NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Optimizing Service Provider Selection and Managing Data Translation Processes

BSc in Electrical and Computer Engineering

AFONSO TIAGO MARTINS NUNES

NOVA University Lisbon
September, 2024

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF
Electrical and Computer Engineering

Optimizing Service Provider Selection and Managing Data Translation Processes

AFONSO TIAGO MARTINS NUNES

BSc in Electrical and Computer Engineering

Adviser: João Almeida das Rosas
Assistant Professor, NOVA University Lisbon

Co-adviser: Filipe de Carvalho Moutinho
Assistant Professor, NOVA University Lisbon

Examination Committee:

Chair: Paulo José Carrilho de Sousa Gil
Professor, NOVA University Lisbon

Rapporteur: João Pedro Leal Abalada de Matos Carvalho
Professor, Universidade Lusófona

Adviser: João Almeida das Rosas
Professor, NOVA University Lisbon

MASTER IN ELECTRICAL AND COMPUTER ENGINEERING

NOVA University Lisbon
September, 2024

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my adviser, João Rosas, and my co-adviser, Filipe Moutinho, for their invaluable guidance and strong support throughout this journey. There is nothing I would change about the way I was mentored. Both helped me in different ways that complimented each other. Besides the availability to help for hours at the time if needed, in the moments where I felt like I lost my way, a new path to move forward was shown to me. Never have I felt pressured or that what I had worked on so far "wasn't good enough" even if that was the case, I would be guided to a better path, not be reminded of it. If I was ever losing interest in my work and its potential, every new meeting my spark would be reignited through their thoughtful advice and insight. The knowledge they shared with me, both technical and in writing, was far beyond what I could have hoped for, and for that, I am profoundly thankful.

I would also like to extend my appreciation to Nova School of Science and Technology for providing me with the necessary tools and resources to complete this journey.

To the friends I've made along the way, who have stayed by my side, I owe a debt of gratitude. These friendships are among the most important things in life, the ones forged during these challenging journeys. Especial thanks to Luís, for being my partner in almost every project I have ever done, and to Mafalda, who has been my strong companion throughout the demanding process of completing this dissertation.

Finally, I want to thank my parents, who have given me everything I needed to succeed and allowed me to grow into the person I am today, or rather, the person I am capable of becoming. I am also grateful to my sister, who set the standard for how a degree should be pursued and completed.

“Today I escaped anxiety. Or no, I discarded it, because it was within me,
in my own perceptions — not outside.” (Marcus Aurelius).

Abstract

This thesis addresses the critical challenge of achieving seamless interoperability among heterogeneous systems, a fundamental requirement to meet the dynamic demands of Industry 4.0. As the industrial landscape evolves with the increasing integration of IoT devices, automation, and data-driven decision-making, ensuring effective communication across diverse systems is imperative. This research focuses on bridging the gap between the Arrowhead Framework (AF), a service-oriented architecture designed to facilitate IoT and automation interoperability, and the Translator Automatic Generator (TAG), a data translation tool. Together, these technologies are leveraged to overcome limitations in service selection and data compatibility.

The primary objective is to develop the Data Translation Manager (DTM), a novel tool capable of automating the selection of the most appropriate service providers for consumers within an Arrowhead cloud environment. Simultaneously, the DTM facilitates dynamic data translation through the TAG Tool, enabling communication between previously incompatible systems at both syntactic and semantic levels.

The DTM empowers systems to dynamically select optimal service providers, adapt to real-time changes, and execute automated data transformations by establishing a cohesive integration between AF and TAG. This enhances the interoperability, communication, and integration of IoT systems within Industry 4.0 ecosystems.

RESUMO

Esta dissertação aborda o desafio de aumentar as capacidades de interoperabilidade entre sistemas heterogéneos, um requisito fundamental para atender às exigências dinâmicas da Indústria 4.0. À medida que o panorama industrial evolui com a crescente integração de dispositivos IoT, automação e a tomada de decisões baseadas em dados, torna-se imprescindível garantir uma comunicação eficaz entre sistemas diversos. Este trabalho centra-se em colmatar a lacuna entre o Arrowhead Framework (AF), uma arquitetura orientada a serviços concebida para facilitar a interoperabilidade no contexto do IoT e da automação, e o Translator Automatic Generator (TAG), uma ferramenta de tradução de dados. Estas tecnologias são integradas para superar as limitações relacionadas à seleção de serviços e à compatibilidade de dados.

O objetivo principal é desenvolver o Data Translation Manager (DTM), uma ferramenta nova capaz de automatizar a seleção dos fornecedores de serviços mais adequados para os consumidores num ambiente cloud Arrowhead. Simultaneamente, o DTM facilita a tradução dinâmica de dados através da ferramenta TAG, permitindo a comunicação entre sistemas previamente incompatíveis, tanto em níveis sintáticos como semânticos.

Ao integrar de forma eficiente o AF e o TAG, o DTM permite que sistemas selecionem os fornecedores de serviços mais adequados de maneira dinâmica, que se ajustem a alterações em tempo real e que realizem transformações de dados automatizadas. Esta abordagem não apenas responde a alguns desafios atuais da interoperabilidade, mas também promove uma comunicação mais fluida e integração robusta entre sistemas IoT, posicionando-se como um contributo para a evolução dos ecossistemas da Indústria 4.0.

CONTENTS

LIST OF FIGURES

LISTINGS INDEX

ACRONYMS

AF - Arrowhead Framework

API - Application Programming Interface

CPS - Cyber-Physical Systems

DTM - Data Translation Manager

HTTP - HyperText Transfer Protocol

IoT - Internet of Things

JSON - JavaScript Object Notation

RDF - Resource Description Framework

REST - Representational State Transfer

SAWSDL - Semantic Annotations for WSDL and XML Schema

SD - Service Discovery

SOA - Service-Oriented Architectures

SoS - System of systems

SPARQL - SPARQL Protocol and RDF Query Language

SR - Service Registry

TAG - Translator Automatic Generation

WSDL - Web Services Description Language

XML - eXtensible Markup Language

1

INTRODUCTION

1.1 Motivation

Each day brings continuous evolution in the technological frameworks that support our modern world. As Industry 4.0 reshapes the landscape of various sectors [1], [2], from smart city infrastructure [3] and factory automation to healthcare information systems [4], the demand for adaptive and efficient communication systems becomes a necessity. This transformation is driven by the integration of sensors and smart devices within these sectors, creating complex networks that require seamless data exchange. However, achieving this requires more than just technological advancements—it requires the ability for diverse systems to communicate effortlessly across different platforms and architectures.

Interoperability is a key factor in this dynamic, enabling systems that were never designed to interact to exchange information in real-time. This concept is central to Industry 4.0's vision [2], where systems must dynamically adapt to changes in the environment, resource availability, and user needs. Despite advancements, many existing frameworks still struggle to achieve the high levels of interoperability needed for complex systems. They often require the support of additional layers or tools to better interactions, particularly in environments where heterogeneous systems coexist.

This work is driven by the need to enhance existing interoperability frameworks, with a focus on improving communication between heterogeneous systems, services, and platforms. The research investigates how current systems can be optimized to enable seamless interaction, addressing both data and semantic interoperability challenges. A critical aspect of this work is the development of tools and methods that integrate into established frameworks, facilitating interoperability not only at the structural level but also at the semantic level.

Furthermore, the goal is to ensure these systems can dynamically identify the most efficient communication pathways, enabling previously incompatible consumer and provider systems to exchange information effectively.

Through this work, we aim to create tools that contribute to an environment where communication between previously incompatible systems becomes possible.

1.2 Problems and Objectives

While technologies like the Arrowhead Framework [5] are designed to facilitate system interoperability, they come with certain limitations. For instance, Arrowhead focuses primarily on service-level communication, but it does not address data and semantic interoperability. This becomes an issue when systems need to exchange information but have different semantics and data formats. The TAG Tool [6], [7], was developed to address this gap by enabling data transformation, thus enhancing data and semantic interoperability. However, despite their complementary purposes, the Arrowhead Framework and the TAG Tool are not currently integrated.

This work aims to bridge the gap between these two technologies by directly linking them to work with each other through an intermediary. By integrating the TAG Tool and the Arrowhead, we can facilitate seamless communication between systems that not only need to connect but also need to translate and interpret each other's data. This integration will also put into practice what is proposed in [6], [7] demonstrating how these tools can complement each other. A major objective is to explore both technologies and develop a working solution that connects them effectively.

In addition to bridging these technologies, another core objective of this work is to develop a selection process when multiple provider systems are available. This research aims to enhance that selection process, as well as manage and execute the generated translators, ensuring that the most suitable provider, based on semantics and data quantity, is chosen for any given task. By integrating the TAG Tool with Arrowhead, we hope to achieve a more intelligent and efficient system for selecting providers, ultimately improving overall system interoperability.

1.3 Document Structure

This thesis is organized into several chapters, each building upon the foundational concepts and presenting both the theoretical and practical advancements made throughout this work.

Chapter 2 - State of the art: This chapter explores the relevant context and background for the thesis, including key concepts and technologies central to the work. A thorough review of the existing literature and related works in the field is provided to frame the research problem and justify the need for the proposed solution. Topics such as Industry 4.0, interoperability frameworks, microservices, and the Arrowhead framework are discussed in detail.

Chapter 3 - Proposed system: In this chapter, the system design, methodology, and implementation are introduced and discussed in detail. The architectural components of the proposed system, such as the Data Translation Manager (DTM), consumer systems, and provider systems, are explained. It also details the development process, describing key implementation steps, integration with Arrowhead, and the challenges encountered during development, along with the solutions applied.

Chapter 4 - Tests and Validation: This chapter focuses on testing the system under real-world conditions. It presents a variety of test cases to assess how the system connects the different elements, reacts to different provider availability scenarios, metadata requirements, and the translation of data.

Chapter 5 - Conclusions: The thesis concludes by summarizing the work, its contributions to system interoperability, and the technological advances made. It also identifies potential areas for further research, suggesting ways in which the system could be improved or extended to other frameworks and applications.

2

STATE OF THE ART

The goal of this chapter is to provide a comprehensive review of the concepts and state of the art technologies relevant to the development of the Data Translation Manager (DTM), which integrates the TAG Tool with the Arrowhead Framework. By examining existing literature and current advancements in the fields of service orchestration, data interoperability, and system integration, we establish a solid foundation for the ideas proposed in this thesis.

2.1 Automation, IoT and cyber-physical systems

In modern scientific and technological landscapes, Automation, Internet of Things (IoT), and Cyber-Physical Systems (CPS) collectively represent a transformative pathway, it is, therefore important to have a basic understanding of these concepts.

Automation is the concept of technology performing tasks seamlessly, reducing the need for direct human intervention. The purpose is to enhance efficiency and facilitate processes by allowing systems to operate independently.

The **Internet of Things** takes the centre of the stage by connecting everyday devices to the Internet, enabling them to exchange data. IoT is seen as a global network of interconnected things, both physical and virtual, that can share information over standard and interoperable communication protocols, leading to enhanced functionality and real-time insights [8].

Cyber-Physical Systems merge computational and physical processes into integrated units. These systems combine virtual and physical components to enable real-time monitoring, control, and decision-making across various domains. In many ways, CPS can be seen as a subset of IoT, focused on the seamless interaction between the digital and physical worlds.

All these domains are used to create intelligent, connected systems capable of real-time data acquisition and analysis. Scientific exploration in this field often delves into developing efficient algorithms, communication protocols, and security mechanisms to reach the full potential of said systems. There is endless motivation to do so as the number of installed IoT devices has grown exponentially in recent years and will continue to do so in the future [8] [4]. This work focuses on improving communication within these vast networks of interconnected systems.

2.2 Interoperability

Interoperability can be simply defined as "the ability of two or more systems or components to exchange data and use information" [9]. It is a crucial pillar in technological integration, particularly within IoT and cyber-physical systems. As one author noted, "The IoT is and will continue to be a heterogeneous, multi-vendor, multi-service and largely distributed environment; these are good ingredients for interoperability problems" [10]. This concept refers to the seamless communication and information exchange between diverse systems and their components, regardless of their differences.

In IoT, achieving fundamental objectives depends on establishing interoperability, whether it's transmitting data from a sensor to trigger an actuator or displaying remote readings on a user interface. This work focuses on reducing the number of devices unable to communicate or understand one another’s messages, using specific tools to address these interoperability challenges.

There are four primary sub-types of interoperability: technical, syntactic, semantic, and organizational interoperability. While many modern systems are designed with these in mind, compatibility issues can arise, particularly when integrating older systems or modules. In this work, the focus is on syntactic and semantic interoperability, two key aspects of "data interoperability" [10]. These ensure that systems can not only read and interpret structured data but also understand the meaning of the associated metadata, which is essential in the context of this thesis.

The most comprehensive form of interoperability is dynamic interoperability, which ensures the sustainability of seamless communication in highly diverse, dynamic, and large-scale environments [10], [11]. Given the inherent heterogeneity of IoT systems [8], dynamic interoperability becomes critical. In practical scenarios, devices from different vendors may need to

communicate, even if they use different message formats or protocols. The ability to adapt to these differences and maintain communication makes them dynamically interoperable. A framework like the Arrowhead Framework can often address these challenges, but in cases of significant heterogeneity, additional steps, such as message translation, may be required.

A real-world example illustrating the importance of interoperability is in the healthcare domain. Healthcare professionals need up-to-date information about patients and medical equipment, while healthcare associations and laboratories must track patients who require financial assistance or specific disease research. The ability for all these systems to communicate effectively is essential for providing quality care.

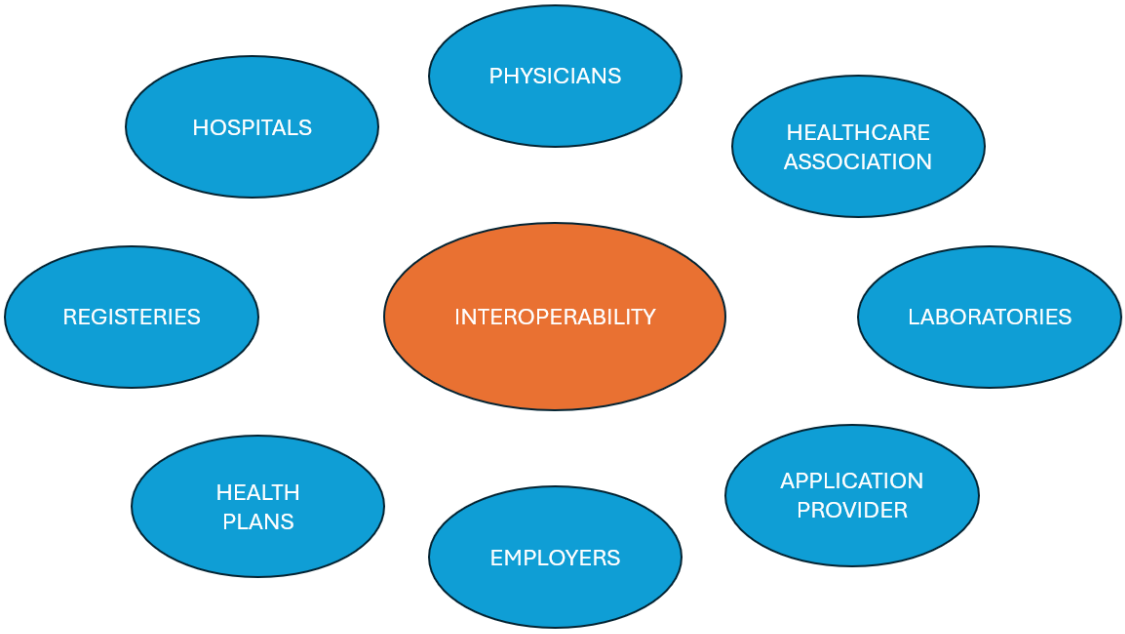


Figure 2.1 - Interoperability in the Health Domain, adapted from [4]

2.3 The Arrowhead Framework

New systems, even if from different technologies, can be custom-built in compliance with framework specifications, allowing them to interact directly with several different systems. One such framework is the Arrowhead Framework (AF), an innovative solution in the domain of service-oriented architectures and interoperability. AF addresses the challenges of seamlessly integrating heterogeneous systems rooted in diverse technologies [12]. Based on service-oriented architecture (SOA) principles, AF facilitates the dynamic interaction between systems by providing a robust set of core services. These services include:

Service Registry (SR): A key component that acts as a centralized repository for registering and storing information about services within the Arrowhead ecosystem. It maintains a directory of services, including their descriptions, interfaces, and locations. When a new service is deployed, it registers with the SR, allowing other systems to dynamically discover and access these services.

Service Discovery (SD): A process that enables systems within the Arrowhead Framework to dynamically find and locate available services in real time. Closely linked to the SR, SD involves querying the registry for service information, this service might be included in the orchestrator and SR. Systems use this service to connect to services based on their specific requirements, promoting flexibility and adaptability in system architecture.

Orchestration (ORC): Involves coordinating and managing multiple services to achieve a specific goal. It enables the composition and execution of workflows involving different services' interaction. This core service manages the workflow, ensuring that each service is provided to the most appropriate consumer.

Authorization (AUT): A critical service for ensuring security and controlled access within the Arrowhead ecosystem. It verifies the permissions and privileges of systems or users attempting to access specific services. Authorization mechanisms enforce access control policies, protecting the system from unauthorized use.

In a system using the core Arrowhead Frameworks services, the typical interaction between a consumer and a provider could be described in the following way: As illustrated in Figure 2.2, first the providing device must register its services in the SR, so when a consumer needs a service, it can access the SD and search for available services, ORC service will then get information about the providing services that better satisfy the requirements of the consumer. After a choice is made, the Authorization service will allow communication between the consumer and the provider to start [6], [7].

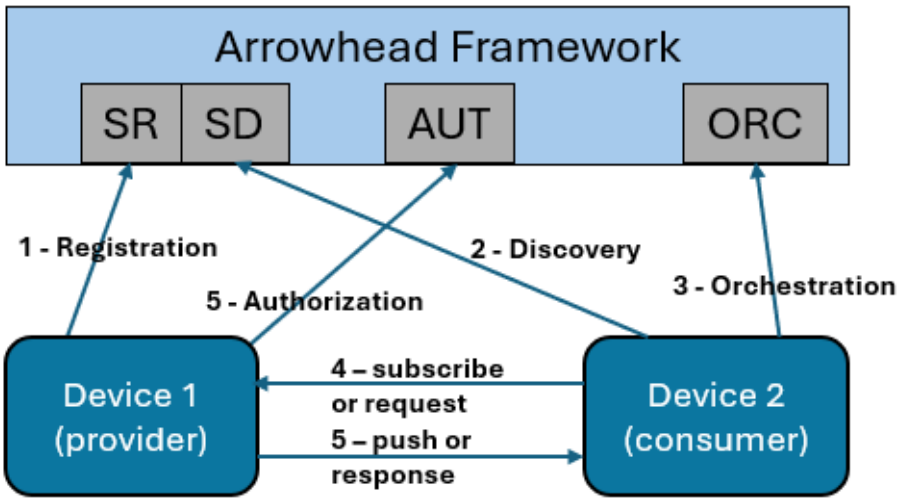


Figure 2.2 - Interactions between providers and consumers using the AF, adapted from [11]

The Arrowhead Framework effectively addresses significant syntactical, semantic, and technical interoperability challenges that arise when devices with different metadata, semantics, and protocols try to communicate [9]. Through the workflow depicted in Figure 2.2, AF makes it easier for systems to interact across these interoperability dimensions [6].

2.3.1 Service-Oriented Architecture

Service-oriented architectures (SOA) were a relevant notion while explaining the Arrowhead Framework. SOA represents a paradigm that organizes a collection of capabilities distributed across networks governed by distinct owners, essentially serving as an alternative for centralized architectures. In SOA, a service, defined as a core building block, represents a software application executing specific tasks [12]. Each service possesses a formal interface articulated using a standard description framework. The Framework's design shields users from many complexities, enabling a dynamic and collaborative environment where services from different domains operate cohesively.

2.3.2 Modern systems adaptation

In the era of advanced technological integration, the heterogeneity factor between systems keeps increasing. Modern systems are built with more thought put into adaptability and interoperability because it is known they will be inserted in heterogeneous domains. Systems are designed with a forward-looking approach towards frameworks like Arrowhead and seamlessly communicate and collaborate with diverse entities. These modern architectures often integrate the core services the frameworks offer, such as robust Service Registries, dynamic Service Discovery, intelligent Orchestration, and secure Authorization mechanisms [13]. This dynamic alignment with the frameworks grants modern systems enhanced flexibility and adaptability.

2.3.3 Legacy systems adaptation

Unlike modern systems, integrating legacy systems into contemporary frameworks represents a distinct challenge but is still essential for interoperability evolution. Legacy systems, rooted in older technologies, often lack compatibility with modern interoperability frameworks like the AF. An older sensor might have more difficulty communicating with a newer consumer system, but it is still necessary for their communication to occur. These contrasts between modern systems embracing frameworks and legacy systems undergoing adaptation are a good example of the challenges and opportunities within the domain of interconnected technologies.

Consider a practical example: A smart city's modern consumer system requires the readings of various sensors for various applications, many of which were placed many years ago, have outdated message types, and lack compatibility with modern frameworks. It is easy to perceive how this could be problematic: interaction will not be possible due to the differences in their protocols. In such cases, legacy systems need to adapt by changing their specifications or using an adapter [6]. More and more adapters would be required between more and more systems, raising the question of how this could be implemented dynamically and with a good scalability factor.

2.3.4 Arrowhead Technology

Arrowhead technology, although still under development, offers a promising framework for the integration and orchestration of systems in the Internet of Things (IoT). These technologies, while not yet fully documented or ready for industrial use, have been explored in this project due to their potential to provide advanced functionality.

Two versions of Arrowhead were utilized in this context: a basic HTTP version and a cloud-based version. The HTTP version allowed communication through the Swagger API, a widely adopted tool for describing and interacting with RESTful APIs. This provided a straightforward method of testing and communication between systems. On the other hand, the cloud version of Arrowhead enabled the integration of Systems-of-Systems (SoS) with minimal friction, a key requirement for IoT environments. The primary references for this work were the Arrowhead Framework GitHub repository [14] and instructional videos provided by "AITIA International Zrt", available on YouTube.

The cloud environment offered a suitable platform for developing Arrowhead-compatible systems and services. It facilitated access to core Arrowhead Framework services through built-in interfaces, simplifying the management and integration of different systems. These systems could either be sourced from existing GitHub examples or newly developed by the user, providing flexibility in design and implementation. The next section delves into the specific technologies and frameworks employed to build these services from the ground up. Important to note that all the services explored and developed were made so using Spring Boot, which will be explored in the next section.

2.4 Spring Boot Services

Spring Boot has emerged as a leading framework for building microservices and stand-alone, production-ready applications with minimal configuration overhead [15]. Due to its flexibility and wide adoption, it has become an ideal choice for the rapid development of service-oriented architectures, such as those required by Arrowhead systems.

One of the key features of Spring Boot is its auto-configuration capability, which significantly reduces the complexity associated with setting up applications. This feature is particularly beneficial in distributed systems like the AF, where multiple services must communicate and interact seamlessly. Spring Boot's auto-configuration dynamically adjusts application components based on the dependencies included in the project, streamlining the process of creating microservices that can operate efficiently within a complex ecosystem like the AF.

In addition, Spring Boot’s embedded server support (such as Tomcat, Jetty, and Undertow) enables the deployment of self-contained applications. This eliminates the need for external servers, simplifying deployment. This embedded architecture also aligns with the independence required in Arrowhead’s System-of-Systems approach.

A thorough exploration of Spring Boot was conducted, focusing on the creation of various standard microservices, an example of a controller for one such service can be seen in Figure 2.3. These initial examples provided the foundational understanding necessary for working within a microservices architecture. Key concepts such as RESTful web services, dependency injection, and service discovery were explored, all of which are integral to building distributed systems like the AF. A Spring Boot beginner certificate was obtained during this process.

```java
@RestController
@RequestMapping("sensor")
public class ProviderController {
    @Autowired
    private InMemorySensorDB sensorService;
    // ------------------------------------------------------------------------------------------
    @GetMapping(path = CommonConstants.ECHO_URI)
    public String echoService() {···
    // ------------------------------------------------------------------------------------------
    @GetMapping("")
    public List<SensorResponseDTO> getAllSensors(@RequestParam(name = SensorProviderConstants.REQUEST_PARAM_BRAND, required = false)

    @GetMapping("/{sensorId}")
    public ResponseEntity<Sensor> getSensorById(@PathVariable Integer sensorId) {···

    @PostMapping("")
    public ResponseEntity<Sensor> addSensor(@RequestBody SensorRequestDTO sensorRequest) {···

    @DeleteMapping("/{sensorId}")
    public ResponseEntity<Void> deleteSensor(@PathVariable Integer sensorId) {···
}
```

Figure 2.3 - Spring Boot controller developed for a provider system

Following this learning phase, the focus shifted to practical use cases, particularly the development of sensor-related services capable of receiving sensor messages in specific formats. These services involved integrating real-world databases, such as PostgreSQL and MySQL, as well as in-memory databases for rapid prototyping and testing. This was a good step in understanding how to make Arrowhead-compatible services, which we'll delve further into in the development chapter. The following Figure 2.4 schematizes the typical Spring Boot application architecture.

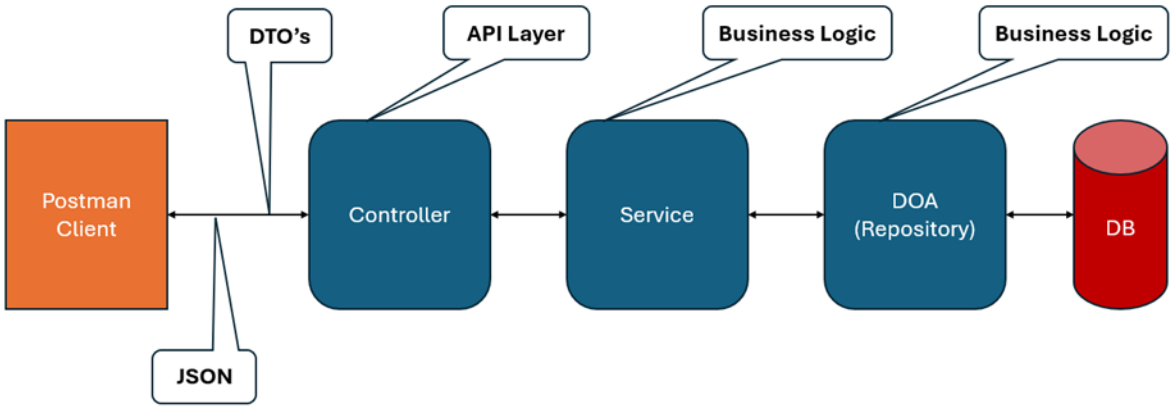


Figure 2.4 - Spring Boot architecture

2.5 Extensible Markup Language

XML [16], or eXtensible Markup Language, is a versatile and widely adopted standard for representing structured data in a human-readable and machine-understandable format. Its core components include tags, elements, attributes, and optional Document Type Definitions (DTDs) or XML Schemas, which can express XML metadata, such as languages like DTDs and XSD (XML Schema Definitions), collectively forming a flexible framework for data organization.

Tags contain elements, creating a hierarchy that gives semantic meaning to the data. Attributes provide additional details, enhancing the richness of information. XML's adaptability makes it invaluable for various applications, from configuration files to web services. In this work, the interest in XML is particularly for its data representation capabilities, such as readings from sensors, like the one exemplified in listing 2.1.

In the example, XML organizes information about two sensors with different locations and reading values. The structure and readability of XML make it a good choice for data interchange in various contexts.

```
 1. <sensorData xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLoca-
tion="u.xsd">
 2.     <temperature>
 3.       <location>Room A</location>
 4.       <value>22.5</value>
 5.     </temperature>
 6.     <pressure>
 7.       <location>Room B</location>
 8.       <value>1013.2</value>
 9.     </pressure>
10. </sensorData>
```

Listing 2.1 - XML document containing information about sensors

Metadata: A simple way to explain the meaning of metadata would be to say it is data about data [17]. It offers context and details about the characteristics, content, quality, and structure of a particular set of data. In the next sub-section, how metadata can be expressed will be clarified.

2.5.1 XML Schemas

Also known as XSD (XML Schema Document) it's main purpose is to define the structure and constraints of XML documents, ensuring data consistency and facilitating validation. By providing a blueprint for expected document elements, their data types, and relationships, XML Schemas can be used to validate the format of an XML document and contribute to a better understanding between systems exchanging information.

In Listing 2.2, the XML Schema defines complex types for temperature and pressure sensors, specifying the expected structure of the data. The subsequent XML document adheres to this schema, ensuring the data conforms to the predefined rules.

```
1.  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
2.
3.    <xs:complexType name="TemperatureSensorType">
4.      <xs:sequence>
5.        <xs:element name="location" type="xs:string"/>
6.        <xs:element name="value" type="xs:float"/>
7.      </xs:sequence>
8.    </xs:complexType>
9.
10.   <xs:complexType name="PressureSensorType">
11.     <xs:sequence>
12.       <xs:element name="location" type="xs:string"/>
13.       <xs:element name="value" type="xs:float"/>
14.     </xs:sequence>
15.   </xs:complexType>
16.
17.   <xs:element name="sensorData">
18.     <xs:complexType>
19.       <xs:sequence>
20.         <xs:element name="temperature" type="TemperatureSensorType"/>
21.         <xs:element name="pressure" type="PressureSensorType"/>
22.       </xs:sequence>
23.     </xs:complexType>
24.   </xs:element>
25. </xs:schema>
```

Listing 2.2 - XML schema for Listing 2.1

2.6 JavaScript Object Notation

JavaScript Object Notation (JSON) is another popular data format for lightweight and human-readable data interchange. Its simplicity and flexibility make it a good choice for various applications, including web services and IoT. Amongst other differences, relative to the XML, it is easy for machines to parse and generate [18]. JSON represents data as key-value pairs, providing an easily interpretable structure. Providers and consumers often use it as their default message format output and input, respectively.

In Listing 2.3, the JSON structure encapsulates data from a temperature sensor. Key-value pairs define attributes such as sensor type, location, recorded value, and timestamp. The simplicity of JSON facilitates efficient data exchange between systems, making it particularly well-suited for scenarios where lightweight communication is crucial.

```
1. {
2.     "sensorType": "Temperature",
3.     "location": "Room A",
4.     "value": 22.5,
5.     "timestamp": "2023-04-10T10:30:00"
6.  }
```

Listing 2.3 - JSON document containing information about a sensor

2.6.1 JSON Schemas

JSON Schemas provide a formalized way to define the structure, data types, and constraints of JSON documents make sure the data has integrity.

In Listing 2.4, the JSON Schema defines the expected structure for data from a temperature sensor. It specifies the data types of attributes, sets constraints (such as an enumerated sensor type), and requires specific fields to be present. The subsequent JSON document adheres to this schema, ensuring that the data conforms to the predefined rules.

```
1. {
2.     "$schema": "http://json-schema.org/draft-07/schema#",
3.     "type": "object",
4.     "properties": {
5.       "sensorType": { "type": "string", "enum": ["Temperature"] },
6.       "location": { "type": "string" },
7.       "value": { "type": "number" },
8.       "timestamp": { "type": "string", "format": "date-time" }
9.     },
```

```
10.     "required": ["sensorType", "location", "value", "timestamp"]}
```

Listing 2.4 - JSON schema containing information about a sensor

2.7 Translation of messages

Consider a scenario where two systems, A and B, need to communicate. If system B cannot understand what system A sends due to differences in syntax or structure, a translation process is required. Translation, in this case, refers to the necessary adjustments made to the message from A so that B can interpret it correctly.

As in this example, translation is often needed to enable dynamic interoperability between heterogeneous systems. In the context of this thesis, translation refers to the modifications necessary to a message sent by a provider in one system, regardless of the message format, to ensure that the receiver can properly interpret it.

In this document, translation is considered an additional step or augmentation that may be required when communication between systems is still not successful after utilizing the core services of a common framework like the Arrowhead Framework. As illustrated in Figure 2.5, translation is typically required when systems exchange messages with differing structures or semantics, whether in XML, JSON, or other message formats. The translation process can involve converting messages from one format to another to ensure that systems employing different data representations can communicate effectively. It is especially important when legacy systems are involved, as it enhances collaboration and integration across diverse technological ecosystems.

The TAG Tool [6], [7], mentioned in the introduction, is an augmentation tool designed to address these translation challenges. It can be added as an additional layer on top of the

Arrowhead Framework to enable message translation between systems that remain non-interoperable even after using AF services.

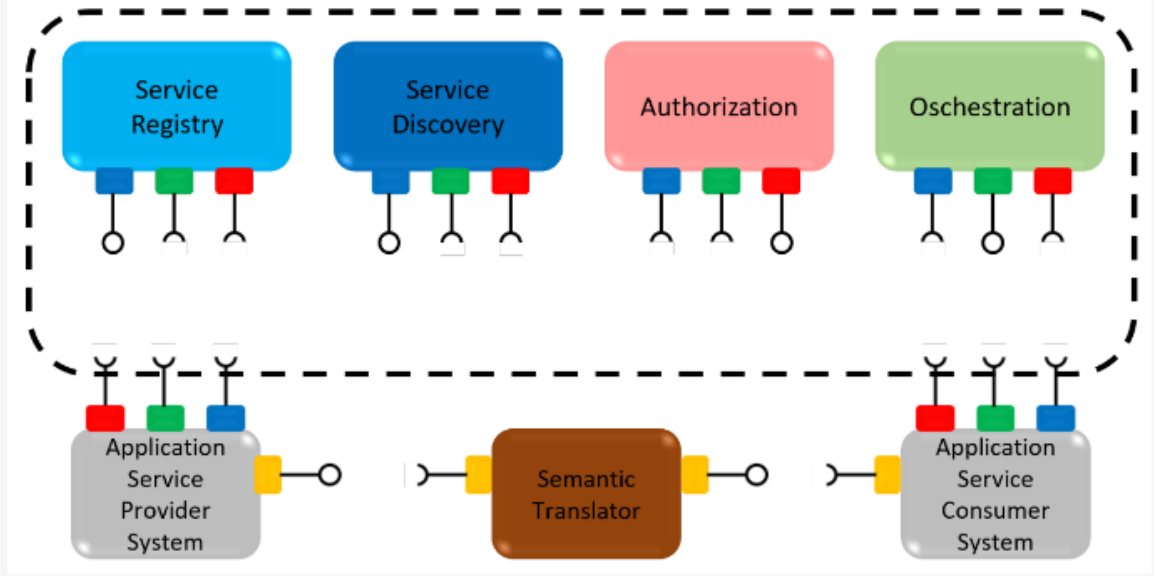


Figure 2.5 - Translator integrated to aid the Arrowhead Framework, taken from [7]

2.7.1 Popular Translation methods and tools

Translation is a crucial step in achieving interoperability. There are various modern methods used alongside specific tools with the sole purpose of facilitating the translation of information between systems. In this sub-section, some examples of common methods for achieving this purpose will be provided:

Adaptive Middleware: This method uses adaptive middleware modules that can dynamically interpret and convert messages between different formats. These middleware solutions often employ configurable rules and mappings to facilitate the translation process. [19]

Data Transformation Languages: Translation methods may use data transformation languages such as XSLT (eXtensible Stylesheet Language Transformations) or JSONPath for XML and JSON formats, respectively. These languages provide declarative ways to define transformations between disparate data structures.

Semantic Mapping: Another approach involves semantic mapping, where the meaning and intent of data elements are defined independently of their syntax with the support of ontologies. This method helps overcome differences in message formats by focusing on the information's semantics. This topic will be further explained in the following sections.

2.7.2 Translation platforms

While the platforms discussed in this section are not directly employed in this work, they serve as valuable knowledge regarding state-of-the-art platforms utilized for data translation. These platforms offer insight into the advancements and functionalities available in the field of translation.

Apache Camel: Provides a comprehensive framework for integrating different systems by supporting a wide array of data formats and protocols. It supports 20 different languages including XML and JSON and includes components for data transformation, enabling seamless translation between diverse message structures [20].

IBM Transformation Extender: This platform offers a graphical mapping tool that allows users to define translation rules and transformations visually. It supports various message formats and is designed for complex integration scenarios [21].

MuleSoft Anypoint Platform: Its main tools are for designing, building, and managing APIs and integrations. But like the other two platforms, it also includes data mapping and transformation capabilities to ensure smooth communication between systems with different message formats. It is a good example of how important it is for platforms to have translating capabilities, even if it isn't their primary goal [22].

2.8 Ontology fundamental concepts

An Ontology is a general logical theory constituted by a vocabulary [23]. In the context of information science, it refers to a formal representation of knowledge that defines the concepts within a domain and the relationships between them. It serves as a structured framework for organizing information in a way that various machines can understand. The importance of

ontology in the system communication domain lies in its ability to give semantic meaning to data, looking for more meaningful and context-aware interactions between systems.

In Figure 2.6, we can see the appearance of an ontology built-in Protégé, an ontology building software. The exemplified ontology gives meaning to sensor data.

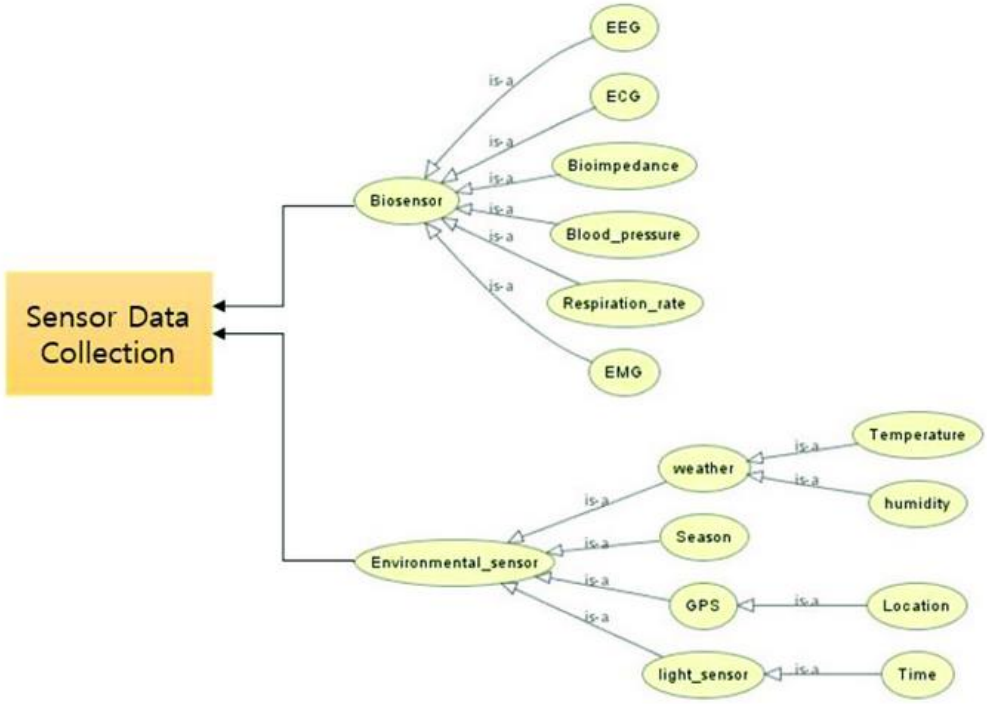


Figure 2.6 - Ontology model for sensor network constructed in an ontology building software, from [24]

At its core, ontology provides a shared vocabulary and a set of rules that enable machines to infer relationships, reason about concepts, and make informed decisions. By defining classes, properties, and instances, ontology establishes a common understanding of a domain, allowing machines to communicate not just based on syntax, but on the semantics of the information exchanged too.

In practice, ontologies are used as structured semantic models that refer to a specific vocabulary because good developmental practices recommend the reuse of existing vocabularies for interoperability purposes [25]. Labelling data with a reference ontology is an example of a way to let systems know what that data is referring to, even when their data syntax is completely different. This method is of extreme importance as it is how we approach the use of ontologies in this work, which we will delve into semantic annotations and their use in the TAG Tool is explained.

Semantic Web: Inseparable from ontologies, the Semantic Web (Figure 2.7) plays a main role in enhancing the meaning of information. It aims to allow computers to extract meaning from exchanged information by interpreting how data's connections are linked to other definitions. These links provide data with an ontological significance, leading to a clearer interpretation and comprehension. As Tim Berners-Lee, the inventor of the World Wide Web, aptly put it: "The Semantic Web is not a separate Web but an extension of the current one, enabling information to be given well-defined meaning. This extension better equips both computers and people to work in cooperation" [23], steering towards a new era of more intelligent and context-aware interactions on the web.

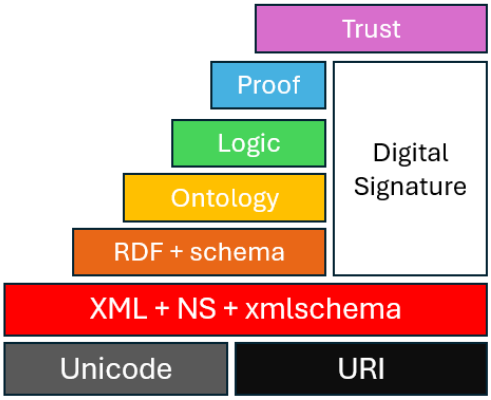


Figure 2.7 - Semantic Web, representative layers adapted from[23]

2.8.1 Resource Description Framework

Usually referred to as RDF, it is a foundation in the realm of ontologies. It serves as a standardized format for representing language for the Semantic Web, allowing the creation of structured, machine-readable data. RDF's role is vital in establishing connections within ontologies, enabling systems to precisely model relationships between entities and attributes.

A fundamental unit of knowledge in the Resource Description Framework realm is the RDF Triples concept. These serve as the elemental building blocks for representing knowledge in a semantic web environment. An RDF Triple is composed of three interconnected nodes, each assuming fixed distinct roles: Subject, Predicate, and Object [23].

Subject: The Subject represents the resource or entity about which information is being asserted. It is the starting point of the triple, the primary focus of the statement. To be identified properly, the use of a Uniform Resource Identifier (URI) is common.

Predicate: The Predicate signifies the property or attribute associated with the Subject. It denotes the nature of the relationship between the Subject and the Object. It is also identified by an URI.

Object: The Object designates the actual value or target of the property specified by the Predicate. It embodies the actual data, representing either a literal value or another resource identified by a URI. The Object encapsulates the information, or the object of description associated with the Subject.

Represented in Figure 2.8 we can see a data graph model representing an RDF triple.

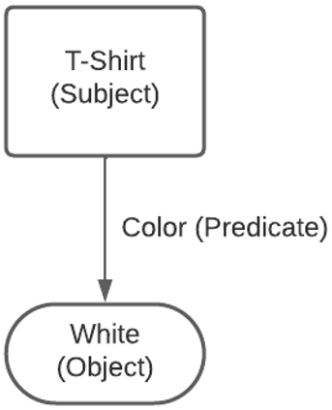


Figure 2.8 - RDF triple

Let's look at the next Listing 2.5, as a simple RDF/XML statement, the standard syntax for serializing RDF graphs for storage or transmission [26]. In line 5 the definition of the subject about which a description will be provided is made, following that, in line 6 both the predicate (feature size) is declared, and its literal value (12) is expressed.

```
1. <?xml version="1.0" encoding="UTF-8"?>
2.  <rdf:RDF
3.   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4.   xmlns:feature="http://www.linkeddatatools.com/clothing-features#">
5.   <rdf:Description rdf:about="http://www.linkeddatatools.com/clothes#t-shirt">
6.       <feature:size>12</feature:size>
7.   </rdf:Description>
8. </rdf:RDF>
```

Listing 2.5 - RDF document representing a T-shirt

Graph data model: The model used for storing data in the semantic web consists of relations between entities, with no regard for their order or hierarchy. Figure 2.8 is an example of a simple graph data model.

SPARQL: For RDF data to be queried the use of a SQL-like query language arises, SPARQL, which stands for, SPARQL Protocol and RDF Query Language [27], is exactly what was created to fix this issue. Listing 2.6 provides a short example to illustrate how SPARQL queries are used. If executed on the Portuguese government's Open Semantic Database, this example will return the names of all the colleges in the Portuguese district of Setúbal and order the results in alphabetical order.

```
1. PREFIX pt-ont: <http:// education.data.gov.pt/def/college>
2. SELECT ?collegeName
3. WHERE {
4.    ?college a pt-ont:College.
5.    ?college pt-ont:collegeName ?collegeName.
6.    ?college pt-ont:districtAdministrative <http://statistics.data.gov.pt/district/Setubal>.
7. }
8. ORDER BY ?collegeName
```

Listing 2.6 - SPARQL query

2.8.2 Web Ontology Language

RDF is the foundation of defining data structures for the semantic web, but by itself it does not describe the meaning behind the data, choosing other words, it doesn't describe the semantics. Web Ontology Language [28], known as OWL, is a formal language designed to represent knowledge about entities, their properties, and relationships within a specific domain and it is utilized to provide RDF data models with semantics. Semantic metadata can be applied to RDF through the application of OWL. This language is made for applications that need to process the content of information instead of presenting information to humans, which is the case for many web applications.

While RDF structures the data, OWL adds a layer of semantic meaning. The inclusion of OWL in RDF documents allows machines to understand the significance of concepts and reason about relationships, leading to more intelligent and context-aware interpretations.

Looking at the example in Listing 2.7, the RDF document represents a sensor ontology, where a non-mandatory header was added for clarification purposes about what the ontology is for (in this case it defines sensors), followed by a class definition for a specific sensor type.

```
<rdf:RDF>
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl=http://www.w3.org/2002/07/owl#
    xmlns:dc="http://purl.org/dc/elements/1.1/">
    <!-- OWL Header Example -->
    <owl:Ontology rdf:about="http://www.linkeddatatools.com/sensors">
    <dc:title>The LinkedDataTools.com Example Sensor Ontology</dc:title>
    <dc:description>An example ontology for LinkedDataTools.com, focusing on sen-
sors.</dc:description>
    </owl:Ontology>
    <!-- OWL Class Definition Example -->
    <owl:Class rdf:about="http://www.linkeddatatools.com/sensors#SensorType">
    <rdfs:label>The sensor type</rdfs:label>
    <rdfs:comment>The class of sensor type.</rdfs:comment>
    </owl:Class>
</rdf:RDF>

```

Listing 2.7 - RDF ontology defining document

2.9 Annotations

Annotating, in a broader sense, simply means to attach extra data to another piece of data [29]. In the context of data representation, annotations serve as data that adds context and descriptive information to elements within a document.

They provide the means to convey additional details about the data, such as its purpose, origin, or intended usage, offering a richer understanding of the data's meaning. This enhances the data's interpretability by machines within the semantic web. The additional data that was annotated could be done so in various ways, manually and semi or fully automatically.

2.9.1 Semantic Annotations for WSDL

The essence of semantic annotations lies in their ability to ensure semantic interoperability, making sure that the information exchanged between systems holds the same meaning for both senders and receivers. This is, once again, particularly significant in heterogeneous environments where diverse systems with varying perspectives need to communicate.

A key characteristic of semantic annotations is their dual readability – both human-readable and machine-readable. Additionally, they respect a set of formal and shared terms, often drawn from ontologies, which are widely accepted and understood within a specific domain [11], [29]. This ensures the clarity and consistency of the annotated information in different contexts.

There are multiple mechanisms to semantically annotate data or metadata in documents. In the context of this work, the SAWSDL model reference that allows the annotation of each element of an XSD with concepts of a semantic model is highly relevant [11]. It may be used to generate translators between devices that wouldn't be able to communicate otherwise, alongside other methods that will be later explained.

In Listing 2.8, SAWSDL [30]model references are added as attributes to XML elements. These references link specific elements to corresponding semantic models or ontologies. The model references, such as #BookTitle, #BookAuthor, #PublicationYear, and #BookDescription, indicate the semantic annotations associated with each element.

```
<library>
  <book>
    <title>Introduction to Semantics</title>
    <author>John Doe</author>
    <year>2022</year>
  </book>
  <book>
    <title>Data Modeling with XML</title>
    <author>Jane Smith</author>
    <year>2021</year>
  </book>
</library>
<library xmlns:sawsdl="http://www.w3.org/ns/sawsdl">
  <book sawsdl:modelReference="#BookDescription">
    <title sawsdl:modelReference="#BookTitle">Introduction to Semantics</title>
    <author sawsdl:modelReference="#BookAuthor">John Doe</author>
    <year sawsdl:modelReference="#PublicationYear">2022</year>
  </book>
  <book sawsdl:modelReference="#BookDescription">
    <title sawsdl:modelReference="#BookTitle">Data Modeling with XML</title>
    <author sawsdl:modelReference="#BookAuthor">Jane Smith</author>
    <year sawsdl:modelReference="#PublicationYear">2021</year>
  </book>
</library>
```

Listing 2.8 - Annotated XML document

2.9.2 Annotation Paths

Path-based annotations, or Annotation paths [29], allow a targeted method for associating data with specific paths or elements within a document. This approach provides the means for further enriching data with contextual information. Annotation paths can be directly applied to elements or along paths in the document structure, offering a high degree of precision.

Standard SAWSDL model references are limited by their ability to point only to named concepts of an ontology. The direct annotation of documents or schemas with general reference ontologies faces challenges in being precise. To address this limitation, the annotation paths method was introduced, specifically for schemas, which is what this work aims to

annotate, further insights into why will be provided in the explanation of how the TAG Tool operates. Annotation paths serve as explicit paths through a reference ontology, consisting of concepts and properties. The string representation of these paths can be used as SAWSDL model references, as illustrated in Listing 2.9, paired with "modelReference" (lines 10, 14, and 18). The message schemas used in the development part, often use SAWSDL-referenced annotation paths.

```
 1. {
 2.     "$schema": "http://json-schema.org/draft-07/schema",
 3.     "type": "object",
 4.     "properties": {
 5.       "dataValues": {
 6.         "type": "object",
 7.         "properties": {
 8.           "units": {
 9.             "type": "string",
10.             "modelReference": "/LumenSensor/hasLumenUnits/LumenUnits"
11.           },
12.           "sensor1lumen": {
13.             "type": "number",
14.             "modelReference": "/LumenSensor/hasDecValue"
15.           },
16.           "sensor2lumen": {
17.             "type": "number",
18.             "modelReference": "/LumenSensor/hasDecValue"
19.           }
20.         }
21.       }
22.     }
23. }
```

Listing 2.9 - Semantically annotated JSON Schema using annotation paths

2.10 The TAG Tool

The TAG Tool [6], [7] was developed with the aim to augment the interoperability capabilities between heterogeneous systems. As has been mentioned, legacy systems have a hard time adapting to the current frameworks since they were not built according to their specifications. To adapt legacy systems to modern frameworks such as the AF, it is necessary to either change their specifications or create extra tools such as adapters to fix their incompatibilities [6]. TAG Tool aims to allow the interoperability between poorly adapted systems that are using the AF's core services yet fail to establish communication due to different message requisites. This is where the tool comes in and provides a translator between the two systems' messaging path (the so-called adapter).

This prototype was specifically designed to adapt messages sent between consumers and providers in either XML or JSON format. The consumer and provider must provide the Tool with their message schemas (XSD or JSON Schema files). An OWL file must also be available for the Tool to serve as the domain ontology. After executions when both annotated XSDs are provided, the Tool will check if the appropriate mandatory provider's XSD elements the consumer expects exist. If no errors arise in this comparison step, the Tool will generate an XSLT file that will transform the provider XML to an XML compatible with the consumer's expected format.

The Tool's ability to translate other message formats like JSON, is due to the extensions that were added after the translation core for XML was already functional. Said extensions were proposed in [7] and based on the work done in [6], by providing a way to annotate JSON Schemas, and later translating them to XML Schemas. It is important to note that the translation of provider messages is always made with the XSD files (from the consumer and provider), so any other data format, like JSON Schemas, needs to be translated to XSD before the message translation core service can take place.

In the scenario illustrated in Figure 2.9. The provider sends a JSON message, and the consumer is expecting a message in JSON format. After being provided with the JSON message schemas from both systems, the Tool translates them to XML Schemas, makes the translation of the message to XML, transforms it according to the needs of the annotated consumer schema (if possible), translates it back to JSON, and finally the message is in a JSON format the consumer can understand. This is a great example of the tool's ability to adapt to different message formats, as long as the systems aiming to communicate provide their respective message schemas.

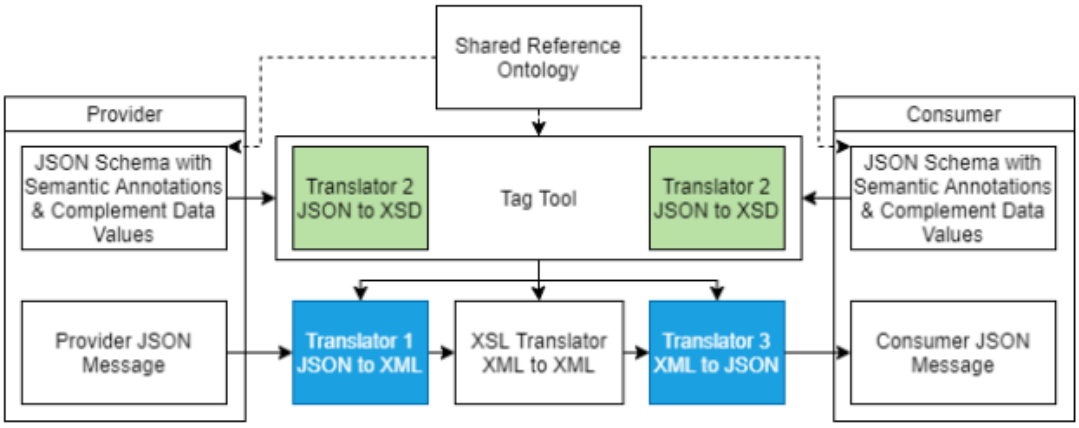


Figure 2.9 - Extended Architecture of the TAG Tool, taken from [7]

2.11 Java version of TAG Tool

The Java version of TAG Tool is an evolved version of the earlier TAG Tool and serves as the foundation for the development work in this thesis. It builds upon the principles of its predecessor for handling interoperability between provider and consumer systems in service-oriented architectures.

The new version of TAG tool operates in two distinct stages:

1. **Compatibility Report and Translator Generation**: In the first stage, the tool is provided with the provider and consumer schemas, along with a common ontology (typically an OWL file) shared between the two systems. Based on this input, it generates a compatibility report that evaluates whether the provider-consumer pair can potentially communicate. If the report deems the pair compatible, the tool proceeds to generate a Java translator.
2. **Message Translation**: In the second stage, the translator generated in the first step is used to perform the actual translation of messages. This translator enables message conversion between the provider and consumer systems, ensuring that messages conform to the correct format (such as XML or JSON) required by the consuming system.

Unlike the previous version, which relied on XSLT for transforming messages, the new tool generates a Java translator to handle the translation process. This shift to Java allows for more flexibility in the translation logic and enables more complex transformations. Once the

translator class is generated, it can be executed to produce a translated message that is compatible with the consuming system.

The Tool will be used in development, to make systems with different message formats previously unable to communicate, interoperable, making it one of the crucial steps in the DTM platform. From this point on, in this document, every time the TAG Tool is referred, it will be referred to this Java version. It can also be referred to as just TAG.

3
PROPOSED SYSTEM

In this chapter, a general overview of the proposed system will be presented, along with insights into the development steps taken throughout its implementation. The focus will be on the core elements of the environment and the methods used to integrate the Data Translation Manager with the Arrowhead Framework and TAG Tool, demonstrating the progression from conceptualization to functionality.

3.1 Introduction

The TAG serves as a crucial tool in enabling interoperability between previously incompatible systems by translating messages between them. As discussed earlier, when a consumer requires a specific message format that a provider cannot deliver, the TAG functions as an intermediary translation tool, increasing the likelihood of successful communication. However, the TAG Tool currently operates as an isolated node that requires external inputs, such as message schemas and ontologies, to function. For real-world applications, an intermediary system is necessary to supply these inputs and execute the TAG Tool. Ideally, this intermediary would gather information from systems and services within the Arrowhead framework and pass it along to the TAG for execution. The **Data Translation Manager (DTM)** fulfils this intermediary role.

When the TAG is integrated with the DTM within the Arrowhead platform, another question arises: when multiple provider systems offer the same service, how does the Arrowhead framework determine the most appropriate provider? The framework does not inherently choose the most suitable provider; instead, it selects one without considering some important characteristics. In scenarios with only two providers for a selected service, there is a 50% chance of selecting the best one. As the number of providers increases, the likelihood of choosing the optimal one diminishes.

This limitation underscores the need for a systematic approach to selecting the most appropriate provider based on specific criteria. Implementing a method to evaluate and choose the best provider according to their metadata could significantly enhance system interoperability. The TAG Tool generates reports detailing the translation requirements between a consumer and provider, which are essential for analysing how well a given provider meets the consumer's needs.

The main objectives of this project are to manage the TAG and address the challenge of identifying the best provider. The core goal is to develop an Arrowhead-compatible system that includes a consumer capable of retrieving relevant data from all available providers offering a particular service and then selecting the most suitable provider. By analysing the data gathered from providers, the system can assess the ontology they use, determine if it aligns with the consumer's ontology, and evaluate their metadata. If both the provider and consumer are already integrated into the Arrowhead ecosystem and can communicate without translation, then the TAG Tool is not required. However, as previously mentioned, if the consumer cannot communicate with the provider due to differing message formats, the TAG must intervene.

In situations where multiple providers offer a service that the consumer requires, it is crucial to have a mechanism for determining which provider is best suited to fulfil the consumer's service needs. This necessity led to the development of the Data Translation Manager, which plays a vital role in this process.

To begin the project, foundational knowledge of Spring Boot services was acquired, as expertise in microservices was deemed crucial for developing Arrowhead-related services and systems. This was followed by an exploration of existing Arrowhead tools and examples, which, despite limited documentation, provided valuable "hands-on" experience. A complex consumer system was then developed, serving as the foundation upon which the Data Translation Manager will work with.

Ultimately, besides bridging the gap between the TAG and the AF, the goal of this project is to create an Arrowhead compatible system that can retrieve relevant data from all providers and select the most suitable one based on various criteria, enhancing the overall interoperability and efficiency of the SoS in the AF, wanting to exchange data.

3.2 Architecture

In this architecture, Figure 3.1, the providers register themselves automatically upon execution, registering all their relevant metadata fields to the Arrowhead Cloud. The metadata includes essential fields such as the provider's URI, the services offered, and any required details about its capabilities. The Arrowhead system saves this metadata, making it available for consumers who request provider service metadata. The orchestration service facilitates this by granting the consumer access to the list of available providers, which includes all the relevant metadata fields for evaluation.

Once the consumer has the necessary provider information, it proceeds to generate two essential information pieces: one containing the consumer's own metadata, and another containing the provider's metadata. These data pieces are then placed into the DTM node for further processing.

This information files will be then placed on the DTM, where they will be filtered into XSD schema files and inputted to the TAG tool, which will be executed to return reports and translators for each provider present.

The TAG reports are fed back into the DTM and saved in a database, along with any required Translator classes. If no translation is necessary for a particular provider, the corresponding entry in the report will remain empty. The DTM then evaluates all providers based on the reports, using the Provider Analyser script (Listing 3.8), selecting the one that best matches the consumer's requirements, not only considering the minimum elements but also additional optional capabilities that could enhance the system's overall performance. The selected provider's Translator class is then sent to the Translation node (the part of the DTM responsible for executing the Translator class), at the same time as the selected provider service URI is sent back to the consumer.

With the provider's URI in hand, the consumer makes GET requests to the selected provider service, initiating communication. If the provider's message format differs from what the consumer can handle, as expected in this scenario, the consumer will send the received message to the Translation Node, which is a sub part of the DTM. At the Translation Node, the message will be translated, if necessary, from JSON to XML, and then using the Translator Class previously generated by the TAG Tool, this part is handled by the Provider Translation script. Once the message has been translated, it is sent back to the consumer, allowing the consumer to process it.

The DTM also continually monitors the availability of providers. During each execution cycle, after a determined amount of GET requests made by the consumer, the DTM checks for the provider's availability and re-evaluates if necessary to ensure optimal service selection. This monitoring ensures that the system dynamically adapts to any changes in provider availability.

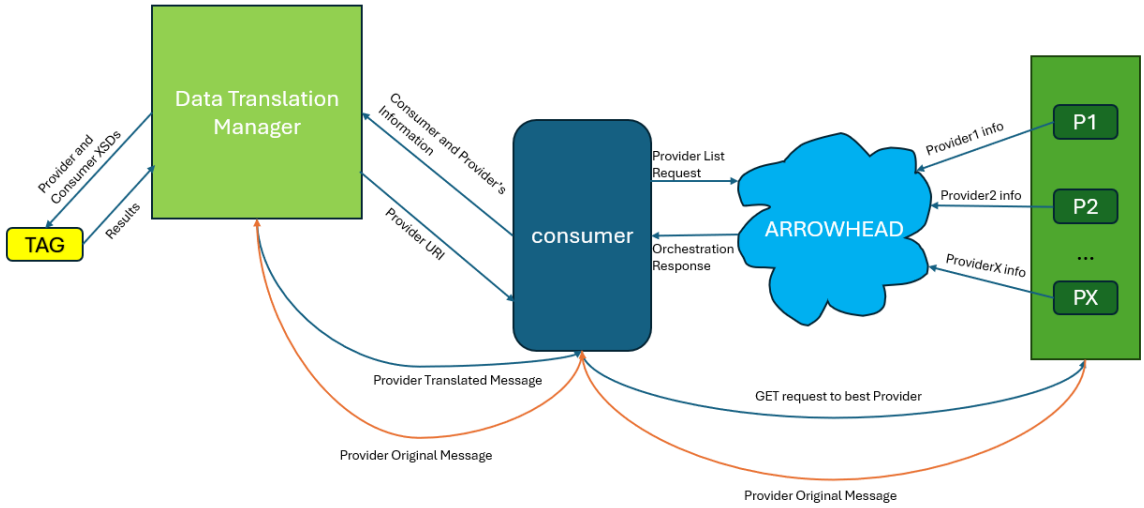


Figure 3.1 - DTM interactions with other components of the system

A sequence diagram, shown in Figure 3.2, demonstrates the workflow of the Data Translation Manager (DTM) system, starting with the providers on the far right and ending with the translated message being returned to the consumer.

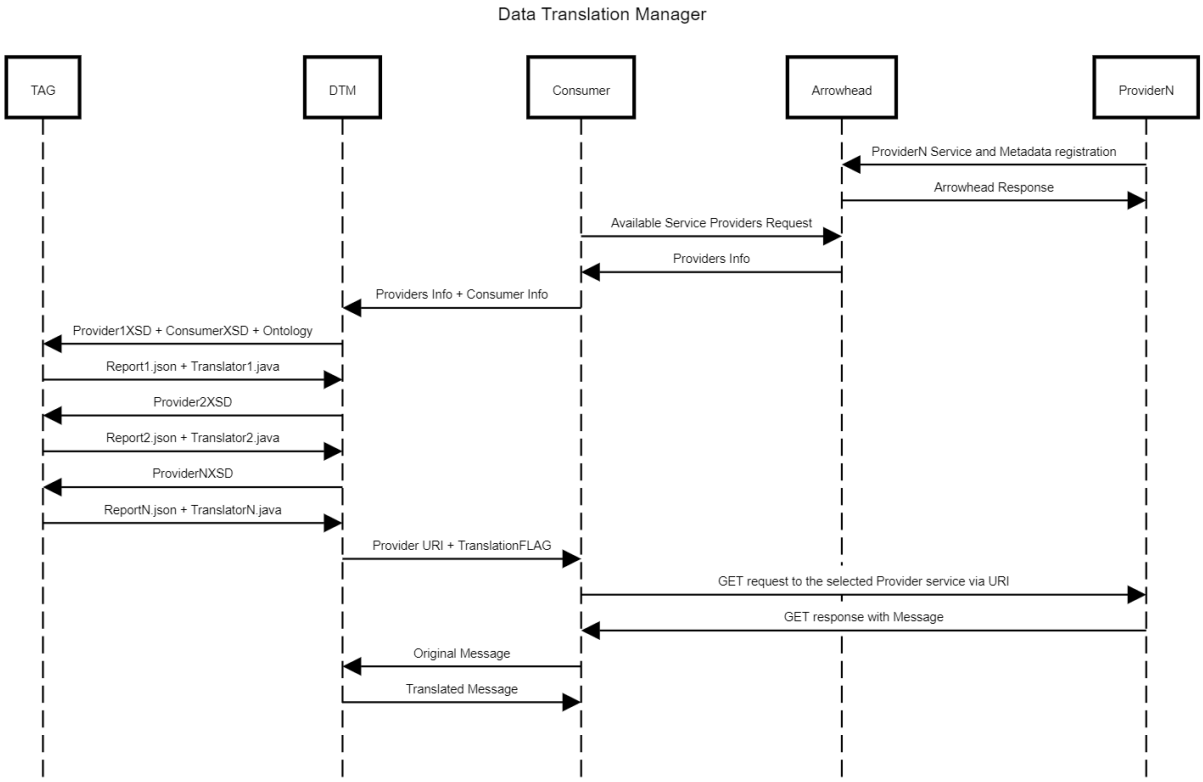


Figure 3.2 - System sequence diagram for the typical scenario

In some cases, a provider with identical metadata to the consumer might be detected. If the metadata fully matches, all the Consumer requirements are satisfied and there is no need for further translation. The consumer can directly communicate with the provider, and the DTM bypasses the TAG Tool and its Translation process entirely. The provider's URI is simply sent to the consumer, establishing a communication channel for direct interaction. This scenario, represented in Figure 3.3, is beneficial because it eliminates a number of steps, simplifying the workflow and improving efficiency.

Just as in the previous typical scenario where there is a need for translation. The DTM will continue to look for the providers' availability.

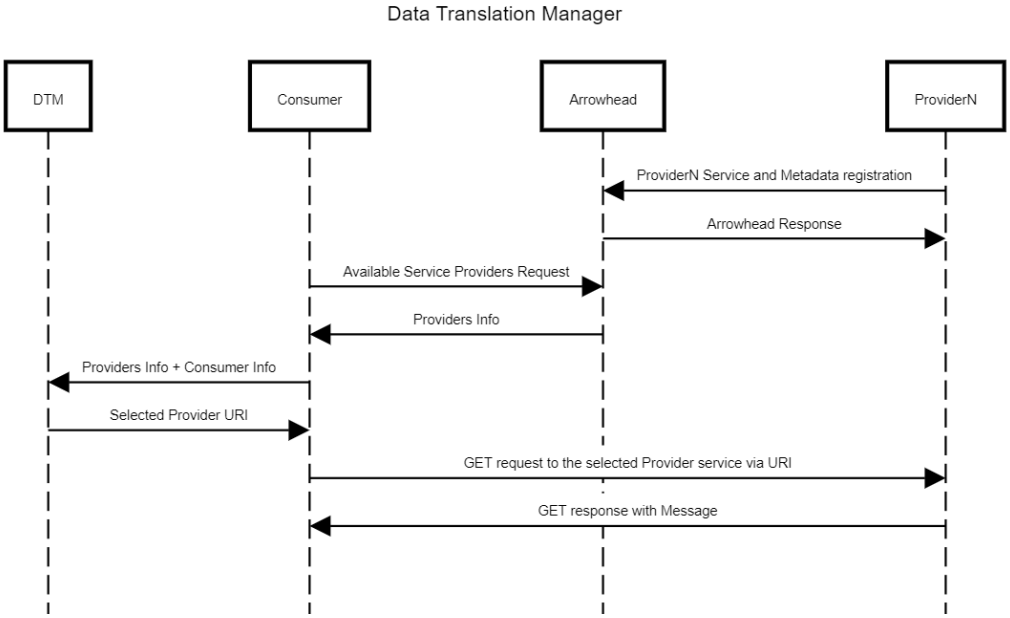


Figure 3.3 - System sequence diagram when there is data compatibility

3.3 Use case Scenarios

The Data Translation Manager (DTM) is the central component of this work. It is responsible for managing the TAG executions and provider storage, as well as for evaluating and selecting the most appropriate provider services based on pre-defined criteria and metadata gathered from the Arrowhead, after which it will enable a connection between the Consumer and the selected Provider. By processing and filtering the available Providers using the TAG, the DTM ensures that the consumer application interacts with the best-suited provider services in a dynamic, scalable microservices environment. This chapter will explore several key use case scenarios that illustrate how the DTM can be leveraged in different contexts.

3.3.1 Use Case 1: Energy-Efficient Smart City Lighting System

A smart city project is looking for an energy-efficient solution to manage its street lighting system. The consumer system, responsible for dynamically adjusting streetlight brightness based on traffic and ambient light conditions, requires sensor data to make informed decisions.

Consumer Metadata Requirements:

- **Minimum Needed Elements:** Traffic sensor data and ambient light sensor data (minimum 1 of each)
- **Optional Elements:** Weather sensor data (to adjust lighting during fog or rain) and pedestrian movement sensors (to adjust lighting for pedestrians).

Provider Capabilities:

1. **Provider A:** Offers a traffic sensor and an ambient light sensor.
2. **Provider B:** Offers one traffic sensor, an ambient light sensor, a pedestrian movement sensor and a weather sensor.
3. **Provider C:** Offers a traffic sensor, an ambient light sensor, and a weather sensor.

Process:

1. **Metadata Extraction:** The DTM analyses each provider services' metadata to evaluate sensor offerings.
2. **Evaluation:**
    - Provider A meets the bare minimum but lacks any optional elements (providing two elements).
    - Provider B not only meets the minimum, but also offers an additional traffic sensor (potentially more accurate data) and a pedestrian movement sensor (providing four elements).
    - Provider C meets the minimum and offers weather data, which could be useful during foggy or rainy conditions (providing three elements).

Selection: The DTM evaluates the extra sensors provided. While Provider A fulfills the basic requirements, Provider B offers pedestrian data, while Provider C offers weather data. In this case, Provider B is chosen because it provides the most comprehensive data set, providing four elements (two optional and two mandatory) versus the three elements provided by Provider service C.

3.3.2 Use Case 2: Smart Farming System for Crop Monitoring

A smart farming system needs to integrate with a provider that can supply detailed environmental data input for a precision agriculture controller. The controlling system is used to monitor soil conditions, weather, and plant growth to optimize irrigation and fertilization schedules.

Consumer Metadata Requirements:

- **Minimum Needed Elements:** Soil moisture sensor data, temperature sensor data, Soil pH sensor data (at least one sensor for each).
- **Optional Elements:** Air humidity data (one sensor), and crop health monitoring (via two multispectral sensors).

Provider Capabilities:

1. **Provider X:** Offers a soil moisture sensor, a temperature sensor, and a soil pH sensor.
2. **Provider Y:** Offers a soil moisture sensor, a temperature sensor, a soil pH sensor, and real-time crop health monitoring.
3. **Provider Z:** Offers two soil moisture sensors, two temperature sensors, and an air humidity sensor but no single crop health monitoring sensor.

Process:

1. **Metadata Extraction:** The DTM retrieves the environmental sensor data provided by each potential provider service.
2. **Evaluation:**
    - Provider X meets only the basic requirements (providing three elements).
    - Provider Y offers advanced crop health monitoring along with all the necessary data, making it highly suitable for this precision agriculture controller (providing four elements).
    - Provider Z provides comprehensive environmental data but does not meet the minimum requirements for a possible translation by the DTM. It is providing the most optional elements, but it is missing the mandatory single temperature sensor (providing five elements).

Selection: The DTM determines that both Provider Y and Provider X respect the minimum requirements, but Provider Y has the extra ability to monitor crop health counting as an extra optional requirement. While Provider Z offers detailed environmental data It fails to meet the minimum requirements needs for the consuming system. Provider Y is, therefore, the optimal choice.

3.3.3 Use Case 3: Healthcare IoT System for Remote Patient Monitoring

A healthcare provider is looking for a service to monitor patients remotely, focusing on vital signs such as heart rate, blood oxygen levels, and temperature. These measurements are essential for managing patients with chronic conditions.

Consumer Metadata Requirements:

- **Minimum Needed Elements:** One heart rate and two blood oxygen level sensors.
- **Optional Elements:** Two body temperature sensors, two blood pressure sensors, one extra heart rate and one extra blood oxygen level sensor.

Provider Capabilities:

1. **Provider A:** Provides three heart rate and two blood oxygen level sensors.
2. **Provider B:** Provides one heart rate, one blood oxygen level, and one temperature sensor.
3. **Provider C:** Provides two heart rates, two blood oxygen levels, and two blood pressure sensors.

Process:

1. **Metadata Extraction:** The DTM extracts data on the vital signs that each provider can monitor.
2. **Evaluation:**
    - Provider A meets the minimum requirements and provides an extra blood level sensor (providing five elements as the third temperature sensor is not counted).
    - Provider B provides extra temperature monitoring but fails the minimum requirements. Not allowing translation.
    - Provider C meets the minimum requirements, with two heart rate sensors, two blood oxygen and one blood pressure sensor (providing five elements).

Selection: In this case both Providers A and C could be selected due to respecting the system's minimal needs while providing one extra optional one. The choice will come down to the Provider that is first on the Providers list. In this case let's assume A comes first, there for it will be the one selected to communicate with the Consuming system from this healthcare provider.

3.4 Development

The relevant development steps will be detailed in the following sub-sections, providing the reader with an insight into how the DTM was made possible from the get-go.

3.4.1 Arrowhead Compatible Systems

The development of Arrowhead services began with an exploration of the Arrowhead system of systems (SoS) examples, which were obtained from the official repository on GitHub [14]. The Demo Car project that is available in the mentioned repository served as the initial reference point for this investigation. The project included three main components: a car consumer, a car provider, and a shared common folder. The first step in the learning process was to modify the demo so that the car consumer and provider could operate independently, without relying on the common folder. This modification allowed these services to be deployed on separate machines while maintaining the ability to communicate with each other.

Both the car consumer and provider were developed using Spring Boot, with each service providing RESTful APIs that supported standard CRUD operations. Through this separation, the car demo systems were designed to communicate via HTTP POST and GET requests, which were only successful once appropriate authorization rules were established within the

Arrowhead Management Cloud Tool. These rules, known as intracloud rules, were necessary to secure the secure exchange of data between the consumer and provider within the same cloud. A critical point in this configuration was that the provider system automatically registered itself with the Arrowhead cloud upon execution.

The car demo SoS was a starting point from where a sensor demo could be based, just by adapting the old car demo, for both the consumer and the provider. The fundamental idea remained the same: to separate the consumer and provider systems, register them with the Service Registry, and create Authorization rules for the services provided between them. For this demo, two core services were developed: "createSensor()" (a POST request to simulate sensor data creation) and "getSensor()" (a GET request to retrieve sensor data). These services were mirrored from the car demo, with the consumer calling the provider's services at runtime, assuming the provider had been previously launched and its services were active.

Despite the successful adaptation of the car demo into a sensor demo, it became evident that this approach—reusing a demo as a foundation for a new system—was not the most efficient or clean method of development. Many cars demo's specific parts would be unnecessary for the new systems to create. Efforts to create a functional, minimal Arrowhead Spring Boot skeleton from scratch were made, but challenges persisted as removing one problem often led to the emergence of another. Fortunately, an Arrowhead Spring Skeleton was available online, and with the foundational understanding gained from the earlier demo projects, this skeleton was identified as a more appropriate starting point. Leveraging the existing skeleton, new testing consumers and providers were created, including the main components used in the final project.

Having established that a single consumer could interact with a single provider (as demonstrated by both the car, sensor demos and the previously developed skeleton consumer and providers), the next logical step was to test whether a consumer could interact with multiple providers offering the same service. The objective was to determine if the Arrowhead Framework could support a scenario where a consumer could dynamically select between multiple providers based on the services offered.

Although multiple providers were successfully registered and their corresponding intracloud rules were created, a limitation was observed. The Arrowhead Orchestrator handled the consumer's service request by allowing only one service to be consumed from each provider. For instance, when two providers offered both getSensor() and createSensor() services, the consumer could only consume getSensor() from one provider and createSensor() from the other. This behaviour meant the consumer could not consume both services from the same

provider or both providers, as initially expected. The orchestrator enforced a rule where service consumption was limited to one service per provider.

This behaviour highlighted the need for an additional mechanism to select specific providers based on the consumer's requirements. The inability to consume all available services from multiple providers led to the necessity of a Data Translation Manager to identify and prioritize which provider's services would best meet the consumer's needs in a multi-provider environment.

3.4.2 Extended Arrowhead Consumer

As highlighted before, there is a necessity to know which provider's services to consume, it is proposed in this project, that this choice is to be made according to the provider's own information that can be gathered via arrowhead orchestration responses by the consumer.

When a provider system is registered in the Arrowhead cloud, an orchestration process enables the consumer to discover the services offered by the provider. This process is facilitated by the Orchestrator service, which collects information from both the consumer and the provider to establish a connection. Each provider system can include metadata fields, configured during its registration process. In this process, details about each service the provider contains can be described in its application properties. Details such as its message schemas, ontologies, URIs, etc…

The Arrowhead Cloud Management Tool too allows for the manual configuration of system and service details, including metadata fields, which consist of key-value pairs. In Figure 3.4, details for a specific "getSensor" service for a "provider1", upon clicking on the "show" button, next to "metadata" an information tab as seen in the same figure will pop up. Besides containing the "ontology-id", which is very relevant in case the provider messages need translation, there is a changeable "xsd-schema" field key that, supposedly contains in its value, the message schema for this specific service.

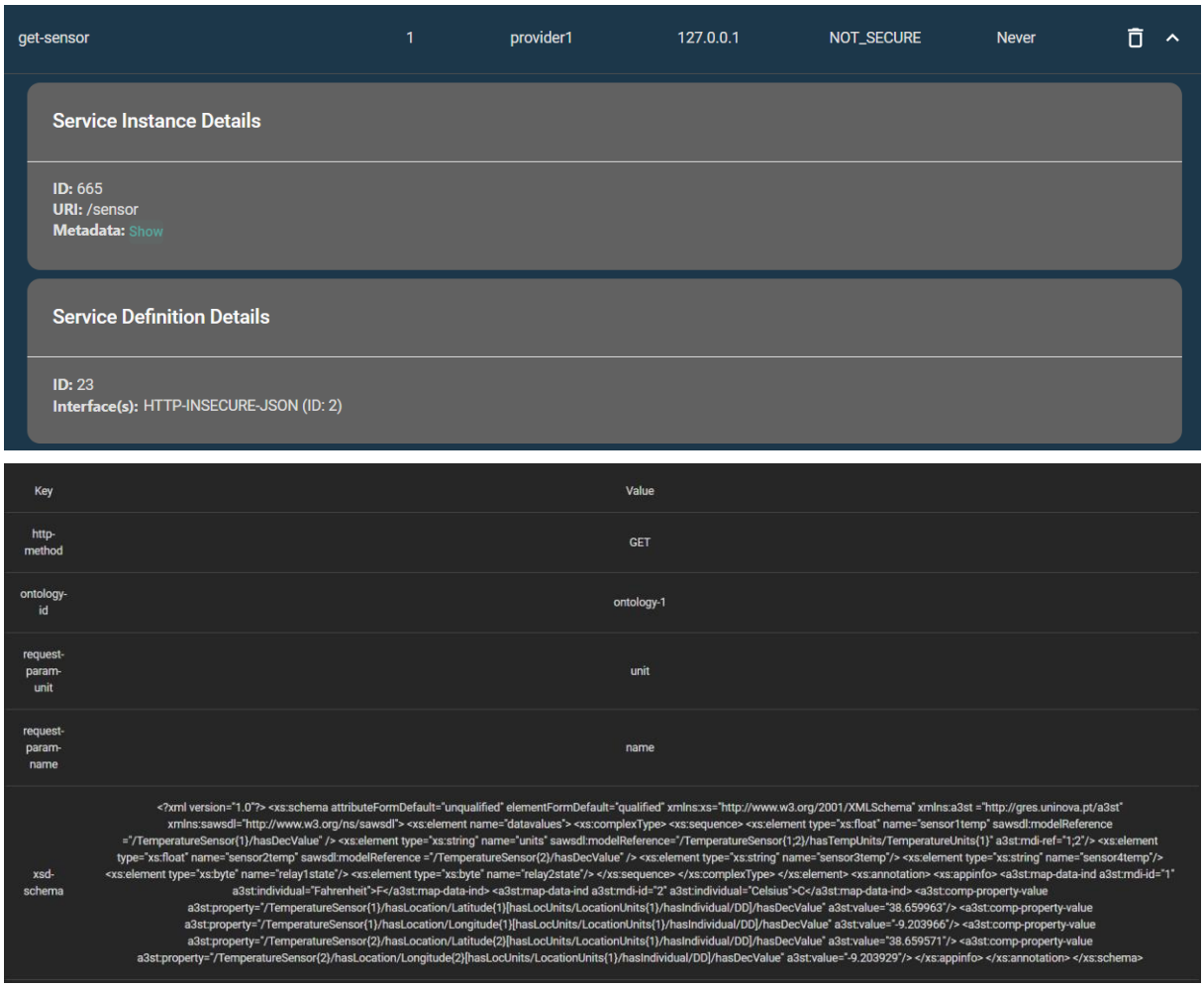


Figure 3.4 - Provider's information in the Arrowhead system

With the ability to add ontologies and XSD message schemas for new providers and their services, selecting the most suitable provider is possible by comparing the provider's service metadata with the consumer's requirements for consuming the same service. Given that a provider may offer multiple services, manually adding metadata for each service would be a time-consuming task. A more practical approach is the automatic registration of metadata upon provider system registration.

To address this need, a method was developed to automate the inclusion of metadata during the service registration process in the Arrowhead Cloud. This method automatically adds essential fields, such as "ontology-id" and "xsd-schema", for each service. Listing 3.1 shows a portion of the code responsible for this automation, significantly improving efficiency. It is important to note that, in real-world scenarios, a user can manually insert metadata fields when registering a provider. However, the outcome of this work remains consistent regardless of the approach chosen by the user.

```
 1.          // Register services into ServiceRegistry
 2.          final ServiceRegistryRequestDTO createSensorServiceRequest = createServiceRegis-
tryRequest(SensorProviderConstants.CREATE_SENSOR_SERVICE_DEFINITION, SensorProviderCon-
stants.SENSOR_URI, HttpMethod.POST);
 3.          createSensorServiceRequest.getMetadata().put("ontology-id", "ontology-1");
 4.          createSensorServiceRequest.getMetadata().put("xsd-schema1",
 5.          "<?xml version=\"1.0\"?>\n" +
 6.          "<xs:schema attributeFormDefault=\"unqualified\" elementFormDefault=\"qualified\"
xmlns:xs=\"http://www.w3.org/2001/XMLSchema\" xmlns:a3st=\"http://gres.uninova.pt/a3st\"
xmlns:sawsdl=\"http://www.w3.org/ns/sawsdl\">\n" +
 7.          "  <xs:element name=\"datavalues\">\n" +
 8.          "    <xs:complexType>\n" +
 9.          "      <xs:sequence>\n" +
10.          "        <xs:element type=\"xs:string\" name=\"units\" sawsdl:modelReference=\"/Tem-
peratureSensor{1;2}/hasTempUnits/TemperatureUnits{1}\" a3st:mdi-ref=\"1;2\"/>\n" +
11.          "        <xs:element type=\"xs:float\" name=\"sensor1temp\" sawsdl:modelRefer-
ence=\"/TemperatureSensor{1}/hasDecValue\" minOccurs=\"1\"/>\n" +
12.          "        <xs:element type=\"xs:float\" name=\"sensor2temp\" sawsdl:modelRefer-
ence=\"/TemperatureSensor{2}/hasDecValue\" minOccurs=\"1\"/>\n" +
13.          "        <xs:element type=\"xs:float\" name=\"sensor3temp\" sawsdl:modelRefer-
ence=\"/TemperatureSensor{3}/hasDecValue\" minOccurs=\"1\"/>\n" +
14.          "      </xs:sequence>\n" +
15.          "    </xs:complexType>\n" +
16.          "  </xs:element>\n" +
17.          "  <xs:annotation>\n" +
18.          "  </xs:annotation>\n" +
19.          "</xs:schema>");
20.
21.          arrowheadService.forceRegisterServiceToServiceRegistry(createSensorServiceRequest);
22.
```

Listing 3.1 - Code for registering a provider's "createSensor" service metadata

By automating metadata registration, providers with multiple services can be seamlessly integrated into the system. This approach has been tested with several providers and services, confirming that the "ontology-id" and "xsd-schema" fields are correctly included. These fields are crucial for ensuring that systems can interpret the data exchanged during service consumption, especially when translation of messages is necessary due to differences in ontologies or message formats.

A consumer service has been developed with the capability to gather provider metadata fields from the Orchestration service's response. This consumer retrieves metadata for relevant providers and writes it to an external file named "provider_info.txt" for further analysis.

The metadata gathered from the orchestration response includes only the ontology identification and XSD schema fields, which are critical for identifying whether message translation is possible and ensuring that the exchanged data adheres to the appropriate format. The code shown in Listing 3.2 below is responsible for iterating over the orchestration response and writing the relevant information to a file (line 32). It also checks if providers are online by calling the method "isProviderRunning" in line 22.

```
 1.     public Boolean listProviders(String title, OrchestrationResponseDTO response) {
 2.         System.out.println("Doing listProviders");
 3.         if (response == null || response.getResponse().isEmpty()) {
 4.             // If no proper providers found during the orchestration process, then the re-
sponse list will be empty. Handle the case as you wish!
 5.             System.out.println("Orchestration response is empty");
 6.             return false;
 7.         }
 8.         try (FileWriter writer = new FileWriter("providers_info.txt")) {
 9.             writer.write("-------------------------------------------------------------------
--------------\n");
10.             writer.write(title + "\n");
11.             writer.write(String.format("%-20s %-40s %-10s%n", "Provider Name", "URL",
"Port"));
12.             writer.write("-------------------------------------------------------------------
--------------\n");
13.
14.             List<OrchestrationResultDTO> results = response.getResponse();
15.             for (OrchestrationResultDTO result : results) {
16.                 String providerName = result.getProvider().getSystemName();
17.                 String address = result.getProvider().getAddress();
18.                 int port = result.getProvider().getPort();
19.                 String serviceUri = result.getServiceUri();
20.                 String url = address + ":" + port + "/" + serviceUri;
21.
22.                 if (isProviderRunning(address, port, serviceUri, result)) {
23.                     writer.write(String.format("%-20s %-40s %-10d%n", providerName, url,
port));
24.                     System.out.printf("%-20s %-40s %-10d%n", providerName, url, port);
25.
26.
27.                     Map<String, String> meta = result.getMetadata();
28.                     if (meta != null && !meta.isEmpty()) {
29.                         writer.write("  Metadata:\n");
30.                         System.out.println("  Metadata:");
31.                         for (Map.Entry<String, String> entry : meta.entrySet()) {
32.                             writer.write(String.format("     %-20s: %s%n", entry.getKey(),
entry.getValue()));
33.                             System.out.printf("     %-20s: %s%n", entry.getKey(), entry.get-
Value());
34.                         }
35.                     }
36.                 } else {
37.                     System.out.println("Skipping provider: " + providerName + " (not run-
ning)");
38.                 }
39.             }
40.         } catch (IOException e) {
41.             e.printStackTrace();
42.             return false;
43.         }
44.         return true;
45.     }
```

Listing 3.2 - Method for listing and saving data from a selected service

From the iteration through "Orchestration Result DTO" (Data Transfer Object) objects, which contain the relevant metadata of the provider systems. The "ontology-id" and "xsd-schema" values are extracted from the provider metadata and written to the file. These fields are necessary for compatibility analysis and translator generation between providers and consumers that rely on different data structures or ontologies.

Additionally, it is crucial that the consumer also provides its own metadata, which is saved in a file named "consumer_info.txt". This mirrors the provider information gathered earlier. The consumer can register its metadata manually in the Arrowhead environment, but in this project, the metadata is specified within the "application.properties" file, that is native to spring boot projects (Listing 3.3), immediately following the "metadata fields" comment on line 42. The rest of the configurations for the Consumer system are all present in Listing 3.3 too: Its name on line 6, address and part on lines 11 and 12, arrowhead security variables below line 24 etc.

```
1. ###########################################
2. ###       CUSTOM PARAMETERS              ###
3. ###########################################
4.
5. # Name of the client system
6. application_system_name=consumerDTMV1
7.
8. # Set the web application type to 'servlet' if your consumer client should act as a web-
server
9. # and fill the address and port propetries with valid values [Defaults, when not adjusted:
localhost:8080]
10. spring.main.web-application-type=none
11. server.address=127.0.0.1
12. server.port=8010
13.
14. # Service Registry Core System web-server parameters
15. sr_address=127.0.0.1
16. sr_port=8443
17.
18. ###########################################
19. ###           SECURE MODE               ###
20. ###########################################
21.
22. # configure secure mode
23.
24. # Set this to false to disable https mode
25. server.ssl.enabled=false
26.
27. server.ssl.key-store-type=PKCS12
28. server.ssl.key-store=classpath:certificates/consumerskeleton.p12
29. server.ssl.key-store-password=123456
30. server.ssl.key-alias=consumerskeleton
31. server.ssl.key-password=123456
32. server.ssl.client-auth=need
33. server.ssl.trust-store-type=PKCS12
34. server.ssl.trust-store=classpath:certificates/truststore.p12
35. server.ssl.trust-store-password=123456
36.
37. #path configuration
```

```
38. runall_bat_path= D:/TranslatorTool/consumerMetadaTAG2/runAll.bat
39. translate_bat_path=D:/TranslatorTool/consumerMetadaTAG2/translate.bat
40. selecter_provider_url_file=D:\\TranslatorTool\\consumerMetadaTAG2\\src\\main\\TAG-transla-
tor\\selected_provider.txt
41.
42. # metadata fields
43. arrowhead.system.metadata.key1=ontology1
44. arrowhead.system.metadata.key2=<?xml version="1.0"?><xs:schema attributeFormDefault="unqual-
ified" elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:a3st
="http://gres.uninova.pt/a3st"  xmlns:sawsdl="http://www.w3.org/ns/sawsdl"><xs:element
name="datavalues" ><xs:complexType><xs:sequence><xs:element type="xs:string" name="units"
sawsdl:modelReference="/TemperatureSensor{1;2;3;4}/hasTempUnits/TemperatureUnits{1}" a3st:mdi-
ref="1;2"/><xs:element type="xs:float" name="sensor1temp" sawsdl:modelReference ="/Temperature-
Sensor{1}/hasDecValue" /><xs:element type="xs:float" name="sensor2temp" minOccurs="0" max-
Occurs="1" sawsdl:modelReference ="/TemperatureSensor{2}/hasDecValue"/><xs:element
type="xs:float" name="sensor3temp" minOccurs="0" maxOccurs="1" sawsdl:modelReference ="/Tempera-
tureSensor{3}/hasDecValue"/><xs:element type="xs:float" name="sensor4temp" minOccurs="0" max-
Occurs="1" sawsdl:modelReference ="/TemperatureSensor{4}/hasDecValue"/></xs:sequence></xs:com-
plexType></xs:element><xs:annotation><xs:appinfo><a3st:map-data-ind a3st:mdi-id="1" a3st:indi-
vidual="Fahrenheit">Fah</a3st:map-data-ind><a3st:map-data-ind a3st:mdi-id="2" a3st:individ-
ual="Celsius">Cel</a3st:map-data-ind></xs:appinfo></xs:annotation></xs:schema>
45.
```

Listing 3.3 - Application properties file for the Consumer

To extract the metadata fields into a file, a method named "listConsumerMetadata()" is called. Said method is represented in Listing 3.4, and it is set to get the consumer data from its programmed configuration interface (line 3) and write its metadata entries into the desired file. As shown in Listing 3.3, above, the metadata keys are its ontology, and message schema.

```
 1.     public void listConsumerMetadata() {
 2.         SystemRequestDTO consumerSystem = consumerSystemConfig.getConsumerSystem();
 3.         Map<String, String> metadata = consumerSystem.getMetadata();
 4.         // Write metadata to a file
 5.         try (FileWriter writer = new FileWriter("consumer_info.txt")) {
 6.             writer.write("-------------------------------------------------------------------
--------------\n");
 7.             writer.write("Consumer Metadata\n");
 8.             writer.write("-------------------------------------------------------------------
--------------\n");
 9.             if (metadata == null || metadata.isEmpty()) {
10.                 System.out.println("No metadata found for this consumer.");
11.                 writer.write("No metadata found for this consumer.\n");
12.             } else {
13.                 System.out.println("Consumer Metadata:");
14.                 writer.write(String.format("%-20s %-40s%n", "Key", "Value"));
15.                 writer.write("-------------------------------------------------------------------
------------------\n");
16.
17.                 for (Map.Entry<String, String> entry : metadata.entrySet()) {
18.                     String key = entry.getKey();
19.                     String value = entry.getValue();
20.                     System.out.printf("  %-20s: %s%n", key, value);
21.                     writer.write(String.format("  %-20s: %s%n", key, value));
22.                 }
23.             }
24.             writer.write("-------------------------------------------------------------------
--------------\n");
25.             System.out.println("Consumer metadata saved to consumer_info.txt");
26.         } catch (IOException e) {
27.             System.err.println("Error writing consumer metadata to file.");
28.             e.printStackTrace();
29.         }
30.     }
```

Listing 3.4 - Method for writing the consumer's service data into a file

The next phase of this work focuses on extracting the actual XSD files for each provider. With the metadata gathered and written to an external file, the consumer service can identify the schema files required to interact with specific providers. This will be explained in the following section, where the process for extracting and utilizing the XSD files from the providers is detailed.

3.4.3 Extract Providers and Consumer Schemas

The process of generating both provider and consumer XSD schema files relies on filtering relevant metadata from unprocessed information gathered from the Arrowhead system. The initial data collected in the "provider_info.txt" and "consumer_info.txt" files contains various details, some of which are unnecessary for the current use case. Therefore, it's crucial to isolate the essential message schema data while disregarding unrelated provider information such as names and URIs.

As previously mentioned, the retrieval of the provider's service metadata from the "provider_Info.txt" file is crucial. By obtaining the providers and the single consumer XSD files, the TAG Tool can be executed to generate a report in JSON format. This report is key, as it will be further analysed to assess the suitability of the provider, a process that will be elaborated upon in subsequent sections.

The consumer.xsd is the first to be filtered out of the information file, as it is a much simpler process. The "consumer_info.txt" contains data about a single consumer. Following this first step, by filtering the provider message schemas out of the provider information text files into provider XSD schema files, it is then possible to execute the TAG with a given ontology and generate the data to learn more about each provider.

To enable the generation of provider schema files, a script named "ProviderXSDGenerator" was initially developed in Java. This script is responsible for extracting the various provider schemas from the "provider_Info.txt" file and placing them into an input directory, as specified in a configuration JSON file.

The configuration JSON file, "config.json", is also an essential component of the project, as it dictates how and where the system stores and processes the provider schemas as well as where certain files are to be saved in the next parts of the project. The "ProviderXSDGenerator" class is presented in Listing 3.5.

This automated approach not only simplifies the process of gathering provider schemas but also ensures that the necessary information is readily available for the TAG -Tool to perform its analysis, ultimately assisting in the validation and selection of the most appropriate provider within the Arrowhead ecosystem.

The script takes an input file, typically "providers_info.txt", and an input directory where the resulting provider XSD files will be stored. It searches the input file for lines starting with "xsd-schema", which indicate the start of a provider's service XML message schema. From the line containing "<?xml version=", the script collects all relevant XML content until the closing "</xs:schema>" tag, storing it in a StringBuilder. Once the XML block is complete, it writes the schema to a file named "providerX.xsd" (where X represents the provider index). For example, if four schemas are found in "provider_info.txt", four schema files will be generated: "provider1.xsd", "provider2.xsd", and so on.

```
1. import java.io.BufferedReader;
2. import java.io.FileReader;
3. import java.io.FileWriter;
4. import java.io.IOException;
5. import java.nio.file.Files;
6. import java.nio.file.Path;
7. import java.nio.file.Paths;
8.
9. public class ProviderXSDGenerator {
```

```
10.     public static void main(String[] args) {
11.         if (args.length < 1) {
12.             System.err.println("Usage: java ProviderXSDGenerator <in-put_file> [output_directory]");
13.             System.exit(1);
14.         }
15.
16.         String inputFileName = args[0];
17.         String outputDirectory = args.length > 1 ? args[1] : ".";
18.         generateXSDFiles(inputFileName, outputDirectory);
19.
20.
21.     }
22.     public static void generateXSDFiles(String inputFileName, String outputDirectory) {
23.         try (BufferedReader reader = new BufferedReader(new FileReader(inputFileName))) {
24.             String line;
25.             int count = 0;
26.             int providerCount = 0;
27.             boolean inXsdElementBlock = false;
28.             StringBuilder metadataBuilder = new StringBuilder();
29.
30.             // Ensure the output directory exists
31.             Path outputPath = Paths.get(outputDirectory);
32.             if (!Files.exists(outputPath)) {
33.                 Files.createDirectories(outputPath);
34.             }
35.             while ((line = reader.readLine()) != null) {
36.                 if (line.startsWith("providerID")) {
37.                     if (metadataBuilder.length() > 0) {
38.                         String xsdFileName = outputDirectory + "/provider" + providerCount + ".xsd";
39.                         writeXSDFile(xsdFileName, metadataBuild-er.toString().trim());
40.                         metadataBuilder.setLength(0); // Clear StringBuild-er for new provider
41.                     }
42.                     providerCount++;
43.                     System.out.println("Processing " + line.trim());
44.                 }
45.
46.                 if (line.trim().startsWith("xsd-schema") && line.contains("<?xml version=")) {
47.                     inXsdElementBlock = true;
48.                     System.out.println("Found <?xml version= start");
49.                     int startIndex = line.indexOf("<?xml version=");
50.                     if (startIndex != -1) {
51.                         metadataBuilder.append(line.substring(startIndex).trim()).append("\n");
52.                     }
53.                 } else if (inXsdElementBlock) {
54.                     metadataBuilder.append(line.trim()).append("\n");
55.                 }
56.                 if (inXsdElementBlock && line.trim().endsWith("</xs:schema>")) {
57.                     inXsdElementBlock = false;
58.                     System.out.println("Found </xs:schema> end");
59.                 }
60.             }
61.             if (metadataBuilder.length() > 0) {
62.                 String xsdFileName = outputDirectory + "/provider" + pro-viderCount + ".xsd";
63.                 writeXSDFile(xsdFileName, metadataBuild-er.toString().trim());
64.                 count ++;
65.             }
66.             if (providerCount == count)
67.             {
68.                 System.out.println("Could Use the reports now ig");
69.             }
70.         } catch (IOException e) {
71.             e.printStackTrace();
72.         }
73.     }
74. }
```

Listing 3.5 - Class for generating XSDs out of the provider data file

Although referring to the output directory as the "input directory" might seem counter-intuitive, it serves as the input for the TAG Tool, which uses the schema files to generate reports for evaluating providers. This will be detailed further in the next subsection, where the TAG Tool's role in provider evaluation is discussed.

Java was chosen for its compatibility with the Arrowhead framework. This makes the script robust and adaptable, and it could potentially be integrated directly into the Arrowhead consumer system in the future.

However, as the project progressed, it became apparent that a more efficient solution could be achieved by integrating the provider and consumer XSD generation process into the same script that executes subsequent steps. This new approach, implemented in Python, was adopted towards the end of the project. The reason for this shift was the realization that filtering the provider schemas into the input directory, would be more effectively done during the same process that stores the provider information in the database which will be explained in its own section.

The provider information, can be filtered directly and linked with specific metadata in the database, making it easier to extract and associate relevant related provider details. This allows the TAG Tool to retrieve the necessary provider information while it is being stored in the database alongside other useful data.

This change simplifies the workflow and improves efficiency. Nonetheless, the Java-based "ProviderXSDGenerator" remains included in this subchapter because it could still be directly integrated into the Arrowhead consumer system if desired. By allowing the DTM to handle provider filtering, the consumer does not need to filter XSD files directly. Instead, it can simply output its own metadata and provider metadata, setting up the DTM to feed TAG with the necessary inputs. Therefore, making it a more adaptable tool for numerous consumer systems. Instead of requiring a very specific consumer to work with.

3.4.4 Managing the TAG Tool for File Generation

The integration of the TAG Tool within the project is crucial for generating the necessary reports and translator files that facilitate communication between consumers and providers in the Arrowhead system. This subsection will cover the setup and execution of the TAG Tool and its eventual integration into the main DTM Python script, "ProviderConsumerManager.py", which was enhanced to automate these steps more efficiently.

After extracting provider XSD schema files in the first step of the main DTM script, the TAG Tool is used to analyse these files. As the tool processes one provider-consumer pair at a time, a batch script is designed to sequentially handle multiple provider files, producing individual report files and translator Java classes for each provider. The steps performed by the batch script include detecting provider files, executing the TAG Tool in a loop, and renaming the output files accordingly.

The referenced batch file is present in Listing 3.6 was created, and it was based on the already available original batch script for the TAG tool. This script enables the tool to process an unlimited number of provider files sequentially, generating individual report files and translator classes for each provider-consumer pair. The modified batch file automates this process by detecting the number of provider.xsd files in the input directory and using a for loop to execute the TAG Tool for each file.

The batch script operates as follows:

1. **Detection of Provider Files**: The script first counts the number of provider.xsd files in the input directory.
2. **Execution Loop**: It then enters a loop that runs for each detected provider file. For each iteration, the TAG Tool's JAR file is executed with the relevant provider schema (provider.xsd), consumer schema (consumer.xsd), and ontology data (ontology.owl).
3. **File Renaming**: After each execution, the script checks if the output files (translator.java, report.html, and report.json) exist. If they do, the files are renamed to correspond to the specific provider being processed (e.g., translator1.java, report1.json, etc.).

In a practical scenario where there are three provider schema files in the input directory, the TAG Tool will run three times, generating separate output files for each provider:

- For provider1.xsd, it will generate translator1.java, report1.json, and report1.html.
- For provider2.xsd, it will generate translator2.java, report2.json, and report2.html.
- For provider3.xsd, it will generate translator3.java, report3.json, and report3.html.

It's important to note that while the report.html files are produced by the TAG Tool too, they are not directly relevant to the provider selection process. However, their generation does not pose any issues for the execution of the script, and the html interface provides a visually appealing format of the report.

```
@echo off
setlocal enabledelayedexpansion

if "%~1"=="" (
    echo No provider IDs specified.
    pause
    exit /b
)
:: Change to the output directory
cd /d D:\TranslatorTool\consumerMetadaTAG2\src\main\TAG-tool

echo Processing specified provider files...

for %%i in (%*) do (
    echo Processing provider%%i.xsd
    java -jar TAG-tool.jar ^
    --providerData="provider%%i.xsd" ^
    --providerType=file ^
    --consumerData="consumer.xsd" ^
    --consumerType=file ^
    --ontologyData="ontology.owl" ^
    --ontologyType=file ^
    --outType=file ^
    --outDirectory=.

    if exist translator.java (
        ren translator.java translator%%i.java
    )
    if exist report.html (
        ren report.html report%%i.html
    )
    if exist report.json (
        ren report.json report%%i.json
    )
)
echo Done processing specified provider files.

```

Listing 3.6 - One of the DTM batch files, used for processing each provider - upgraded from the original TAG version

To streamline the workflow and reduce the number of batch files required, the execution of the TAG Tool was integrated directly into the "ProviderConsumerManager" python script, eliminating the need for an external batch file. This integration was achieved by using Python's

subprocess module, which makes the process more flexible and easier to manage within a single script.

The function responsible for this process, "run_translator_tool", accepts three key parameters: providers, "input_directory", and "jar_path". The provider's parameter is a dictionary that contains the necessary information for each provider, including their associated XSD file paths. This function is visible in Listing 3.7. It calls the TAG Tool using "subprocess.run()". The command executed via "subprocess.run()" mirrors the same functionality previously handled by a batch file but with added benefits. The process allows for dynamic error handling and logging within the Python script itself. With "capture_output=True", the tool's output (including errors) is captured, making it easier to debug and diagnose any issues that arise during execution.

After the TAG Tool processes each provider, the generated files ("translator.java", "report.json", and "report.html") are renamed to include a sanitized version of the provider's URL, sanitized simply meaning the names have no invalid characters, which would be common for files named after URIs. This ensures that the output files for each provider are clearly identified and associated with the correct provider in a structured manner.

Using Python's subprocess module offers several advantages over the original batch file approach. First, it introduces cross-platform compatibility, as Python can run on various operating systems, unlike a batch script which is limited to Windows environments. And it was also a way to explore Python possibilities, therefore acquiring useful knowledge for the next development steps.

```
def run_translator_tool(providers, input_directory, jar_path):
    consumer_data = os.path.join(input_directory, "consumer.xsd")
    ontology_data = os.path.join(input_directory, "ontology.owl")

    for provider_url, provider_data in providers.items():
        provider_file_path = os.path.join(input_directory, provider_data['pro
vider_file_name'])
        with open(provider_file_path, 'wb') as file:
            file.write(provider_data['provider_content'])
        print(f"Running TAG-tool for provider with URL {provider_url}")
        result = subprocess.run([
            'java', '-jar', jar_path,
            '--providerData=' + provider_file_path,
            '--providerType=file',
            '--consumerData=' + consumer_data,
            '--consumerType=file',
            '--ontologyData=' + ontology_data,
            '--ontologyType=file',
            '--outType=file',
            '--outDirectory=' + input_directory
        ], capture_output=True, text=True)

        if result.returncode != 0:
            print(f"Error running TAG-tool for provider {provider_url}: {result.stderr}")
        else:
            print(f"TAG-tool output: {result.stdout}")

        sanitized_url = sanitize_filename(provider_url)
        translator_java_path = os.path.join(input_directory, "translator.java")
        if os.path.exists(translator_java_path):
            os.rename(translator_java_path, os.path.join(input_directory, f"translator_{san-
itized_url}.java"))
        report_html_path = os.path.join(input_directory, "report.html")
        if os.path.exists(report_html_path):
            os.rename(report_html_path, os.path.join(input_directory, f"report_{sani-
tized_url}.html"))
        report_json_path = os.path.join(input_directory, "report.json")
        if os.path.exists(report_json_path):
            os.rename(report_json_path, os.path.join(input_directory, f"report_{sani-
tized_url}.json"))
```

Listing 3.7 - Method for executing the TAG Tool

The TAG Tool itself can be somewhat slow when processing new providers, meaning that as the number of available providers increases, the execution time will also increase. In scenarios where a large number of providers are included in the most recent orchestration message, this can lead to higher-than-ideal execution times, from testing responses execution times went as up as 5 seconds for each provider. Therefore, it would be beneficial to implement a system that avoids reprocessing providers that have already been analysed by the TAG Tool, unless there has been a change in one of the provider's metadata. This metadata changes detection feature for each and every previously registered provider is also implemented.

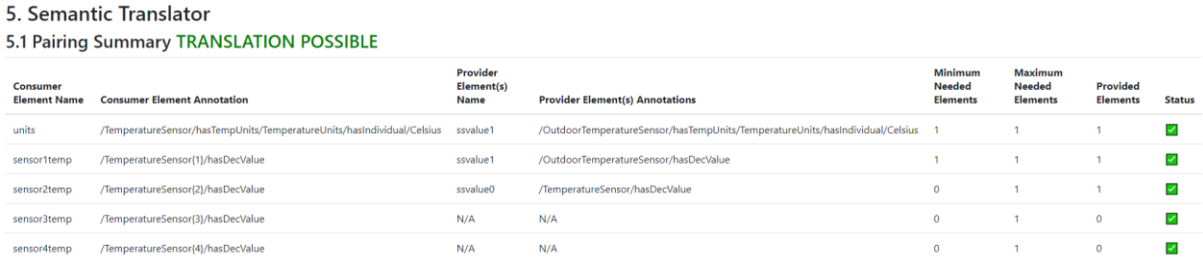
To address this, a dynamic approach is suggested. By storing the relevant information from previous TAG Tool runs in a database, the system could avoid unnecessary re-executions, thereby significantly speeding up the provider selection process. This is of special relevance as

the TAG Tool's execution is currently the most time-consuming part of the entire process. Incorporating a database to track and manage processed providers would optimize the system's performance and ensure that only new or updated providers are re-evaluated.

3.4.5 Dynamic selection of the most appropriate provider

Selecting the most appropriate provider is crucial for ensuring that the consumer's needs are optimally met, as different consumers may have varying requirements. While multiple providers may be suitable for a particular consumer, the challenge is selecting the "best" provider from the available options. The provider information file, generated by the Consumer Arrowhead system, offers a list of potential providers. For each provider, schema files are generated, followed by a detailed report for each one.

Understanding the content of these JSON report files is essential in determining what makes a provider "good" or appropriate for a specific consumer. Figure 3.5, (a snippet of the report in HTML format) illustrates a section of such a report. This section first indicates whether a translation for the provider message, based on its metadata, is possible. While this is an important factor, it is not enough to make an informed decision; a more detailed analysis of the report is required.



5. Semantic Translator

5.1 Pairing Summary TRANSLATION POSSIBLE

Consumer Element Name	Consumer Element Annotation	Provider Element(s) Name	Provider Element(s) Annotations	Minimum Needed Elements	Maximum Needed Elements	Provided Elements	Status
units	/TemperatureSensor/hasTempUnits/TemperatureUnits/hasIndividual/Celsius	ssvalue1	/OutdoorTemperatureSensor/hasTempUnits/TemperatureUnits/hasIndividual/Celsius	1	1	1	✅
sensor1temp	/TemperatureSensor{1}/hasDecValue	ssvalue1	/OutdoorTemperatureSensor/hasDecValue	1	1	1	✅
sensor2temp	/TemperatureSensor{2}/hasDecValue	ssvalue0	/TemperatureSensor/hasDecValue	0	1	1	✅
sensor3temp	/TemperatureSensor{3}/hasDecValue	N/A	N/A	0	1	0	✅
sensor4temp	/TemperatureSensor{4}/hasDecValue	N/A	N/A	0	1	0	✅

Figure 3.5 - TAG generated html report

In the report, the key fields to focus on are the "Minimum Needed Elements" and "Provided Elements." The "Minimum Needed Elements" field indicates the essential elements required for translation. If the required elements (those marked as "1" in the "Minimum Needed Elements" column) are met (i.e., there is a corresponding "1" in the "Provided Elements" column), it is reasonable to assume that the translation is feasible. However, the goal is not only to meet the minimum requirements but also to assess how many additional, non-essential elements the provider can satisfy.

For example, if a provider grants one extra provided element, beyond the minimum requirements, this could indicate an additional capability that enhances the system's

performance. In a practical case, imagine an air conditioning (AC) system that requires at least two sensor readings to function but can utilize up to five for maximum efficiency. While the system will work with the two required sensors, using additional sensors could improve its overall performance. Hence, a provider that offers more than the required two sensor readings would be more advantageous than one offering only the minimum, making it the preferred choice.

To implement this selection process, a method was developed to choose the provider based on the number of "1"s in the "Provided Elements" column. The script responsible for this selection is called the Provider Analyser, developed in Python and illustrated in Listing 3.8. This script operates on the JSON version of the report, as JSON is straightforward to analyse using Python libraries.

Workflow of the Provider Analyser Script:

1. **Configuration Loading**:
    - The script begins by reading the "config.json" file (line 43), which contains the database configuration (e.g., the path for SQLite) and the output directory. The output directory is the folder where the system will place the selected provider's translator class. Typically, this would be the TAG Translator folder.
2. **Database Interaction**:
    - The script connects to the database and retrieves the relevant report JSON files for all providers. These reports contain detailed information on how well each provider meets the consumer's needs as explained above.
3. **Analysis and Selection**:
    - Each provider’s report is analysed to count how many elements in the "Provided Elements" field have a value of 1 (lines 51 to 58). This count reflects how many of the essential and additional elements the provider can fulfil. The provider with the highest count of 1s is selected as the most appropriate provider for the consumer.
4. **Retrieving and Saving Files**:
    - Before saving the new translator file and provider schema, the script deletes any outdated files (function at line 24) from the output directory that start with the file prefix, ensuring that old data doesn't interfere with the new execution. Once the most suitable provider is determined, the corresponding translator file and

provider schema are retrieved from the database and saved to the output directory. The files are renamed appropriately for further use in the system.

```
import os
import json
import sqlite3
import re

def load_config(config_file):
    with open(config_file, 'r') as file:
        return json.load(file)

def analyze_report(report_content):
    data = json.loads(report_content)
    translator = data.get("translator", [])
    count_provided_1 = sum(1 for entry in translator if entry.get("provided") == "1")
    return count_provided_1

def get_provider_data_from_db(conn):
    cursor = conn.cursor()
    cursor.execute("SELECT provider_url, report_content, translator_content, provider_file_name,
provider_content FROM ProviderInfo WHERE report_content IS NOT NULL")
    return cursor.fetchall()

def sanitize_filename(url):
    return re.sub(r'[\\/:"*?<>|]+', '_', url)

def delete_old_files(output_directory, file_prefix):
    for filename in os.listdir(output_directory):
        if filename.startswith(file_prefix):
            file_path = os.path.join(output_directory, filename)
            os.remove(file_path)
            print(f"Deleted old file: {filename}")

def remove_package_generated(file_path):
    with open(file_path, 'r') as file:
        lines = file.readlines()

    cleaned_lines = [line for line in lines if "package generated;" not in line]

    with open(file_path, 'w') as file:
        file.writelines(cleaned_lines)

    print(f"Removed 'package generated;' lines from {file_path}")

if __name__ == "__main__":
    config = load_config('config.json')
    output_directory = config.get("output_directory")
    db_path = config.get("database_path")
    conn = sqlite3.connect(db_path)
    provider_data = get_provider_data_from_db(conn)
    max_count = -1
    selected_provider_url = None

    for provider_url, report_content, translator_content, provider_file_name, provider_content in
provider_data:
        if report_content:
            count_1s = analyze_report(report_content)
            print(f"Report for provider '{provider_file_name}' has {count_1s} elements with 'pro-
vided' value '1'.")
            if count_1s > max_count:
                max_count = count_1s
                selected_provider_url = provider_url
                selected_provider_file_name = provider_file_name
    if selected_provider_url:
        print(f"\nProvider '{selected_provider_file_name}' with URL '{selected_provider_url}' has
the maximum number of elements with 'provided' value '1': {max_count}")
        selected_data = next(item for item in provider_data if item[0] == selected_provider_url)
```

```
62.         translator_content = selected_data[2]
63.         provider_file_name = selected_data[3]
64.         provider_content = selected_data[4]
65.         clean_url = sanitize_filename(selected_provider_url)
66.         delete_old_files(output_directory, "Translator")
67.         selected_provider_path = os.path.join(output_directory, "selected_provider.txt")
68.         with open(selected_provider_path, 'w') as file:
69.             file.write(selected_provider_url)
70.         print(f"Selected provider URL written to {selected_provider_path}")
71.
72.         if translator_content:
73.             translator_content_str = translator_content.decode('utf-8')
74.             destination_translator_path = os.path.join(output_directory, "Translator.java")
75.             with open(destination_translator_path, 'w') as file:
76.                 file.write(translator_content_str)
77.             print(f"Translator content written to {destination_translator_path}")
78.             remove_package_generated(destination_translator_path)
79.         else:
80.             print(f"Translator content for provider '{selected_provider_file_name}' not found in
the database.")
81.     else:
82.         print("No valid report files found in the database.")
83.     conn.close()
```

Listing 3.8 - Script responsible for selecting providers based on the report data

3.4.6 Storing Relevant Provider Information in a Database

Storing the relevant provider files in a database is an effective way to optimize the workflow of the Data Translation Manager by eliminating the need to repeatedly generate new files. This method allows the system to avoid running external OS commands unnecessarily every time the tool is executed from start to finish.

The first step in this process was determining which information was essential enough to store in the database. The database stores three key files:

1. The **provider schema** (XSD),
2. The **JSON report** (report.json)
3. The **translator Java class** (Translator.java)

The HTML report is excluded from storage, as it serves a primarily visual purpose and is not needed for any subsequent steps in provider selection.

The provider metadata is saved so that each provider can be precisely identified and tracked. The Provider Analyzer uses this metadata during analysis, making it crucial to store the JSON report in the database rather than relying on local files. Similarly, the translator class is stored in the database so that the Provider Analyzer can access and copy its contents to the specified "output directory," as the system is designed to expect these elements to be retrieved from the database. This decision to save critical data helps prevent unnecessary clutter and

reduces the risk of overloading the consumer's local system with redundant files generated upon each DTM execution.

As illustrated in Figure 3.6, which is an image taken from the app developed for database visualization purposes, the database structure is straightforward: a single table is sufficient to manage the necessary data. The table's primary column is the **provider URL**, which uniquely identifies each provider's metadata. This URL acts as a key that links to all related information for that provider. This URL is often referred to as URI.

If any changes occur in a provider's metadata, the system automatically detects the modification during the next execution of the Data Translation Manager. The process works as follows:

- If a provider's metadata is changed or deleted, the system reruns the TAG Tool for that provider, generating a new report and translator class. This updated information is then saved back into the database.
- If a provider's metadata remains unchanged, the system assumes the existing database entry is correct and skips the process of rerunning the TAG Tool, saving both time and system resources.

In a system where there Is necessity for quick updates of provider input, this step Is considered to be of great importance, it makes the workflow much less cramped, more dynamic, and establishes logical relations to every bit of data about each detected provider, both gathered or generated. The database always contains the latest, relevant data. Furthermore, testing of this approach in the final product revealed no errors—each time a provider was added, removed, or changed, the system functioned correctly.

Provider Data Table

Provider URL	Provider File Name	Provider Content (Snippet)	Report Content (Snippet)	Translator Content (Snippet)
127.0.0.1:8021//sensor	provider1.xsd	<?xml version="1.0"?> <xs:schema attributeForm…	{"provider":{"TemperatureSensor{4}":[{"annotati…	package pt.uninova.ditag.translator; import com.fa…
127.0.0.1:8022//sensor	provider2.xsd	<?xml version="1.0"?> <xs:schema attributeForm…	{"provider":{"TemperatureSensor{4}":[{"annotati…	package pt.uninova.ditag.translator; import com.fa…
127.0.0.1:8023//sensor	provider3.xsd	<?xml version="1.0"?> <xs:schema attributeForm…	{"provider":{"TemperatureSensor{4}":[{"annotati…	package pt.uninova.ditag.translator; import com.fa…
127.0.0.1:8024//sensor	provider4.xsd	<?xml version="1.0"?> <xs:schema attributeForm…	{"provider":{"TemperatureSensor{4}":[{"annotati…	package pt.uninova.ditag.translator; import com.fa…

Figure 3.6 - Image of the database contents

3.4.7 Translation of selected provider's xml message

A **POST** request must be made to the provider URI in a typical use case, such as a sensor from an air conditioning system posting its temperature readings to the provider's URI, enabling a consumer to later access this data. In this scenario, the consumer could be the air conditioner's controller system. Any type of system can post its collected data to the provider's URIs, if it respects its message format. This is how sensor readings are registered and subsequently accessed by consumers within the Arrowhead Cloud environment. For a consumer to access the sensor readings available at the provider's URI, a simple **GET** request is required.

In this project, POST requests simulating sensor readings were made using Postman, an API development platform, and through specifically developed provider services. These services initiate a POST request to the available provider's services that require sensor values upon execution. Listing 3.9 shows the code responsible for adding sensor readings to providers as an alternative to Postman. This method was employed for both testing purposes and in the final implementation. It is crucial that the messages sent via POST requests conform to the provider service's metadata format, nothing in the arrowhead or spring boot, as far as was explored, verifies this is respected. This was implemented carefully, but if the message format within the URI does not adhere to the provider's XSD schema, errors may occur, as the pre-generated translator class from the TAG Tool may not be compatible with the provider's message format.

```
1.      // Auto-register sensor after application is fully started
2.      @EventListener(ApplicationReadyEvent.class)
3.      public void autoRegisterSensor(ApplicationReadyEvent event) {
4.
5.        RestTemplate restTemplate = event.getApplicationContext().getBean(RestTemplate.class);
6.          String sensorUri = "http://127.0.0.1:8023/sensor";
7.          SensorRequestDTO sensorRequest = new SensorRequestDTO();
8.          sensorRequest.setUnit("Cel");
9.          sensorRequest.setsensor1temp(25.5f);
10.         sensorRequest.setsensor2temp(22.3f);
11.         sensorRequest.setsensor3temp(23.3f);
12.         HttpHeaders headers = new HttpHeaders();
13.         headers.set("Content-Type", "application/json");
14.       HttpEntity<SensorRequestDTO> requestEntity = new HttpEntity<>(sensorRequest, headers);
15.         try {
16.                 ResponseEntity<String> response = restTemplate.exchange(sensorUri, Http-
Method.POST, requestEntity,String.class);
17.
18.             if (response.getStatusCode() == HttDTMatus.CREATED) {
19.                 System.out.println("Sensor registered successfully.");
20.             } else {
21.                 System.out.println("Sensor registration failed with status code: " + re-
sponse.getStatusCode());
22.             }
23.         } catch (Exception e) {
24.             System.err.println("Error during sensor registration: " + e.getMessage());
25.         }
26.     }
```

Listing 3.9 - Method for registering a sensor reading in the "createSensor" service URI for Provider 3

After selecting the appropriate translator class generated by the TAG Tool and placing it in the output directory, the translation node uses the desired provider message for translation according to the consumer's needs. As previously mentioned, provider messages can be retrieved by executing GET requests to the appropriate provider service URI. With the Provider Analyser, we can also retrieve the URL of the provider service that has the most suitable metadata for the consumer's request. This allows the consumer to receive the URL of the provider service to which it can send GET requests.

In summary, the DTM receives both the translator class and the provider service message. These are sent to the translation node, where the provider's message is translated, provided that the message adheres to its corresponding metadata. As noted earlier, translation will always be successful if the provider's message format conforms to its own metadata.

The translation node in this project handles provider messages by executing the TAG translator.jar, using the files located in the output directory as its required parameters for execution. This is the final step in the workflow, which begins with obtaining the orchestration response and ends with the translation of the provider's XML message. However, merely placing the provider service XML message and the translator class into the translation node's directory is insufficient.

Firstly, the message that comes from the consumer's GET request to the provider is in JSON format, as Arrowhead Spring Boot based services are expected to work with JSON format. Therefore, a reformatting from JSON to XML of the messages from the GET request is required, this part is also a feature of the translation node.

Second, an additional step is required to execute the correct OS commands to run TAG translator.jar for the XML version. This step is analogous to when the TAG Tool is used to generate various JSON reports and translator classes. The script named ProviderTranslator.py, shown in Listing 3.10, is responsible for both the JSON-to-XML reformatting and executing the batch file to run TAG translator.jar. This script is executed after all previous scripts have completed running.

```python
import os
import json
import xml.etree.ElementTree as ET
import subprocess

def load_config(config_file):
    with open(config_file, 'r') as file:
        return json.load(file)

def json_to_xml(json_content):
    root = ET.Element("datavalues")

    for item in json_content:
        for key, value in item.items():
            if key != "id":
                if key == "unit":
                    key = "units"
                ET.SubElement(root, key).text = str(value)

    return ET.tostring(root, encoding='utf-8', xml_declaration=True).decode('utf-8')

def write_xml_file(xml_content, output_directory):
    output_file = os.path.join(output_directory, "provider.xml")
    with open(output_file, 'w') as file:
        file.write(xml_content)
    print(f"XML saved to {output_file}")

def run_batch_file(output_directory):
    batch_file = os.path.join(output_directory, "TAG-translator-xml.bat")

    result = subprocess.run(batch_file, cwd=output_directory, shell=True)

    if result.returncode == 0:
        print(f"Executed batch file: {batch_file}")
    else:
        print(f"Error executing batch file: {batch_file}, return code: {result.returncode}")

def main():
    config = load_config('config.json')
    output_directory = config.get("output_directory")
    input_directory = config.get("package_directory")

    provider_file = os.path.join(input_directory, "provider.txt")
    with open(provider_file, 'r') as file:
        provider_data = json.load(file)

    xml_content = json_to_xml(provider_data)
    write_xml_file(xml_content, output_directory)
    run_batch_file(output_directory)

if __name__ == "__main__":
    main()
```

Listing 3.10 - Method for converting messages and executing the Translation node of the DTM

This script represents the final step in the process of translating messages into the desired XML format.

3.4.8 Execution cycles

All the development phases detailed so far, allowed the DTM to, with just a consumer request for the providers' service data within the AF environment, choose the best provider for the service according to the Consumer's metadata requirements, always relying on the TAG Tool generated data to make this decision.

Now, having selected the most appropriate provider for the service and having its message translated, there is a necessity to make a cycle, that will allow the DTM to first, choose the provider it wants to communicate with, and secondly automatically translate its messages. Said cycle can be seen as an infinite loop, that keeps on re-executing in a defined number of seconds and is represented on Listing 4.2.

To start the consumer, the correct path would be to know the providers currently available and choose from that list, so upon starting the consumer, a method named "simpleMetadataRequest()" is called before anything else. This is the method that will create providers_info.txt. After the method returns the information file, the "runAll" batch is executed returning the translator class, and the URI for the best provider selected with ProviderAnaliser.py, if this is the first time running the consumer execution cycle, an interval of X seconds is made before reaching the next step in the loop. This is so the TAG Tool has enough time to run for the various providers available.

Now that communication with the selected provider can be started, a new method called "getFromProvider()" is executed every Y seconds to get the message from the provider, and the translation node is executed right after. The loop keeps executing and every (Y times 3) seconds the Data Translation Manager will be executed again to check if there are better providers available for the desired service, or simply to verify that the best provider that was previously selected is still available and doesn't need to be removed from the database.

This concludes the development process; we can now have a complex consumer system with an attached Data Translation Manager package that is able to search for and communicate with providers for as long as necessary.

4

TESTS AND VALIDATION

Chapter 4 focuses on the testing and validation process for the newly developed DTM system, particularly its integration with the Arrowhead Framework and the TAG Tool. The primary goal of this chapter is to demonstrate how the DTM system facilitates seamless communication between consumers and providers while ensuring data interoperability through the translation capabilities of the TAG Tool. Each section of the chapter addresses specific aspects of the system's functionality, from setting up the environment, validating provider selection processes, to ensuring successful metadata exchange between different systems.

For this chapter, the features implemented in the previous chapter will be validated by conducting a series of sequential tests. Each implemented step will be individually tested and validated by analysing the response logs from the system's execution and reviewing figures that showcase the results across different system interfaces.

The testing process begins with registering several providers in the Arrowhead cloud system, ensuring they are accurately detected and processed by the DTM system. This registration is followed by tests that evaluate the system's ability to dynamically select the most appropriate provider for the consumer based on the metadata and requirements. The final step of the testing cycle focuses on validating the message translation process, ensuring that the selected provider's data can be successfully translated.

Through detailed testing scenarios, the chapter will highlight the system's capacity to adapt in real-time to changes in provider availability, while also validating its ability to manage complex data transformation requirements. This ensures that the DTM system not only selects the best provider but also ensures interoperability between systems with differing metadata.

4.1 Arrowhead Setup

An Arrowhead cloud, in the Arrowhead Management Tool app, named "Nunes2.Afonso" was created for the purpose of deploying the Consumer and Providers. This Arrowhead cloud was created with the core services the AF demands, with Service Discovery being the exception as it is a part of the Orchestrator core service in this case.

After creating the cloud, it must be prepared to handle systems and services by ensuring that its core services are up and running. The three core services were therefore executed in sequence: first, the Service Registry (Figure 4.1), followed by the Orchestrator (Figure 4.2) and finally, the Authorization (Figure 4.3).

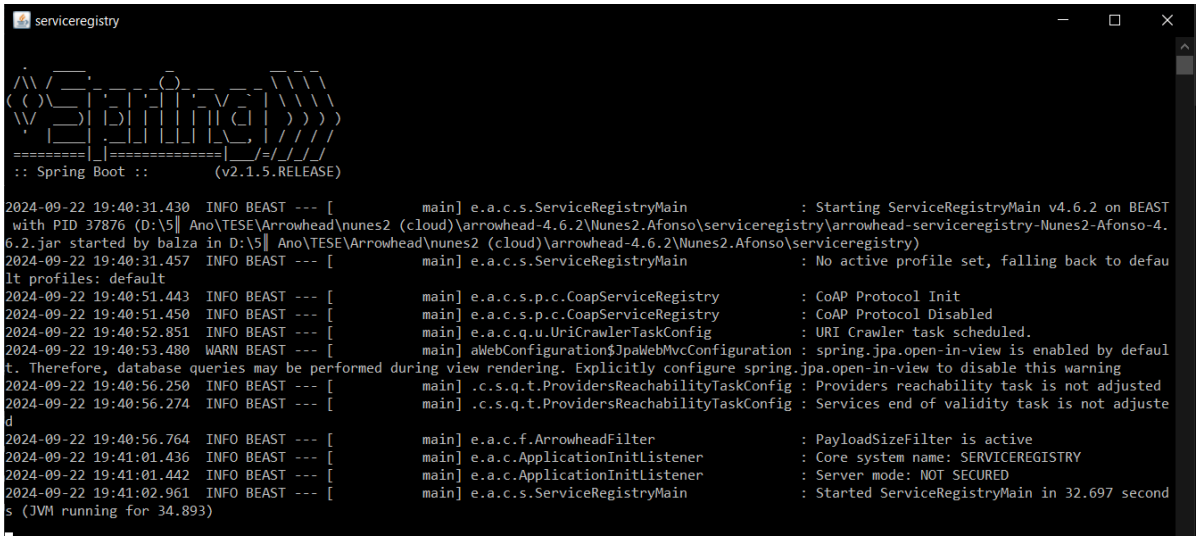


Figure 4.1 - Service Registry registration logs

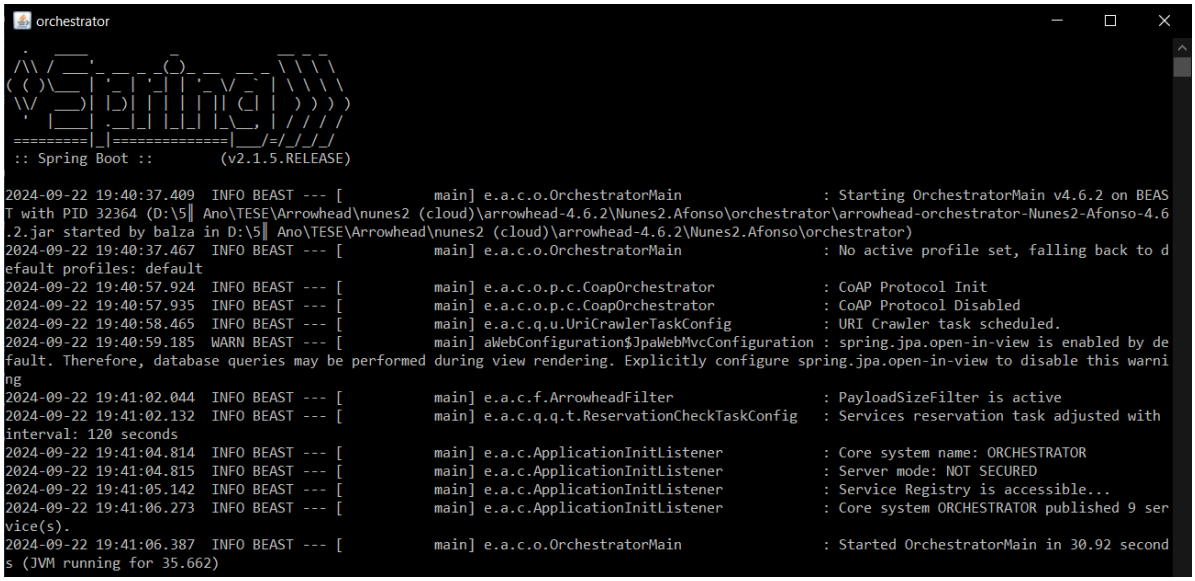


Figure 4.2 - Orchestrator registration logs

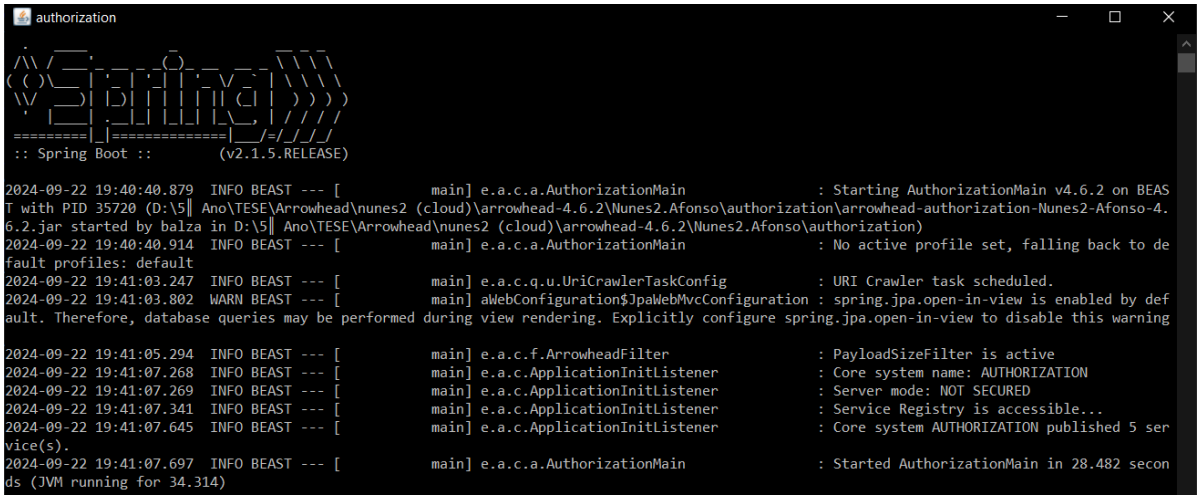


Figure 4.3 - Authorization registration logs

It is important to note that this cloud setup was completed in the early stages of the work and has been consistently used throughout the tests and development steps described earlier. However, this section specifically focuses on validating the cloud's availability and confirming that its core mandatory services are properly functioning.

4.2 Provider Registration

To initiate the testing process, four Providers were created and executed to be registered in the Arrowhead cloud GUI (Graphical user interface), as shown in Figure 4.5. For the example in this testing section, the providers offer sensor services. As previously mentioned, each provider

automatically registers its own system and available sensor services in the Arrowhead environment, as shown in Figure 3.4.

The console responses and the command used to run "Provider2" can be seen in Figure 4.4. All used providers have a similar registration or initialization process. The following Figure represents the initialization of "Provider2", not its registration, it had already been registered in the Arrowhead cloud in a previous development testing process.

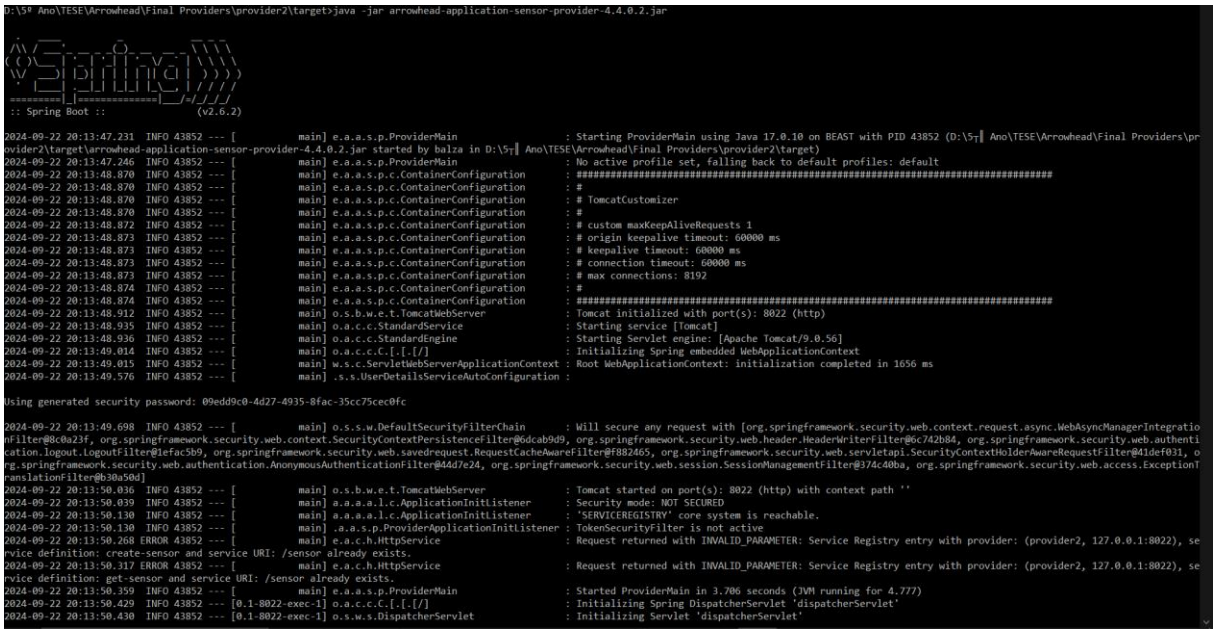


Figure 4.4 - Provider 2 initialization

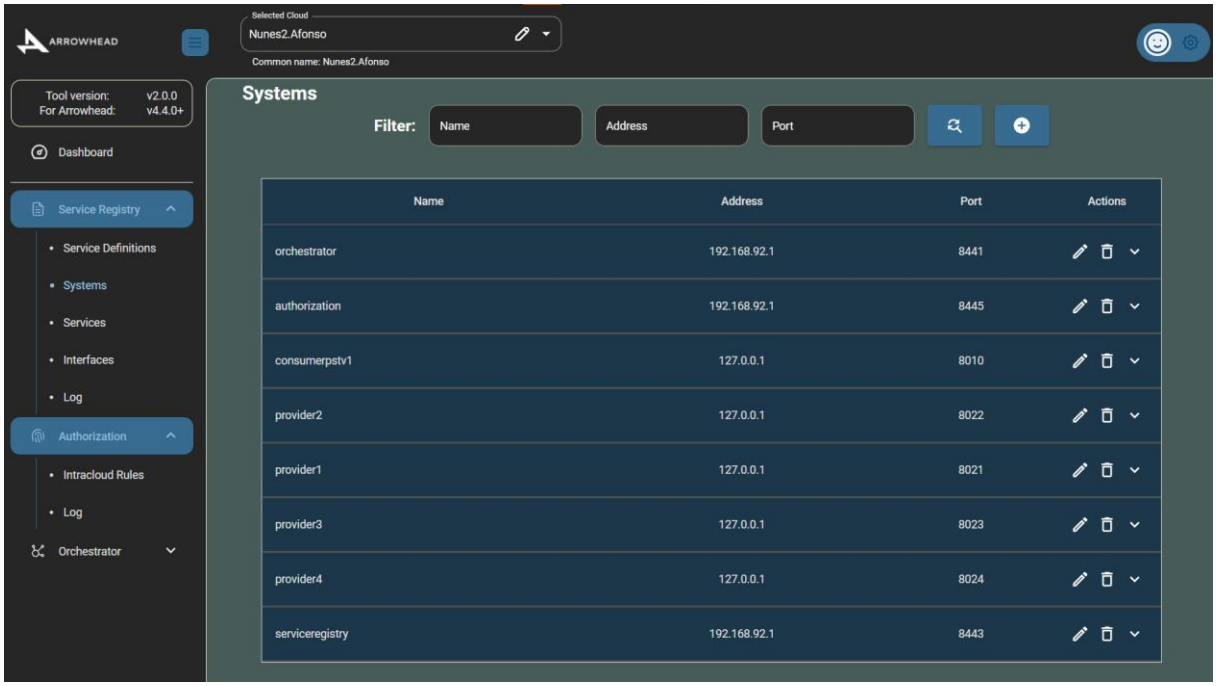


Figure 4.5 - Arrowhead Cloud's system dashboard

Once registered, these specific providers make an automatic POST request to their respective sensor service URIs, emulating a sensor reading. This process can be looped to mimic multiple sensor readings as if real-time data is being collected by a sensor. For simplicity in this demonstration, the same sensor reading is used repeatedly—in this case, temperature sensor readings. The results of these POST requests made to the providers' sensor service URIs are equal to those presented in the following Figures.

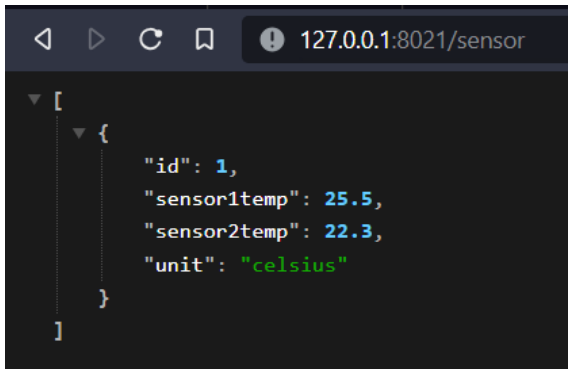


Figure 4.6 - Provider 1 "createSensor" URI

Figure 4.7 - Provider 2 "createSensor" URI

Figure 4.8 - Provider 3 "createSensor" URI

4.3 Consumer's Request (consumer-arrowhead)

The consumer system was manually registered, and intracloud rules were established to enable communication with the Arrowhead Orchestrator. These rules allowed the consumer to interact with the provider sensor services called "createSensor()" for each of the four providers, as illustrated in Figure 4.9.

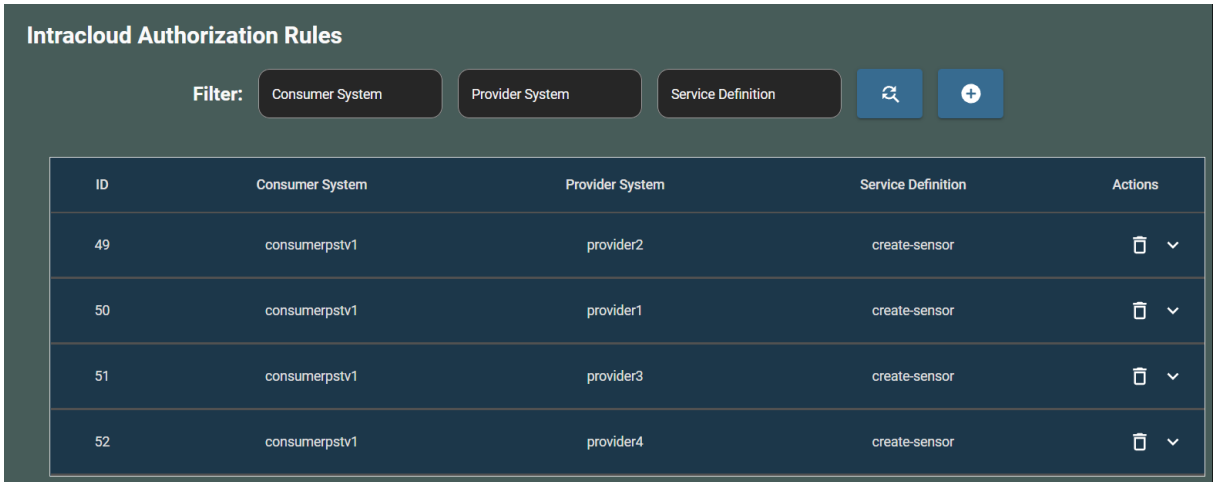


Figure 4.9 - Established intracloud rules

Next, the consumer calls the method "listConsumerMetadata()" to collect metadata from the providers' sensor services, generating a "provider_info.txt" file containing information on each provider. A snippet of this file's contents is shown in Listing 4.1, which was cut at line 37, where diverse Provider service information fields can be seen, such as its Provider system SR name (lines 5 and 35), the services URIs (also lines 5 and 35), its ontologies id (line 8) and finally its metadata schemas starting at line 10.

For simplicity and for the purposes of this test, the example providers' metadata is largely similar, with minor differences such as their message formats. The key difference between the providers' "createSensor()" services is the number of sensor readings they offer.

```
1. ------------------------------------------------------------------------------
2. providers for simpleRequest:
3. Provider Name        URL                                         Port
4. ------------------------------------------------------------------------------
5. provider2            127.0.0.1:8022//sensor                      8022
6.   Metadata:
7.      http-method         : POST
8.      ontology-id         : ontology-1
10.     xsd-schema1         :  <?xml version="1.0"?>
11. <xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:a3st="http://gres.uninova.pt/a3st"
xmlns:sawsdl="http://www.w3.org/ns/sawsdl">
12. <xs:element name="datavalues">
13. <xs:complexType>
14. <xs:sequence>
15. <xs:element type="xs:string" name="units" sawsdl:modelReference="/TemperatureSen-
sor{1;2}/hasTempUnits/TemperatureUnits{1}" a3st:mdi-ref="1;2"/>
16. <xs:element type="xs:float" name="sensor1temp" sawsdl:modelReference="/TemperatureSen-
sor{1}/hasDecValue" minOccurs="1"/>
17. <xs:element type="xs:float" name="sensor2temp" sawsdl:modelReference="/TemperatureSen-
sor{2}/hasDecValue" minOccurs="1"/>
18. <xs:element type="xs:byte" name="relay1state"/>
19. <xs:element type="xs:byte" name="relay2state"/>
20. </xs:sequence>
21. </xs:complexType>
22. </xs:element>
23. <xs:annotation>
24. <xs:appinfo>
```

```
25. <a3st:map-data-ind a3st:mdi-id="1" a3st:individual="Fahrenheit">F</a3st:map-data-ind>
26. <a3st:map-data-ind a3st:mdi-id="2" a3st:individual="Celsius">C</a3st:map-data-ind>
27. <a3st:comp-property-value type="xs:float" a3st:property="/TemperatureSensor{1}/hasLoca-
tion/Latitude{1}[hasLocUnits/LocationUnits{1}/hasIndividual/DD]/hasDecValue"
a3st:value="38.659963"/>
28. <a3st:comp-property-value type="xs:float" a3st:property="/TemperatureSensor{1}/hasLoca-
tion/Longitude{1}[hasLocUnits/LocationUnits{1}/hasIndividual/DD]/hasDecValue" a3st:value="-
9.203966"/>
29. <a3st:comp-property-value type="xs:float" a3st:property="/TemperatureSensor{2}/hasLoca-
tion/Latitude{2}[hasLocUnits/LocationUnits{1}/hasIndividual/DD]/hasDecValue"
a3st:value="38.659571"/>
30. <a3st:comp-property-value type="xs:float" a3st:property="/TemperatureSensor{2}/hasLoca-
tion/Longitude{2}[hasLocUnits/LocationUnits{1}/hasIndividual/DD]/hasDecValue" a3st:value="-
9.203929"/>
31. </xs:appinfo>
32. </xs:annotation>
33. </xs:schema>
34.
35. provider3           127.0.0.1:8023//sensor                  8024
36.  Metadata: (…)
```

Listing 4.1- Provider information file contents

In Listing 4.2, a snippet of the main cycle from the consumer's main class is provided for visual reference.

```
 1.     @Override
 2.     public void run(final ApplicationArguments args) throws Exception {
 3.         while(true){
 4.             if (k == 0)
 5.             {
 6.                 listConsumerMetadata();
 7.             }
 8.             if (k % 3 == 0)
 9.             {
10.             simpleMetadataRequest();
11.             try {
12.                 Runtime.getRuntime().exec("cmd /c start \"\" " + runAllBatPath);
13.                 } catch (IOException e) {}
14.                 //Only in the first time since the info was not stored in the database yet
15.                 if (firstTime == 1){
16.                     Thread.sleep(10000);
17.                     firstTime = 0;
18.                 }
19.             }
20.         getFromUri(selectedProviderUrlFile);
21.         //RUN TRANSLATION NODE HERE
22.         try {
23.             Runtime.getRuntime().exec("cmd /c start \"\" " + translateBatPath);
24.             } catch (IOException e) {}
25.
26.         k = k + 1;
27.         Thread.sleep(3000);
28.         }
```

Listing 4.2 - Consumer execution cycle

4.4 Consumer-DTM Interaction and results

It is important to note that none of the provider services in this test could communicate directly with the consumer using just the Arrowhead services. This is because the message schemas of the providers do not exactly match the consumer's schema. The DTM selection process and the use of the TAG tool enable communication by bridging this gap.

After retrieving the provider and consumer metadata and storing them in their respective files, the "runAll" batch file is executed. The TAG tool transitions from not having it's necessary inputs to containing the provider and consumer schema files, as shown in Figure 4.10. The following CMD logs in Listing 4.3 show the program's results.

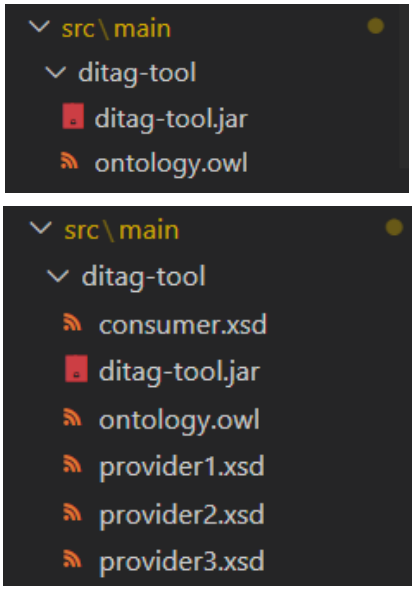


Figure 4.10 - TAG directory contents, before and after

Since only three sensor services are online at this stage of testing, it is expected that only three will be detected and processed by TAG. The tool generates reports and translators for these services, which are then stored in the DTM database. For easier visualization, large files are cut into snippets. The results stored in the database are displayed in Figure 4.11.

```
Filtered and formatted consumer XSD saved to D:/TranslatorTool/consumerMetadaTAG2/src/main/TAG-
tool\consumer.xsd
Processing provider: providers with URL: for
Processing provider: provider2 with URL: 127.0.0.1:8022//sensor
Generated provider2.xsd
Processing provider: provider1 with URL: 127.0.0.1:8021//sensor
Generated provider1.xsd
Processing provider: provider3 with URL: 127.0.0.1:8023//sensor
Generated provider3.xsd
```

```
Processed Providers: []
Unprocessed Providers: ['127.0.0.1:8022//sensor', '127.0.0.1:8021//sensor',
'127.0.0.1:8023//sensor']
Providers with changed metadata: []
Running TAG-tool for provider with URL 127.0.0.1:8022//sensor
TAG-tool output: {annotation=/TemperatureSensor/hasTempUnits/TemperatureUnits/hasIndividual/Cel-
sius, path=/datavalues[1,1], name=units}=[{annotation=/TemperatureSensor/hasTempUnits/Tempera-
tureUnits/hasIndividual/Celsius, path=/datavalues[1,1], name=sensor1temp}]
{annotation=/TemperatureSensor{1}/hasDecValue, path=/datavalues[1,1], name=sensor1temp}=[{anno-
tation=/TemperatureSensor{1}/hasDecValue, path=/datavalues[1,1], name=sensor1temp}]
{annotation=/TemperatureSensor{2}/hasDecValue, path=/datavalues[1,1], name=sensor2temp}=[{anno-
tation=/TemperatureSensor{2}/hasDecValue, path=/datavalues[1,1], name=sensor2temp}]
{annotation=/TemperatureSensor{3}/hasDecValue, path=/datavalues[1,1], name=sensor3temp}=[]
{annotation=/TemperatureSensor{4}/hasDecValue, path=/datavalues[1,1], name=sensor4temp}=[]

Running TAG-tool for provider with URL 127.0.0.1:8021//sensor
TAG-tool output: {annotation=/TemperatureSensor/hasTempUnits/TemperatureUnits/hasIndividual/Cel-
sius, path=/datavalues[1,1], name=units}=[{annotation=/TemperatureSensor/hasTempUnits/Tempera-
tureUnits/hasIndividual/Celsius, path=/datavalues[1,1], name=sensor1temp}]
{annotation=/TemperatureSensor/hasTempUnits/TemperatureUnits/hasIndividual/Fahrenheit,
path=/datavalues[1,1], name=units}=[{annotation=/TemperatureSensor/hasTempUnits/Temperature-
Units/hasIndividual/Fahrenheit, path=/datavalues[1,1], name=units}]
{annotation=/TemperatureSensor{1}/hasDecValue, path=/datavalues[1,1], name=sensor1temp}=[{anno-
tation=/TemperatureSensor{1}/hasDecValue, path=/datavalues[1,1], name=sensor1temp}]
{annotation=/TemperatureSensor{2}/hasDecValue, path=/datavalues[1,1], name=sensor2temp}=[{anno-
tation=/TemperatureSensor{2}/hasDecValue, path=/datavalues[1,1], name=sensor2temp}]
{annotation=/TemperatureSensor{3}/hasDecValue, path=/datavalues[1,1], name=sensor3temp}=[]
{annotation=/TemperatureSensor{4}/hasDecValue, path=/datavalues[1,1], name=sensor4temp}=[]

Running TAG-tool for provider with URL 127.0.0.1:8023//sensor
TAG-tool output: {annotation=/TemperatureSensor/hasTempUnits/TemperatureUnits/hasIndividual/Cel-
sius, path=/datavalues[1,1], name=units}=[{annotation=/TemperatureSensor/hasTempUnits/Tempera-
tureUnits/hasIndividual/Celsius, path=/datavalues[1,1], name=sensor1temp}]
{annotation=/TemperatureSensor/hasTempUnits/TemperatureUnits/hasIndividual/Fahrenheit,
path=/datavalues[1,1], name=units}=[{annotation=/TemperatureSensor/hasTempUnits/Temperature-
Units/hasIndividual/Fahrenheit, path=/datavalues[1,1], name=units}]
{annotation=/TemperatureSensor{1}/hasDecValue, path=/datavalues[1,1], name=sensor1temp}=[{anno-
tation=/TemperatureSensor{1}/hasDecValue, path=/datavalues[1,1], name=sensor1temp}]
{annotation=/TemperatureSensor{2}/hasDecValue, path=/datavalues[1,1], name=sensor2temp}=[{anno-
tation=/TemperatureSensor{2}/hasDecValue, path=/datavalues[1,1], name=sensor2temp}]
{annotation=/TemperatureSensor{3}/hasDecValue, path=/datavalues[1,1], name=sensor3temp}=[{anno-
tation=/TemperatureSensor{3}/hasDecValue, path=/datavalues[1,1], name=sensor3temp}]
{annotation=/TemperatureSensor{4}/hasDecValue, path=/datavalues[1,1], name=sensor4temp}=[]

Processing provider: provider2.xsd with URL: 127.0.0.1:8022//sensor
Looking for report file at: D:/TranslatorTool/consumerMetadaTAG2/src/main/TAG-tool\re-
port_127.0.0.1_8022_sensor.json
Looking for translator file at: D:/TranslatorTool/consumerMetadaTAG2/src/main/TAG-tool\Transla-
tor_127.0.0.1_8022_sensor.java
Loaded report content for 127.0.0.1_8022_sensor
Loaded translator content for 127.0.0.1_8022_sensor
Deleted report file for 127.0.0.1_8022_sensor
Deleted translator file for 127.0.0.1_8022_sensor
Deleted HTML report file for 127.0.0.1_8022_sensor
Processing provider: provider1.xsd with URL: 127.0.0.1:8021//sensor
Looking for report file at: D:/TranslatorTool/consumerMetadaTAG2/src/main/TAG-tool\re-
port_127.0.0.1_8021_sensor.json
Looking for translator file at: D:/TranslatorTool/consumerMetadaTAG2/src/main/TAG-tool\Transla-
tor_127.0.0.1_8021_sensor.java
Loaded report content for 127.0.0.1_8021_sensor
Loaded translator content for 127.0.0.1_8021_sensor
Deleted report file for 127.0.0.1_8021_sensor
Deleted translator file for 127.0.0.1_8021_sensor
Deleted HTML report file for 127.0.0.1_8021_sensor
Processing provider: provider3.xsd with URL: 127.0.0.1:8023//sensor
Looking for report file at: D:/TranslatorTool/consumerMetadaTAG2/src/main/TAG-tool\re-
port_127.0.0.1_8023_sensor.json
```

```
Looking for translator file at: D:/TranslatorTool/consumerMetadaTAG2/src/main/TAG-tool\Transla-
tor_127.0.0.1_8023_sensor.java
Loaded report content for 127.0.0.1_8023_sensor
Loaded translator content for 127.0.0.1_8023_sensor
Deleted report file for 127.0.0.1_8023_sensor
Deleted translator file for 127.0.0.1_8023_sensor
Deleted HTML report file for 127.0.0.1_8023_sensor
Database updated. Missing providers deleted.
Running ProviderAnalyzer.py...
Report for provider 'provider2.xsd' has 3 elements with 'provided' value '1'.
Report for provider 'provider1.xsd' has 4 elements with 'provided' value '1'.
Report for provider 'provider3.xsd' has 5 elements with 'provided' value '1'.

Provider 'provider3.xsd' with URL '127.0.0.1:8023//sensor' has the maximum number of elements
with 'provided' value '1': 5
Deleted old file: Translator.java
Selected provider URL written to D:/TranslatorTool/consumerMetadaTAG2/src/main/TAG-transla-
tor\selected_provider.txt
Translator content written to D:/TranslatorTool/consumerMetadaTAG2/src/main/TAG-transla-
tor\Translator.java
Removed 'package generated;' lines from D:/TranslatorTool/consumerMetadaTAG2/src/main/TAG-trans-
lator\Translator.java
```

Listing 4.3 - Console Logs for DTM's first execution in this testing section

Provider Data Table

Provider URL	Provider File Name	Provider Content (Snippet)	Report Content (Snippet)	Translator Content (Snippet)
127.0.0.1:8023//sensor	provider3.xsd	<?xml version="1.0"?> <xs:schema attributeForm…	{"provider":{"TemperatureSensor{4}":[{"annotati…	package pt.uninova.ditag.translator; import com.fa…
127.0.0.1:8022//sensor	provider2.xsd	<?xml version="1.0"?> <xs:schema attributeForm…	{"provider":{"TemperatureSensor{4}":[{"annotati…	package pt.uninova.ditag.translator; import com.fa…
127.0.0.1:8021//sensor	provider1.xsd	<?xml version="1.0"?> <xs:schema attributeForm…	{"provider":{"TemperatureSensor{4}":[{"annotati…	package pt.uninova.ditag.translator; import com.fa…

Figure 4.11 - Database Contents

As expected, only the three active providers were detected and stored in the database, despite the fact that authorization rules had been set up for four providers. This occurred because a method was developed for the consumer to check whether a provider is actually running. If relying solely on Arrowhead, once the intracloud rules are established as seen in Figure 4.9, the metadata for all four providers would have been registered in the information file, regardless of their online status.

4.5 DTM Translation Node

As seen Listing 4.3 above, Provider 3 was selected in this test since it fulfilled five of the consumer's requirements, more than Providers 1 and 2. The Consumer can now access Provider 3's URI using the "getFromURI()" method, which retrieves sensor readings every 3 seconds. After each message is gathered, the Translation Node executes using the "Translate" batch file which executes de automatically generated Java Translator.

Listing 4.4 shows the response from running the Translation Node, which results in translated messages in both XML and JSON formats. In Figure 4.12, the original XML message made from the GET request to a provider, in Figure 4.13 the translated results are displayed. Logs from the Arrowhead Consumer are also included in Figure 4.14.

```
Running ProviderTranslator.py...
XML saved to D:/TranslatorTool/consumerMetadaTAG2/src/main/TAG-translator\provider.xml
Executed batch file: D:/TranslatorTool/consumerMetadaTAG2/src/main/TAG-translator\TAG-transla-
tor-xml.bat
```

Listing 4.4 - Console log responses for executing the translator

```
<?xml version='1.0' encoding='utf-8'?>
<datavalues>
    <sensor1temp>25.5</sensor1temp>
    <sensor2temp>22.3</sensor2temp>
    <sensor3temp>23.3</sensor3temp>
    <units>cel</units>
</datavalues>
```

Figure 4.12 - Original Provider message in XML format

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<datavalues>
    <units>Cel</units>
    <sensor1>25.5</sensor1>
    <sensor2>22.3</sensor2>
    <sensor3>23.3</sensor3>
</datavalues>
```

```
{
    "units": "Cel",
    "sensor1": 25.5,
    "sensor2": 22.3,
    "sensor3": 23.3
}
```

Figure 4.13 - Translated messages for XML and JSON, respectively

```
Performing GET request to URI: 127.0.0.1:8023//sensor
--------------------------------------------------------------------------------
GET request to 127.0.0.1:8023//sensor
Response: [{"id":1,"sensor1temp":25.5,"sensor2temp":22.3,"sensor3temp":23.3,"unit":"cel"}]
--------------------------------------------------------------------------------
Unfiltered JSON saved successfully to provider.txt
```

Figure 4.14 - GET request to the Selected Provider's Service

4.6 Cycle Repetition Testing

This section describes the Cycle Repetition Testing for the DTM system, focusing on its operation during different scenarios that impact provider availability and selection. The execution loop for the entire DTM system can be seen in Listing 4.2. Every nine seconds (3000 milliseconds times three), the DTM system is triggered to check for new providers or changes in the status of existing ones. This ensures that the Consumer is always establishing updated connections with the best provider that is currently online, maintaining real-time responsiveness. The tests verify that the DTM system continually reacts to provider status changes, keeping the connections between consumers and providers optimized and updated.

4.6.1 No new Providers available

In this scenario, all the previously online providers stay online, and no new ones come online. This means the database already has all the necessary information about the currently online Providers. The DTM system already has up-to-date information; therefore, it does not trigger another TAG execution.

Listing 4.5 showcases the logs for this scenario.

```
Running ProviderConsumerManager.py...
Filtered and formatted consumer XSD saved to D:/TranslatorTool/consumerMetadaTAG2/src/main/TAG-
tool\consumer.xsd
Processing provider: providers with URL: for
Processing provider: provider2 with URL: 127.0.0.1:8022//sensor
Generated provider2.xsd
Processing provider: provider1 with URL: 127.0.0.1:8021//sensor
Generated provider1.xsd
Processing provider: provider3 with URL: 127.0.0.1:8023//sensor
Generated provider3.xsd
Processed Providers: ['127.0.0.1:8022//sensor', '127.0.0.1:8021//sensor', '127.0.0.1:8023//sen-
sor']
Unprocessed Providers: []
Providers with changed metadata: []
Database updated. Missing providers deleted.
Running ProviderAnalyzer.py...
Report for provider 'provider2.xsd' has 3 elements with 'provided' value '1'.
Report for provider 'provider1.xsd' has 4 elements with 'provided' value '1'.
Report for provider 'provider3.xsd' has 5 elements with 'provided' value '1'.

Provider 'provider3.xsd' with URL '127.0.0.1:8023//sensor' has the maximum number of elements
with 'provided' value '1': 5
Deleted old file: Translator.java
Selected provider URL written to D:/TranslatorTool/consumerMetadaTAG2/src/main/TAG-transla-
tor\selected_provider.txt
Translator content written to D:/TranslatorTool/consumerMetadaTAG2/src/main/TAG-transla-
tor\Translator.java
Removed 'package generated;' lines from D:/TranslatorTool/consumerMetadaTAG2/src/main/TAG-trans-
lator\Translator.java
```

Listing 4.5 - Logs for no new providers scenario

As expected, the providers' service data, both from DTM and Arrowhead, was already stored in the database. This confirms that the providers were registered as "Processed Providers" with unchanged metadata, and thus, the TAG execution was skipped. Provider 3 remained the selected provider, saving significant execution time.

4.6.2 New Provider Available

If a new Provider 4 were to go online, it would be detected as a new unprocessed provider and the TAG would execute a single time, resulting in a new provider being added to the database and the Processed Providers list. The logs would be those in
Listing 4.6 below:

```
Running ProviderConsumerManager.py...
Filtered and formatted consumer XSD saved to D:/TranslatorTool/consumerMetadaTAG2/src/main/TAG-
tool\consumer.xsd
Processing provider: providers with URL: for
Processing provider: provider2 with URL: 127.0.0.1:8022//sensor
Generated provider2.xsd
Processing provider: provider1 with URL: 127.0.0.1:8021//sensor
Generated provider1.xsd
Processing provider: provider3 with URL: 127.0.0.1:8023//sensor
Generated provider3.xsd
Processing provider: provider4 with URL: 127.0.0.1:8024//sensor
Generated provider4.xsd
Processed Providers: ['127.0.0.1:8022//sensor', '127.0.0.1:8021//sensor', '127.0.0.1:8023//sen-
sor']
Unprocessed Providers: ['127.0.0.1:8024//sensor']
Providers with changed metadata: []
Running TAG-tool for provider with URL 127.0.0.1:8024//sensor
Error running TAG-tool for provider 127.0.0.1:8024//sensor: log4j:WARN No appenders could be
found for logger (com.hp.hpl.jena.util.FileManager).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "pt.un-
inova.TAG.tool.schema.SchemaNode.getValue(String)" because the return value of "pt.un-
inova.TAG.tool.schema.SchemaNode.getMatch()" is null
        at pt.uninova.TAG.tool.translator.TranslatorCreator.generateIndividialPairings(Transla-
torCreator.java:90)
        at pt.uninova.TAG.tool.translator.TranslatorCreator.<init>(TranslatorCreator.java:78)
        at pt.uninova.TAG.tool.SemanticEngine.runEngine(SemanticEngine.java:103)
        at pt.uninova.TAG.tool.Executor.run(Executor.java:121)
        at pt.uninova.TAG.tool.Executor.main(Executor.java:84)

Processing provider: provider4.xsd with URL: 127.0.0.1:8024//sensor
Looking for report file at: D:/TranslatorTool/consumerMetadaTAG2/src/main/TAG-tool\re-
port_127.0.0.1_8024_sensor.json
Looking for translator file at: D:/TranslatorTool/consumerMetadaTAG2/src/main/TAG-tool\Transla-
tor_127.0.0.1_8024_sensor.java
Report file not found for 127.0.0.1_8024_sensor
Translator file not found for 127.0.0.1_8024_sensor
Database updated. Missing providers deleted.
Running ProviderAnalyzer.py...
Report for provider 'provider2.xsd' has 3 elements with 'provided' value '1'.
Report for provider 'provider1.xsd' has 4 elements with 'provided' value '1'.
Report for provider 'provider3.xsd' has 5 elements with 'provided' value '1'.
```

```
Provider 'provider3.xsd' with URL '127.0.0.1:8023//sensor' has the maximum number of elements
with 'provided' value '1': 5
Deleted old file: Translator.java
Selected provider URL written to D:/TranslatorTool/consumerMetadaTAG2/src/main/TAG-transla-
tor\selected_provider.txt
Translator content written to D:/TranslatorTool/consumerMetadaTAG2/src/main/TAG-transla-
tor\Translator.java
Removed 'package generated;' lines from D:/TranslatorTool/consumerMetadaTAG2/src/main/TAG-trans-
lator\Translator.java
```

Listing 4.6 - Logs for one new provider scenario

As confirmed, only one new provider was detected, listed under "Unprocessed Providers," meaning only a single execution of the TAG was required. However, this provider's metadata could not be processed by the TAG because it did not meet the minimum consumer requirements. As a result, the outcome remained unchanged, with Provider 3 still being the best option. In Figure 4.15, the updated database is displayed, showing empty fields for the report and translator associated with Provider 4, as the TAG was unable to generate them.

Provider Data Table

Provider URL	Provider File Name	Provider Content (Snippet)	Report Content (Snippet)	Translator Content (Snippet)
127.0.0.1:8022//sensor	provider2.xsd	<?xml version="1.0"?> <xs:schema attributeForm…	{"provider":{"TemperatureSensor{4}":[{"annotati…	package pt.uninova.ditag.translator; import com.fa…
127.0.0.1:8021//sensor	provider1.xsd	<?xml version="1.0"?> <xs:schema attributeForm…	{"provider":{"TemperatureSensor{4}":[{"annotati…	package pt.uninova.ditag.translator; import com.fa…
127.0.0.1:8023//sensor	provider3.xsd	<?xml version="1.0"?> <xs:schema attributeForm…	{"provider":{"TemperatureSensor{4}":[{"annotati…	package pt.uninova.ditag.translator; import com.fa…
127.0.0.1:8024//sensor	provider4.xsd	<?xml version="1.0"?> <xs:schema attributeForm…		

Figure 4.15 - Updated database contents

As expected, GET requests to the Provider 3 service endpoint are still made multiple times between each DTM cycle.

4.6.3 Provider goes offline

Now for a final testing scenario, Provider 3 is turned off, and the results should establish a connection to the newly selected best provider for the sensor service, either Provider 1 or Provider 2, as Provider 4 fails to meet the requirements.

Listing 4.7 shows the logs, and Figure 4.16 is an image of the updated database storage.

```
 1. Running ProviderConsumerManager.py...
 2. Filtered and formatted consumer XSD saved to D:/TranslatorTool/consumerMetada-
TAG2/src/main/TAG-tool\consumer.xsd
 3. Processing provider: providers with URL: for
 4. Processing provider: provider2 with URL: 127.0.0.1:8022//sensor
 5. Generated provider2.xsd
 6. Processing provider: provider1 with URL: 127.0.0.1:8021//sensor
 7. Generated provider1.xsd
 8. Processing provider: provider4 with URL: 127.0.0.1:8024//sensor
 9. Generated provider4.xsd
10. Processed Providers: ['127.0.0.1:8022//sensor', '127.0.0.1:8021//sensor',
'127.0.0.1:8023//sensor', '127.0.0.1:8024//sensor']
11. Unprocessed Providers: []
12. Providers with changed metadata: []
13. Deleting provider with URL 127.0.0.1:8023//sensor from the database
14. Database updated. Missing providers deleted.
15. Running ProviderAnalyzer.py...
16. Report for provider 'provider2.xsd' has 3 elements with 'provided' value '1'.
17. Report for provider 'provider1.xsd' has 4 elements with 'provided' value '1'.
18. Provider 'provider1.xsd' with URL '127.0.0.1:8021//sensor' has the maximum number of ele-
ments with 'provided' value '1': 4
19. Deleted old file: Translator.java
20. Selected provider URL written to D:/TranslatorTool/consumerMetadaTAG2/src/main/TAG-transla-
tor\selected_provider.txt
21. Translator content written to D:/TranslatorTool/consumerMetadaTAG2/src/main/TAG-transla-
tor\Translator.java
22. Removed 'package generated;' lines from D:/TranslatorTool/consumerMetadaTAG2/src/main/TAG-
translator\Translator.java
```

Listing 4.7 - Logs for one provider is shut down scenario

Provider Data Table

Provider URL	Provider File Name	Provider Content (Snippet)	Report Content (Snippet)	Translator Content (Snippet)
127.0.0.1:8022//sensor	provider2.xsd	<?xml version="1.0"?> <xs:schema attributeForm…	{"provider":{"TemperatureSensor{4}":[{"annotati…	package pt.uninova.ditag.translator; import com.fa…
127.0.0.1:8021//sensor	provider1.xsd	<?xml version="1.0"?> <xs:schema attributeForm…	{"provider":{"TemperatureSensor{4}":[{"annotati…	package pt.uninova.ditag.translator; import com.fa…
127.0.0.1:8024//sensor	provider4.xsd	<?xml version="1.0"?> <xs:schema attributeForm…		

Figure 4.16 - Database contents for one provider is shut down scenario

The expected results are backed up by the logs and the new provider storage present in the database. Provider 3 is no longer being considered or stored, and Provider 1 is now the selected one. GET requests are now being made to the Provider 1 sensor service URI, as shown in Figure 4.17, alongside the Arrowhead Consumer response.

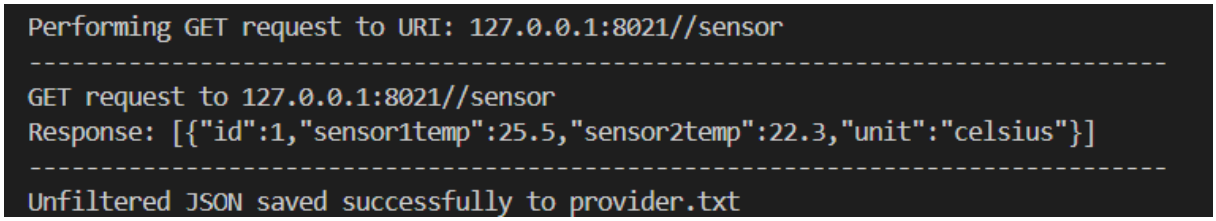


Figure 4.17 - GET request to Provider 1 service response

It is important to clarify that the use case scenarios described in Chapter 3 represent potential real-world applications that could be explored in larger-scale projects, provided the necessary technological and human resources are available. However, these were not the focus of the testing conducted in this chapter.

Instead, the goal here was to demonstrate how the TAG Tool and Arrowhead Framework can interact through the Data Translation Manager (DTM) and how the DTM effectively selects the most suitable service provider for a given consumer. The step-by-step testing of these functionalities confirms that the initial objectives have been successfully met. Ultimately, this work has contributed to bridging different technologies aimed at improving interoperability as well as executing translations, thus enhancing system integration and communication across heterogeneous environments.

5

Conclusions

This thesis set out to address key challenges in system interoperability, particularly in the context of Industry 4.0. The introduction highlighted the growing need for seamless communication between heterogeneous systems across different platforms, focusing on both data and semantic interoperability. Through the integration of the TAG Tool and the Arrowhead Framework, this work aimed to create a system that not only ensures compatibility at the service level but also translates data to facilitate data exchange between previously data incompatible consumer and provider systems.

One of the primary objectives was the integration of the TAG Tool and the Arrowhead Framework, which was successfully achieved through the development of the DTM (Data Translation Manager). This integration uplifted the possibility for communication between the systems registered within the Arrowhead cloud. The DTM was responsible for bridging service and data interoperability by managing the connection between Arrowhead's core services and the TAG Tool's data translation capabilities.

The testing process confirmed that the DTM could gather Consumer and Providers metadata from the arrowhead orchestrator responses and execute the TAG Tool to handle the data operations verifying compatibility and generating translators. This allowed systems with differing data structures to exchange information effectively, meeting the objective of enabling data and semantic interoperability. The successful implementation of this integration lays the foundation for more advanced interoperability tools, improving the flexibility and adaptability of systems operating in heterogeneous environments.

The second key goal was to ensure that the DTM could identify and select the most suitable provider for a given consumer in the Arrowhead system. This feature was crucial for

optimizing the communication pathways between systems, particularly when multiple providers were available with varying capabilities. By analysing both the required and optional elements available from each provider, the DTM was able to establish the most appropriate pairing according to the TAG-Tool reports, ensuring that the consumer system received the most data possible. The cyclic nature of the DTM also allowed the system to continuously monitor changes in provider availability, ensuring that the consumer always remained connected to the best possible option. This dynamic adaptation successfully met the objectives of enhancing provider selection in real time.

Looking ahead, there is still room for further improvement and development of the new features for the DTM. In real life IoT scenario where there could be hundreds of consumers and providers all trying to access the DTM for executing the translation part of the TAG, a bottleneck situation could emerge. Therefore, in the future it would be an improvement if the translator class for each provider could granted to the consumer system itself. This way there would be no necessity to use the DTM and TAG every time a GET request from the consumer to said selected provider would be made, the consumer system would just be able to execute the translator class within this process, reliving the number of requests being made to the DTM for translation.

Additionally, an ontology management system backed by a database could be introduced, similar to the existing setup for the providers' information. This implementation would register the ontology name from the "ontology-Id" metadata field for each provider and have a database containing two columns: one for the matching ontology ID and another for the corresponding OWL file. If a matching ontology name registered in the providers' information file is found in the database, the corresponding OWL file would be sent to the TAG Tool node for execution for that same provider.

REFERENCES

[1] Z. Suleiman, S. Shaikholla, D. Dikhanbayeva, E. Shehab, and A. Turkyilmaz, "Industry 4.0: Clustering of concepts and characteristics," *Cogent Eng*, vol. 9, no. 1, Dec. 2022, doi: 10.1080/23311916.2022.2034264.

[2] T. Burns, J. Cosgrove, and F. Doyle, "A Review of Interoperability Standards for Industry 4.0.," *Procedia Manuf*, vol. 38, pp. 646–653, 2019, doi: 10.1016/j.promfg.2020.01.083.

[3] J. Jokinen, T. Latvala, and J. L. Martinez Lastra, "Integrating smart city services using Arrowhead framework," in *IECON 2016 - 42nd Annual Conference of the IEEE Industrial Electronics Society*, IEEE, Oct. 2016, pp. 5568–5573. doi: 10.1109/IECON.2016.7793708.

[4] Peiru Teo, "Healthcare Interoperability: The Benefits, Challenges and Solutions." Accessed: Jan. 03, 2024. [Online]. Available: https://www.keyreply.com/blog/healthcare-interoperability

[5] "Eclipse Arrowhead™ A framework and implementation platform for SoS, IoT and OT integration." Accessed: Dec. 10, 2023. [Online]. Available: https://arrowhead.eu/eclipse-arrowhead-2/

[6] F. Moutinho, L. Paiva, J. Kopke, and P. Malo, "Extended Semantic Annotations for Generating Translators in the Arrowhead Framework," *IEEE Trans Industr Inform*, vol. 14, no. 6, pp. 2760–2769, Jun. 2018, doi: 10.1109/TII.2017.2780887.

[7] G. Amaro, F. Moutinho, R. Campos-Rebelo, J. Köpke, and P. Maló, "JSON Schemas with Semantic Annotations Supporting Data Translation," *Applied Sciences*, vol. 11, no. 24, p. 11978, Dec. 2021, doi: 10.3390/app112411978.

[8] F. Chauvel, G. Hu, and L. Mei, "Dynamic interoperability between heterogeneous services," in *Proceedings of the 2011 international workshop on Networking and object memories for the internet of things*, New York, NY, USA: ACM, Sep. 2011, pp. 7–8. doi: 10.1145/2029932.2029935.

[9] Martín Serrano, Philippe Cousin, Francois Carrez, Payam Barnaghi, Ovidiu Vermesan, and Peter Friess, "Internet of Things IoT Semantic Interoperability: Research Challenges, Best Practices, Recommendations and Next Steps EUROPEAN RESEARCH CLUSTER ON THE INTERNET OF THINGS," Apr. 2015. [Online]. Available: https://www.egm.io/wp-

content/uploads/2020/08/2015-03-IoT-Semantic-Interoperability-Research-Challenges-Best-Practices-Recommendations-and-Next-Steps.pdf

[10] P. Miguel Negrão Maló "Hub-and-Spoke Interoperability: An out of the skies approach for large-scale data interoperability Dissertação para obtenção do Grau de Doutor em Engenharia Electrotécnica e de Computadores," 2013. Accessed: Jan. 15, 2024. [Online]. Available: https://run.unl.pt/handle/10362/11397

[11] F. Moutinho, L. Paiva, P. Malo, and L. Gomes, "Semantic annotation of data in schemas to support data translations," in *IECON 2016 - 42nd Annual Conference of the IEEE Industrial Electronics Society*, IEEE, Oct. 2016, pp. 5283–5288. doi: 10.1109/IECON.2016.7793691.

[12] F. Blomstedt *et al.*, "The arrowhead approach for SOA application development and documentation," in *IECON 2014 - 40th Annual Conference of the IEEE Industrial Electronics Society*, IEEE, Oct. 2014, pp. 2631–2637. doi: 10.1109/IECON.2014.7048877.

[13] Oyekunle Claudius Oyeniran, Adebunmi Okechukwu Adewusi, Adams Gbolahan Adeleke, Lucy Anthony Akwawa, and Chidimma Francisca Azubuko, "Microservices architecture in cloud-native applications: Design patterns and scalability," *Computer Science & IT Research Journal*, vol. 5, no. 9, pp. 2107–2124, Sep. 2024, doi: 10.51594/csitrj.v5i9.1554.

[14] borditamas, "GitHub - arrowhead-f/sos-examples-spring." accessed at https://github.com/arrowhead-f/sos-examples-spring

[15] K. S. Prasad Reddy, *Beginning Spring Boot 2*. Berkeley, CA: Apress, 2017. doi: 10.1007/978-1-4842-2931-6.

[16] S. Campbell, "What Type of Data Is XML?" Accessed: Jan. 24, 2024. [Online]. Available: https://serverlogic3.com/what-type-of-data-is-xml

[17] "Introducing Linked Data And The Semantic Web." Accessed: Jan. 20, 2024. [Online]. Available: https://linkeddatatools.com/semantic-web-basics/

[18] G. Wang, "Improving Data Transmission in Web Applications via the Translation between XML and JSON," in *2011 Third International Conference on Communications and Mobile Computing*, IEEE, Apr. 2011, pp. 182–185. doi: 10.1109/CMC.2011.25.

[19] C. Paniagua, "Service Interface Translation. An Interoperability Approach," *Applied Sciences*, vol. 11, no. 24, p. 11643, Dec. 2021, doi: 10.3390/app112411643.

[20] Apache Camel, "camel apache." Accessed: Jan. 27, 2024. [Online]. Available: https://camel.apache.org/

[21] J. A. Vayghan, S. M. Garfinkle, C. Walenta, D. C. Healy, and Z. Valentin, "The internal information transformation of IBM," *IBM Systems Journal*, vol. 46, no. 4, pp. 669–683, 2007, doi: 10.1147/sj.464.0669.

[22] M. Seth, "Mulesoft – Salesforce Integration Using Batch Processing," in *2018 5th International Conference on Computational Science/ Intelligence and Applied Informatics (CSII)*, IEEE, Jul. 2018, pp. 7–14. doi: 10.1109/CSII.2018.00009.

[23] Hong-Gee Kim, "Semantic Web," 2003, Accessed: Jan. 15, 2024. [Online]. Available: http://krnet.gagabox.com/board/data/dprogram/792/T2-1.pdf

[24] Y. Jung and Y. Yoon, "Ontology Model for Wellness Contents Recommendation Based on Risk Ratio EM," *Procedia Comput Sci*, vol. 52, pp. 1179–1185, 2015, doi: 10.1016/j.procs.2015.05.155.

[25] N. Villanueva-Rosales, L. Garnica-Chavira, V. M. Larios, L. Gomez, and E. Aceves, "Semantic-enhanced living labs for better interoperability of smart cities solutions," in *2016 IEEE International Smart Cities Conference (ISC2)*, IEEE, Sep. 2016, pp. 1–2. doi: 10.1109/ISC2.2016.7580775.

[26] N. Gibbins and N. Shadbolt, "'Resource Description Framework (RDF)," Encyclopedia of Library and Information Sciences. Accessed: Jan. 21, 2024. [Online]. Available: https://eprints.soton.ac.uk/268264/

[27] E. P. Andy Seaborne, "SPARQL Query Language for RDF." Accessed: Sep. 27, 2024. [Online]. Available: https://www.w3.org/TR/rdf-sparql-query

[28] Deborah L. McGuinness and Frank van Harmelen, "OWL Web Ontology Language Overview ," *W3C Recommendation*, Feb. 10, 2004. [Online]. Available: https://www.w3.org/TR/owl-features/

[29] J. Köpke, "Annotation paths for matching XML-Schemas," *Data Knowl Eng*, vol. 122, pp. 25–54, Jul. 2019, doi: 10.1016/j.datak.2017.12.002.

[30] J. Kopecký, T. Vitvar, C. Bournez, and J. Farrell, "SAWSDL: Semantic Annotations for WSDL and XML Schema," *IEEE Internet Comput*, vol. 11, no. 6, pp. 60–67, Nov. 2007, doi: 10.1109/MIC.2007.134.