### 3.3.1. Detailed Code Description

The [ESI code](#) contains an **index.js** file that initializes all its application components, which are the HTTP API and the WebSocket. It contains two specification files in [JSON and YAML](#) formats. This component also has a configuration folder called **config** where the REST parameters, the MQTT communication parameters, the WebSocket parameters, the taxonomy object and the ontology importer are located, respectively in the [API connector parameters.js](#), [mqtt parameters.js](#), [observation parameters.js](#), [taxonomy.js](#) and [XML Taxonomy reader.js](#) files.

This component contains a folder called [models](#) where both the validation method and the MQTT\_client connection are setup, both located at The validation method is in the **description.js** file and is contained inside a class called **Description**. This class has a constructor with the same structure as the request body from the POST messages sent by external clients, explained in section 2.4.1, that receives these messages and passes them to the **generateRequestObject** method. This latter method gets the requests, validates them the same as done in the **wsdl1Parsing** and **wsdl2Parsing** methods explained above and creates a request in the “**INNOVINT/HAMBURG\_FACTORY1/REQUEST**” topic’s message structure.

The MQTT client connection and communication methods from the External Service Interface and the TSMATCH broker is defined as a class called **MQTTClient** in the **mqtt\_client.js** file. It uses the connection parameters from the **config** folder to create an MQTT connection, and in its constructor defines all topics that need subscription when the connection is established. It has three methods called **connect**, **request**, and **delete** that connect to the broker, create requests to it and delete requests sent to it, respectively.

The HTTP API is in the [routes](#) folder's **index.js**. Inside it has 6 global variables related to the communication with the MQTT broker: the **array** variable defines a global array that saves payloads from messages received from the broker; the **resp** variable defines a global response object to check for HTTP messages; **n** and **count** variables are currently not used but they define a maximum number of messages to wait for from the broker before resetting the **array** variable and counts the number of messages already sent, respectively; the **timer** and **timeOutValue** variables are global definitions to wait for messages from the broker before closing the HTTP connection; and the **uuidList** is an array that saves uuid from messages sent from the connector.

Afterwards, a connection is created with the MQTT broker by calling the **MQTTClient** class and all the procedures to be taken when a message is received are defined inside the **connection.client.on('message', ...)** function. The latter function checks if the matching was successful via the **INNOVINT/RESPONSE/NDATA/TSMATCH\_INNOVINT\_1** topic, if the message contains a **requestId** parameter the matching was successful, otherwise it failed. It proceeds to check if the response message is linked to a request sent from the connector API by checking if the object's id is contained in the **uuidList** array. If all conditions are met the response message is sent back to the external service with a redirect parameter to the WebSocket.

Then, an HTTP API is created at port 3003 and has three "routes" as explained: a POST to **"/openapi/request"**, a DELETE to **"/openapi/request"**, and a DELETE to **"/openapi/allrequests"**. The POST uses the request body sent from the external service to create a **Description** class instance. In case the object wasn't validated it sends the error message back to the client, and in case it was validated it saves the generated uuid in the **uuidList** array and publishes the request in the MQTT topic called **"INNOVINT/HAMBURG\_FACTORY1/REQUEST"**. The DELETE to **"/openapi/request"** receives an id as a path parameter, publishes a delete request for that id and extracts that uuid value from the **uuidList** array. The DELETE to **"/openapi/allrequests"** uses all the uuid inside the **uuidList** array to send delete requests for every single one of them.

The WebSocket is setup in the [observation server folder's index.js](#). It creates a WebSocket server in port 3004 that connects to MQTT broker and sends a response header once a client is connected to it. This server broadcasts observations from TSMATCH's successful matching received from the **"INNOVINT/OBSERVATION/NDATA/TSMATCH\_INNOVINT\_1"** topic.

Currently, there is no distinction in messages sent to each client, instead they are broadcasted to every client connected to the WebSocket. To create a more secure connection the uuid of each client could be sent as a parameter in the WebSocket's path that can be read by the **path**, **uuid**, and **metadata** variables already defined in the WebSocket's code. Afterwards each message can be sent separately to each client using their unique id or encrypted according to the uuid so that only the service related to that uuid can decrypt the message's contents.

The External Service Interface also contains a [Dockerfile with its docker image configurations and a start connector api.sh file that runs the Docker image](#).

### 3.3.2. How to Run

The external service interface needs to be started independently with the following :

```
~/.> cd API_Connector
```



```
~/> chmod +x start_api_connector.sh
```

```
~/> ./start_api_connector.s
```

To use the External Service Interface, the external service, via the current TSMATCH client, needs to connect to the TSMATCH engine, running via IP:port 10.0.33.39:3003.

To send requests to TSMATCH, first a POST message must be sent to the **“/openapi/request”** path with the request body containing the JSON object with the structure shown in code example 11, filled with string values from the ontology.

```
{  
  
    "domain": "",  
  
    "domainDetail": "",  
  
    "monitoringAspect": "",  
  
    "monitoringAspectDetail": "",  
  
    "measurementType": "",  
  
    "featureOfInterest": "",  
  
};
```

#### **Code Example 11** – Request body containing the JSON object structure

Once a request is validated against the ontology format, the ESI wraps the service’s requirements in the **“INNOVINT/HAMBURG\_FACTORY1/REQUEST”** topic’s message structure and publishes it to the MQTT broker. The ESI waits up to five seconds for a response from the broker.

After a response connected to the request previously sent is received from the **“INNOVINT/RESPONSE/NDATA/TSMATCH\_INNOVINT\_1”** topic, the ESI checks if the matching was successful. In case of failure, the response is directly forwarded back to the external service. In case of success, the path 10.0.33.39:3004 to the WebSocket is added to the response and sent back to the TSMATCH client.

Once the external service connects to the WebSocket, it will constantly receive all observations sent by the **“INNOVINT/OBSERVATION/NDATA/TSMATCH\_INNOVINT\_1”** topic from the MQTT broker.

### 3.3.3. Dealing with changes to the TSMATCH Structure and to the Taxonomy

In the API Connector, there are methods that use the message structure of the following topics:

“INNOVINT/HAMBURG\_FACTORY1/REQUEST”, hereby called (1).

“INNOVINT/HAMBURG\_FACTORY1/DELETE\_REQUEST”, hereby called (2).

“INNOVINT/RESPONSE/NDATA/TSMATCH\_INNOVINT\_1”, hereby called (3).

“INNOVINT/OBSERVATION/NDATA/TSMATCH\_INNOVINT\_1”, hereby called (4).

If changes are made to topic (1), the “request” variable in “generateRequestObject” method inside “models/description.js” file will have to change accordingly. If changes are made to topic (2), the “newId” variables in both DELETE methods from “routes/index.js” will have to change accordingly.

Changes regarding topic (3) will imply in modifications to the “connection.client.on(‘message’, ...)” function that receives messages from the MQTT broker. It currently uses the object keys “requestId” and “grouping”, present it topic (3) message structure to identify if the matching was successful and if the response is linked to a request from the API Connector, respectively. Topic (4) simply sends the observations directly to connected client services and requires no changes.

In similarity to the Parser Software, there are 3 elements in the taxonomy structure that are used in the API Connector, they are: “domain”, “monitoring aspect”, and “measurement type”. If those name definitions are changed in the taxonomy or if the ontology is extended, their respective changes and the addition of elements must be included in the Validation Process of the “generateRequestObject” method inside “models/description.js” file. Examples of lines of code where modifications would be necessary are present in the transcript inside **code example 12**.

```
132  if(this.measurementType.toLowerCase().trim() in config['taxonomy']['measurement type'])
{

133          let element = config['taxonomy']['measurement
type'][this.measurementType.toLowerCase().trim()];

135  if(typeof(element[Object.keys(element)[0]]) != 'object') {

136      if(element.includes(this.featureOfInterest.toLowerCase().trim())){

137          request["featureOfInterest"] = this.featureOfInterest.toLowerCase().trim();
```

**Code Example 12** – Lines of code that would require modifications.

The lines of code above use the “measurement type” tag from TSMATCH’s taxonomy and “featureOfInterest” for the request object structure. In addition, the equivalent lines for “monitoring aspect” and “domain” for the taxonomy, and for “monitoringAspectDetail” and “domainDetail” for the request object would also need to be changed accordingly.

## 4. Summary and Next Steps

This work describes different components that have been developed to extend the TSMatch demonstrator of fortiss. The internship project had three main tasks: i) creation of semantic descriptions based on specific languages; ii) development of a parser component capable of checking the service descriptions against a specific ontology; iii) development of an interface (HTTP) to interconnect TSMatch to external services with an OpenAPI specification to handle communication with services that wish to provide their requirements through HTTP.

Further implementations on the software created during this internship could include: improving the parser by adding synonym and typing error recognition; adding data security to the WebSocket interface inside the API Connector by assigning an uuid to each client connection and send observations back only to the client whose request is linked to that observation; using the same taxonomy structure for all components of the demonstrator, currently the App, the Engine and the API have different versions of the taxonomy; improving the current taxonomy; and adding a general ontology validation method that works for other ontologies. However, the latter suggestion would imply in equivalent changes in the TSMatch API and in the Mobile App, since the topic's communication structure used in the MQTT communication require parameters that are linked to the TSMatch ontology, such as "domain", "monitoring aspect", "monitoring aspect detail", "measurement type", and "measurement type detail".

## 5. References

- [1] Jia, Bing, Wuyungerile Li, and Tao Zhou. "A centralized service discovery algorithm via multi-stage semantic service matching in internet of things." In *2017 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC)*, vol. 1, pp. 422-427. IEEE, 2017.
- [2] Kolbe, Niklas, Jérémy Robert, Sylvain Kubler, and Yves Le Traon. "Proficient: Productivity tool for semantic interoperability in an open iot ecosystem." In *Proceedings of the 14th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, pp. 116-125. 2017.
- [3] Palavalli, Amarnath, Durgaprasad Karri, and Swarnalatha Pasupuleti. "Semantic internet of things." In *2016 IEEE Tenth International Conference on Semantic Computing (ICSC)*, pp. 91-95. IEEE, 2016.
- [4] SWAGGER. OpenAPI Specification. Available at (consulted in November 2021): <https://swagger.io/specification/>
- [5] Martin, David, Mark Burstein, Jerry Hobbs, Ora Lassila, Drew McDermott, Sheila McIlraith, Srin Narayanan et al. "OWL-S: Semantic markup for web services." *W3C member submission* 22, no. 4 (2004).
- [6] Booth, David, and Canyang Kevin Liu. "Web services description language (WSDL) version 2.0 part 0: Primer." *W3C recommendation* 26 (2007): 39-41.
- [7] Chinnici, Roberto, Jean-Jacques Moreau, Arthur Ryman, and Sanjiva Weerawarana. "Web services description language (wsdl) version 2.0 part 1: Core language." *W3C recommendation* 26, no. 1 (2007): 19.
- [8] Nitzsche, Jorg, Tammo Van Lessen, and Frank Leymann. "WSDL 2.0 message exchange patterns: limitations and opportunities." In *2008 Third International Conference on Internet and Web Applications and Services*, pp. 168-173. IEEE, 2008.
- [9] Christensen, Erik, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. "Web services description language (WSDL) 1.1." (2001).
- [10] Rozsa, Vitor, Marta Deniszczwicz, Moisés Lima Dutra, Parisa Ghodous, Catarina Ferreira da Silva, Nader Moayeri, Frédérique Biennier, and Nicolas Figay. "An Application Domain-Based Taxonomy for IoT Sensors." In *ISPE te*, pp. 249-258. 2016.
- [11] Bnouhanna, Nisrine, Rute C. Sofia, and Alexander Pretschner. "IoT Thing To Service Semantic Matching." In *2021 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, pp. 418-419. IEEE, 2021.
- [12] E. Karabulut, N. Bnouhanna, R. C. Sofia. *ML-Based Data Classification and Data Aggregation on the Edge*. Poster, student workshop, CoNEXT2021. December 2021.