IT-Studienprojekt (ERASMUS+, IMIT)

Service and device monitoring on devices in IIP-Ecosphere

Version 1.0 vom 24. Juni 2021
(Vor Abgabe entfernen)

Miguel Gómez Casado

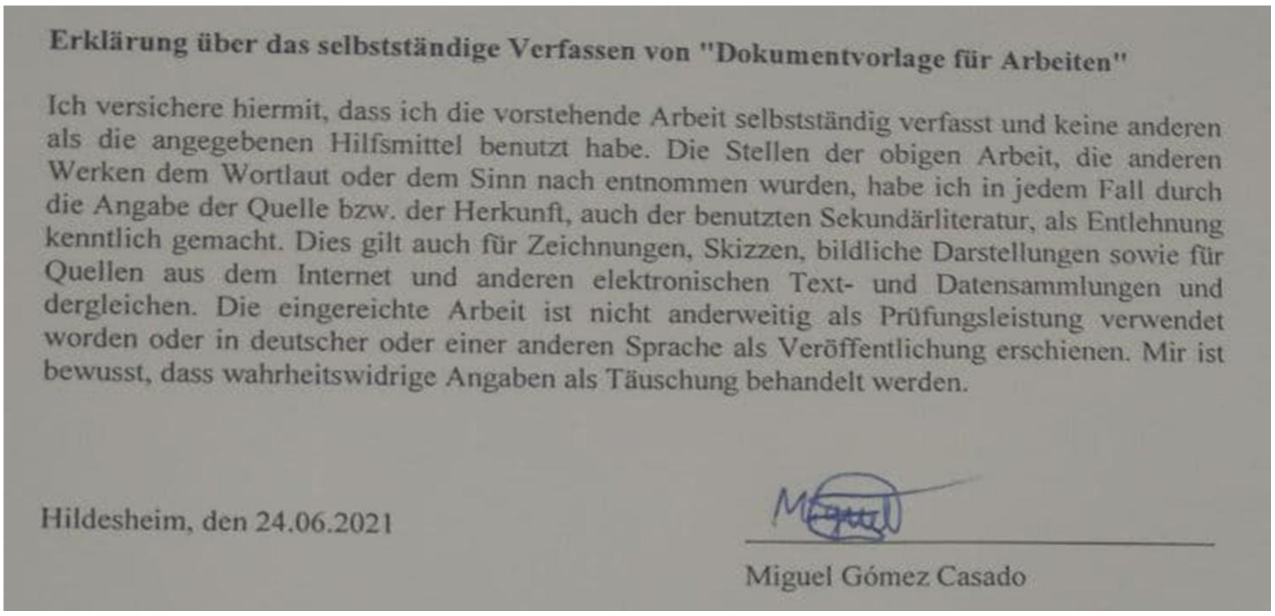
375687

gomezm@uni-hildesheim.de

Supervisors:
Prof. Dr. Klaus Schmid, Dr. Holger Eichelberger, SSE

Arbeitsgruppe Software Systems Engineering • Institut für Informatik
Universität Hildesheim • Universitätsplatz 1 • D-31134 Hildesheim

Eigenständigkeitserklärung



Erklärung über das selbstständige Verfassen von "Dokumentvorlage für Arbeiten"

Ich versichere hiermit, dass ich die vorstehende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der obigen Arbeit, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen wurden, habe ich in jedem Fall durch die Angabe der Quelle bzw. der Herkunft, auch der benutzten Sekundärliteratur, als Entlehnung kenntlich gemacht. Dies gilt auch für Zeichnungen, Skizzen, bildliche Darstellungen sowie für Quellen aus dem Internet und anderen elektronischen Text- und Datensammlungen und dergleichen. Die eingereichte Arbeit ist nicht anderweitig als Prüfungsleistung verwendet worden oder in deutscher oder einer anderen Sprache als Veröffentlichung erschienen. Mir ist bewusst, dass wahrheitswidrige Angaben als Täuschung behandelt werden.

Hildesheim, den 24.06.2021

[illegible]

Miguel Gómez Casado

Abstract

This project is part of a much broader project called IIP-Ecosphere [IIPE19]. The IIP-Ecosphere project aims to create a framework that will act as an industry standard for the upcoming Fourth Industrial Revolution. The focus of this particular project lies within IoT monitoring. More in particular, the monitoring of services and other resources in the environment of Industry 4.0. In a steadily drifting towards ubiquitous information systems, the demand for IoT and IoT standards grows, and the monitoring of these devices and services becomes an important aspect to be taken into consideration. Through monitoring we can decide what actions to take and detect uprising issues and problems before they take place, allowing us to react to them quickly and effectively. This project aims to add value to the aforementioned IIP-Ecosphere project by adding a service monitoring interface and component that fits into the project's requirements and structure, that would allow us to monitor the different resources and the devices they run on.

This document will first introduce the different motivations and goals for this project before delving into the background existing prior to starting the project's development. As the document progresses, we will see come more technical elements of the project, starting with the requirements that this project has and how they fit into the broader scope of the IIP-Ecosphere project, then we will have a look at the project's architecture, starting with a conceptual model and an initial idea to later flesh out a more complete and detailed architectural model. Later on, we will go into implementation details and decisions taken for the project's technical part as well as how they were tested and validated. We will end the document by explaining how this component can be integrated with the rest of the IIP-Ecosphere project and add some concluding remarks.

This project resulted in the creation of a software component that allows a resource we are trying to monitor to record and expose a series of measurements that can later be collected by a monitoring resource. The elements required for both exposing and collecting the metrics are both part of the component in question, reducing the effort and time required to use this type of monitoring. All of this has been achieved following the IIP-Ecosphere's requirements, where we combine the use of Spring Cloud Stream [VMM21] and Micrometer-API [PS21] as well as the use of Asset Administration Shells [PI418]. We will, of course, briefly discuss this technologies in this document. We tried to keep the monitoring as efficient timely as possible, achieving an average 5ms recovery time for recorded meters.

List of contents

List of figures

List of tables

List of abbreviations

API	Application Programming Interface
IoT	Internet of Things
AAS	Asset Administration Shell
CEO	Chief Executive Officer
AI	Artificial Intelligence
HTTP	Hypertext Transfer Protocol
I4.0	Industry 4.0
JSON	Java Script Object Notation
XML	Extensive Markup Language
UI	User Interface
POM	Project Object Model
CRUD	Create, Read, Update, Delete
OS	Operative System
UML	Unified Modeling Language
MQTT	Message Queuing Telemetry Transport
AMQP	Advanced Message Queuing Protocol
RAM	Random Access Memory
JAR	Java Archive
OT	Operational Technology
IT	Information Technology

1 Introduction

This project is not an isolated project aiming to obtain a software solution to an existing problem, it is part of a much broader project known as IIP-Ecosphere[IIPE19]. IIP-Ecosphere is a German initiative that aims to research and improve technology in the scope of Industry 4.0. Due to the current evolution of information technologies, slowly advancing towards the area of ubiquitous information systems, more commonly referred to as Internet of things or IoT for short, IIP-Ecosphere is currently undergoing research in this area to integrate new solutions to the ever-changing and relentless evolution of industry.

The current state of IIP-Ecosphere features a multilayer structure. The best way to describe this would be using the tree analogy. At the very bottom of the tree we would find edge devices, devices directly providing a service, taking measurements or, in short, supplying a product that adds value to the overall industrial system in some way. Using the tree analogy, this would be the "roots" extracting nutrients and required resources for the system. We would then work ourselves up the tree going up the different layers, letting go of specific details of the edge devices in favor of an abstraction from the physical devices, providing important aid to the structure, similarly to what a tree trunk would do. The final layer would involve a cloud or a cluster. This would be the most "visible" part of the system, what would be used to manage the system in general.

To further aid in the construction of this structure, every element and layer of the system is considered a resource. By denoting as resources all devices, layers and services, we draw a further abstraction from the different elements, allowing us to simply treat all of them equally under this umbrella term.

We will now go more into detail about this particular project and what role it plays in the bigger picture that is IIP-Ecosphere. This project aims to create an infrastructure for what will become the resource monitoring layer of the IIP-Ecosphere project. In the following subsections, we will see what the motivation and the targets of this project are.

1.1 Motivation

The resources that are part of a system are hardly ever perfect. It is already something quite usual in the world of information technologies for a device or a service to suddenly stop working unexpectedly, or to work in a different manner to what was planned for it. When working at an industrial level, one of this seemingly small or common issues could cost thousands or even millions of Euros, depending on how critical the system that has failed is. For this reason, being able to monitor a resource is an important matter and one that should be taken into account.

It is very hard to manage a resource if you are unaware of its current state, not to mention, that it is also something critical when it comes to detecting problems before they occur. Failures and problems in information technologies, like most other issues that occur elsewhere, have a cause. Perhaps there is a shortage of a system resource such as memory, or one of the components is damaged and is negatively impacting the throughput. If this resource is being monitored, all this variables can be detected before the issue escalates beyond control, allowing us to take action early on and avoid future complications.

This aspect was something that had indeed been mentioned and planned within the IIP-Ecosphere project but had, until now, been left aside to focus on more pressing matters. With better specified requirements an already existing infrastructure for the project, this part of the IIP-Ecosphere project became available to be picked up and worked on.

Apart from this already mentioned details, there is also a scientific motivation behind this project as many of the technologies in use here are new and have yet to be fully tested and explored. This prompted some research questions that were also used early on in project development to set a more clear goal or path to follow as, due to the aforementioned novelty of these technologies, the steps to follow were never fully clear. By setting trying to answer this questions as a minor goal or focus, we could later build the more concrete goals. These questions were:

- What architecture is feasible in this context, could a standard approach using Micrometer Monitoring API [PS21] help here?
- How can Micrometer-API be integrated with Industry 4.0 (I4.0) Asset Administration Shells (AAS) [PI418] on both Information Technology (IT) infrastructure and Operational Technology (OT) resources?
- Is AAS fast enough for IoT monitoring?
- How could custom probes be integrated?

1.2 Goals of the project

One of the approaches taken by IIP-Ecosphere to aid in drawing an abstraction to the specific and more detailed resources was using a new technology known as Asset Administration Shell (AAS)[PI418]. An AAS represents resources as assets. An asset would be a model of the resource in question while the administration shell would be an interface to interact with this asset in a uniform way. The monitoring functionality that this project aims to add to the IIP-Ecosphere has to fit into the idea of an AAS, which is one of the requirements of the project. The target of this project is, therefore, to provide an infrastructure to the environmental monitoring of resources for the IIP-Ecosphere project. The monitoring mechanism provided must be able to be represented as an asset, or rather as part of an asset, and, as a result, will be accessible using an AAS. We will go more into details of what an AAS is and what it implies later on in this document.

Even though the final product of this project has to be functional, it is not expected to add a fully fleshed monitoring service that would work in every scenario, but to provide a starting point or, using a aforementioned word, to provide an infrastructure to the monitoring layer. The end product will allow an AAS to read the levels of usage of certain system resources as well as providing an interface to allow the creation of customized probes that would allow us to monitor more specific elements of the resource such as, for instance, the throughput of the service it provides.

With this paragraphs in mind, we now present table 1, that shows the specific goals the project has more clearly, showing the main goals that drove the project and adding some smaller sub-goals that were added later on.

Table 1: Project Goals

Goal	Description
G1	Create a component that can read the current state of system resources from an OT resource
G1-1	Extract Micrometer-API metrics from the OT resource
G1-2	Extract memory and disk metrics from the OT resource
G2	Create a component that allows to read the system resources using an AAS
G2-1	Add an infrastructure or a skeleton to building an AAS supporting the monitoring
G3	Create a component that can add and/or modify custom probes to the OT resource we are monitoring
G3-1	Allow the custom probes to be created and/or modified by either the OT resource or the monitoring IT resource
G4	Create a component that provides a way to parse monitoring information into a functional ADT
G4-1	Provide a way to update a custom meter using the ADT that represents it
G5	Keep average meter retrieval and parse times lower than 10ms

1.3 Planning of the project

It has been hard to think of the plan for the project before getting started with it due to the fact that it is a very new concept both generally speaking, as some of the technologies in use are still not fully implemented at this time and still going under research; and at a personal level, as I never worked with anything related to IoT or monitoring. This was, as a result, a perfect opportunity to learn something new and provide value both to the IIP-Ecosphere project and to my own personal knowledge. The plan of this project was, as a result, built at the same time as the project, creating an initial idea and iterating over it as progress was being made. The way this project was driven was by taking note of the known technology, the progress that had been made and adding a small and reachable goal to improve the project. Many of this goals were, of course, motivated or based on the requirements and overall goals of the project, but also on the previously mentioned research questions.

The first part of the project involved getting familiar with the technology being used in the IIP-Ecosphere and understanding what was being developed within the project's scope. Although this would not provide a tangible product or value to the project, it was an important starting point to improve speed and efficiency when it came to technical work. A minimum understanding of how the tools that are going to be used is always a vital component to any type of work. Otherwise, the project would advance at a far slower pace than feasible, practically dooming it from the start.

After that, we continued working, attempting to create a simple monitoring interface using different prototypes and approaches to draw conclusions on which approach was the best one. These prototypes were simply to test how the interface would react to a normal situation. The result would be a small prototypical infrastructure that would allow the addition of more components and functionality as we slowly worked our way towards building a more advanced software component that allowed us to achieve the goals we had set for the project.

The following step involved iterating over the previously mentioned prototypical infrastructure. Starting by making that infrastructure independent from the testing prototypes and refining the existing functionality as well as adding new one. This part aimed to turn the infrastructure into a software library that could later be deployed in a working project to add the monitoring functionality with as few changes as possible. Once again, this strived for achieving a further independence of the monitoring component. This would be the part that added the biggest amount of value to the project, as the added functionality would be tested and validated to ensure its correctness.

More functionality was added after noticing some parts from the testing prototypes could be added into the component for a more standardized and uniform execution and, roughly after the tests started to provide good results, we started working on this document which, in turn, also provided some insight from a different perspective that would also enrich the technical work from the project as, in one way or another, by writing the progress down, these ideas became clearer and more organized, allowing to make additions.

1.4 Structure of the project

To finish off this section, we will speak a bit about the structure of the project. This project is composed of a software component that can be divided into two parts, and this document. One of the parts contains the functionality required to extract and expose system metrics from an OT resource, as well as adding the possibility of creating and modifying custom probes in the same way. The second part corresponds to the functionality required to read the data from the IT resource by using an AAS and create usable ADTs to make their management more comfortable. This document acts as a memoire of the project, documenting its progress and conclusions.

More detailed information of the software component's inner structure will be provided in the section regarding architecture.

2 Project background

In this section we discuss the approach taken for the project and what decisions were made both early on in the project and as it progressed. We will also include some subsections detailing the approach taken to study, research and better understand the technology and tools being used in IIP-Ecosphere and, as a result, are also used in this project.

Starting off with the initial planning of the project, there were two possible approaches that could be taken overall. The first approach would be a top-down approach, where we would start from an abstraction of the component we wanted to created and refine said abstraction into a more specific component until achieving the desired goal. As a result, this approach is often more theoretical at first, trying to find an adequate design prior to implementing a solution. This approach is often used because it is a good way to always keep the main goal or the final target in mind. It is also beneficial in order to have a general idea on how the system works as a whole without initially overwhelming the developers with specific functionality. The second approach would be a bottom-up approach, which essentially is the opposite of the previously mentioned top-down approach. In this case, we start by creating the specific components with some functionality, then add them up to achieve bigger and more complex functionality, slowly building up the system until we have the final product. This approach is more technical than the top-down approach, as it usually works in incremental addition of functionality to slowly build up the bigger picture we started off with in the other approach. This approach is a good idea when we want to take smaller steps and draft a plan in a more flexible way, determining what the next step could be based on the already existing progress.

After giving it some thought, it was decided that the approach the project would follow would be a bottom-up approach. There were, of course, multiple reasons to support this decision. The main reason that fueled this choice was the fact that, being new to this technology, it was difficult to picture the abstract overview required for the top-down approach. Also related to the lack of knowledge in this field, a bottom-up approach would be helpful to understand the way the different components and elements worked individually, and later on experience how they could work together. A more technical or experimental bottom-up approach seemed more likely to render good results than a theoretical top-down approach when taking these circumstances into account.

In the following subsections, we will discuss some of the technologies that have been used and what approach was taken in the initial stages of the project to further comprehend how they worked and what value could be extracted from them.

2.1 Asset Administration Shell (AAS)

Although we have already introduced what an AAS is in the introduction, we will go more into details about it in this subsection.

As previously mentioned, an AAS represents resources as Assets which, in turn, are interacted with by using an interface, the administration shell, that treats all the assets equally. To further illustrate this idea, we could simply use the hierarchical structure in a normal company. We can imagine there is a management board or a CEO at the head of the company, a series of department directors and then the workers working in each department. When the CEO wants something done, they would never go directly to each worker from a specific department to personally request them to do their specific part that would, eventually, help them reach their goal, mostly because he would probably be unsure of what each employee does specifically and would most certainly end up giving the wrong instructions to the employees. The CEO would talk to the department director and file their entire request directly to them. It would be the department director, who knows exactly what each worker is supposed to do, the one that would give out the specific orders to each individual worker. In this analogy, the workers would be assets and the department directors would be Asset Administration Shells. The CEO would be the process or processes that use the asset administration shell to manipulate the different assets.

So, why was AAS an idea to begin with? AAS is the implementation of the concept of a digital twin that is being developed fur Industry 4.0. It aims to establish cross company interoperability and to create a series of standards for the integration of IoT into this Fourth Industrial Revolution. This idea allows the combination of intelligent resources, which are resources with AIs, and passive resources in a uniform way. By representing different elements involved in the industrial production as assets, we can create a digitalization of our resources, which would, in turn, make the overall system more efficient.

An AAS is composed of a header and a body. The header part of an AAS contains the relevant information to identify, configure and use an asset. The body of the AAS contains what we call "submodels". A submodel contains a hierarchy of the different properties that the asset has. The properties represent a feature or some data that the asset might have. With this two components combined, we can identify an individual asset among multiple similar resources and know what type of information we can obtain from this asset as well as what we can do with it.

The project behind the creation of the concept of an AAS is BaSyx [EF21]. BaSyx is a project owned by Eclipse that aims to create an open source platform to aid in the automation of industry in the Industry 4.0. The idea behind AAS is to create a standard version of a digital twin, a digital representation of a real device or resource. The target is to help all interested stakeholders to advance and shape the Fourth Industrial Revolution. BaSyx is, therefore, an off-the-shelf component implemented in different languages to allow its use in the most extensive way possible. This project is also partially funded by the Federal Ministry of Education and Research of Germany.

This technology lies at the base of the AAS implementation of IIP-Ecosphere, although, as one can imagine, IIP-Ecosphere does draw an abstraction from this base technology, providing a slightly different, more flexible interface for this technology. Understanding how BaSyx worked underneath was helpful to further understand this abstraction by the IIP-Ecosphere of this technology. By having a look and experimenting with some of the prototypes provided by Eclipse, we were able to gather quite a good understanding of this technology prior to starting with the development of a prototype.

Further prototypes of AAS were also studied, this time from the IIP-Ecosphere. Although based on the same principals as BaSyx, having a look at this particularity was also vital, since, after all, this was the version of AAS we would be using for the project. Some of the functionality was left open, allowing for a more flexible use of this technology as well as providing tools to also simplify its use.

2.2 Spring Cloud Stream

Spring Cloud Stream is a framework designed to create highly scalable and event-driven microservices while also allowing the option of connecting said microservices with messaging systems. Spring supports multiple binder implementations to allow the use of different messaging systems and technologies to be used together with Spring Cloud Stream[VMM21]. Spring Cloud Stream is actually quite wide spread and can be found in multiple software solutions form important and well known companies for different reasons. One of such is Netflix, who started using Spring Cloud Stream in 2015 in substitution to a pre-existing internal solution due to the improved reliability, scalability, efficiency and security that Spring Cloud Stream offers.[WCY+18] It is also used for the online services used by Target for online shopping, as it provides scalable, nimble and efficient microservice communication.[BD18] These two examples prove that Spring Cloud Stream can be used for multiple and different tasks

Spring had become of the technologies that had been chosen by IIP-Ecosphere to the point of actually becoming a requirement [IH21]. Spring would be used in the transport layer of IIP-Ecosphere to connect components that could be completely independent from one another, via the usual I4.0 protocols such as MQTT [MO20] or AMQP [O14]. A simple prototype, for example, was being built at the time this project was being carried out that combined an artificial intelligence (AI) programmed in Python that allowed to stream the produced data using Java. This is, of course, the type of flexibility required by the IIP-Ecosphere project.

There were, of course, a series of different options considered by the IIP-Ecosphere when it came down to choosing an adequate streaming component that could be used as the transport component[IH21]. Several of them were discarded due to them not meeting requirements from the project, in particular those regarding flexibility. The requirement to also use AAS wherever possible was also an important limiting factor when it came to deciding what platform to integrate to the IIP-Ecosphere project. A few still remained, but some experiments run on the components started to reveal issues that discarded other options, such as severe lack of documentation to execution errors. The decision of using Spring Cloud Stream was then settled, after determining that it was both the most stable of the remaining candidates as well as fitting many of the requirements. The ability to exchange transport protocols for individual streams allows the user to define the formats they require for each transport necessity.

2.3 Micrometer-API

One of the technologies that had been considered already by IIP-Ecosphere in regards to monitoring had been the Micrometer-API[PS21]. Micrometer is a vendor-neutral application metrics facade. It provides an interface over the instrumentation clients. Micrometer allows the creation of probes and meters that operate under the same facade and, registering them under a simple name, we can use to retrieve them later. There are already a series of bindings that are already configured in Micrometer-API, simplifying the work required to use it. One final detail that makes is very attractive for the IIP-Ecosphere project is that it is integrated into Spring, the previously mentioned technology. With Spring already as a requirement for the project, adding Micrometer seems to be a simple idea as the part of the work involving its start up is already handled by Spring under the hood, relieving us from some of the workload as well as simplifying much of the work that would be otherwise required for this.

In spite of the Micrometer-API evidently being by multiple monitoring services, including Spring Boot, Prometheus or JMX[MRC06], Micrometer-API is not that way known, or at least is not well publicized, as other than these mentions found on the official page implementation and the example we used to learn how Micrometer-API worked in unison with Spring Cloud Stream[MH20], we have been unable to find any other examples of its usage easily. Perhaps this is due to its integration in other tools, which it uses as support to obtain the meters, meaning that you don't have a standalone Micrometer-API component that can be used.

3 Requirements

In this section we will cover some of the requirements that this project has and their relation with some of the requirement in the IIP-Ecosphere project. Apart from mentioning the requirements, each section will also include a small paragraph explaining how the different components are expected to be covered by the project.

3.1 Overall Requirements

The table below shows a series of requirements from the IIP-Ecosphere platform that were extracted from the platform handbook[IH21]. Since this project is part of the IIP-Ecosphere project, these requirements have also had to be taken into account:

Table 2: General requirements from the IIP-Ecosphere platform

Requirement	Summary
R1	Vendor and technology neutral
R2	Use of standards
R3	Design as virtual platform
R4	Design based on components and services
R5	Use of open source
R6	Open for optional/commercial components
R7	Use of AAS for interfaces
R8	Use of systematic variant management techniques
R9	Means for availability
R10	Soft real-time processing (<100 ms) for production critical functions
R11	Documentation
R12	Documentation of processing steps

These requirements cover the entire IIP-Ecosphere platform and are general requirements that have to be fulfilled by each component of the IIP-Ecosphere platform, including this project. Some of the requirements are, to a certain point, somewhat out of the scope of this project and are, in one way or another, fulfilled indirectly, simply due to its integration into the IIP-Ecosphere platform. Of course, on the other hand, most of these requirements have been refined to be more specific to this particular project, specifying some of the technologies to be used that fit inside the previously mentioned requirements:

Table 3: Specific requirements for the project based on the general requirements of IIP-Ecosphere

IIP-Ecosphere Requirement	Requirement	Summary
R1	R1-1	Use of Spring Cloud Stream boot component to manage and inject dependencies required for monitoring
R1	R1-2	Use Spring Cloud Stream Micrometer-API component to manage and create monitoring probes and values
R1	R1-3	Use of Spring Cloud Stream REST controller to design a RESTful service

R4	R4-1	Design a monitoring component that can be added to the IIP-Ecosphere platform
R4	R4-2	Access to monitoring values using a RESTful service
R4	R4-3	Modification and deletion of custom metric values using a RESTful service
R5	R5-1	Use of Spring Cloud Stream as framework (open source)
R5	R5-2	Use of Micrometer-API as monitoring platform (open source)
R7	R7-1	Use of AAS interfaces to access the metric values
R7	R7-2	Use of AAS interfaces to modify or delete custom metric values
R10	R10-1	Time required for retrieving and parsing meters cannot be above 100ms
R10	R10-2	Average time required for retrieving and parsing meters should be below 10ms in normal workload circumstances
R11	R11-1	Documentation of the available operations in the code
R11	R11-2	Documentation of the available operations available as JavaDoc handbook
R11	R11-3	Documentation of requirements to allow the component to function properly
R12	R12-1	Documentation of processing steps taken by the designed component

Most of the software components used to create and test the functionality of this project have been provided by the Spring Cloud Stream framework. Spring is an open source framework, meaning that its components also fall under the category of open source. Additionally, since Spring Cloud Stream already has a series of mechanisms that allow to make the transport and communication flexible as we saw before, we can also consider it technology neutral, at least in the sense of communication protocols, meaning that we fit the requirements handsomely by using this framework. Micrometer-API also falls under this category. We already saw that, in the official description of the API provided in their homepage [PS21], Micrometer-API is defined as vendor-neutral, once again successfully complying with the requirements. For this reason, the use of these two technologies seems to be very appropriate for the project.

With regards to the design based on components and services, since this project is going to be a part of the larger scale IIP-Ecosphere platform, it already makes sense that it is designed as a semi-independent component for the platform. Of course, we need to include some dependencies with the technologies we require to make it work, such as the already mentioned Spring Cloud Stream framework and the Micrometer-API, but this dependencies will already be dependencies of the IIP-Ecosphere project, meaning that this component is, in fact, independent inside the IIP-Ecosphere environment. The idea behind designing this project's product as a component is simple as well as sensible. The resulting component could simple be "plugged" into the resource we want to monitor and the resource carrying out the monitoring and that would be it. Components are also a fantastic idea when it comes to creating scalable systems, which is something that can come in very handy in the near future.

The existing integration of Micrometer-API in the Spring Cloud Stream framework already makes the metric values available to be viewed using an HTTP [BFF96] request. This is already something that, already fueled many of the ideas behind different decisions taken when implementing this project. HTTP is independent from the language and can be executed from different processes in completely different execution environments, even in remote and distant devices. This allows for a great flexibility when it comes to access the data. The resulting interface (naturally, an AAS interface) can simply access the metrics using an HTTP request. This would hide all the underlying process of requesting the metric, calculating it and retrieving it in an HTTP response. After this handy way of accessing the data was made clear, it seemed to be a good option to add a RESTful service to allow the modification of custom metrics to be done in the same way as the reading. REST services use HTTP as an application protocol as well as a transport protocol, creating a very simple, yet efficient and effective service. By using a REST service, every operation is equal to the eyes of the user, as well as creating a uniform and consistent type of operation.

With a fully operational RESTful service that allows the access and modification of metric values, we can easily create an AAS interface that can map the different metrics as properties so their access is completely uniform. Under the hood, the AAS would actually be using the RESTful service to obtain the different values and, similarly, to modify or delete custom metrics that we might want to include.

Regarding the quality requirement that involves the time limit of 100 ms mentioned in R10, we might need a further explanation as to why this is necessary. Monitoring a resource is important, to the point that it may be critical to the system, so keeping the limit is important. Due to some complications that appeared during the development regarding a specific type of metric that was used to gather system resource values that involved a significant overhead time to calculate them, it was decided to sacrifice the real time calculation of these values in favor of a more effective and less time consuming alternative which involved scheduling the calculation of these metrics rather than calculating them when requested. This schedule must also fit into the time limit and, as a result, has to be included as a requirement.

Of course, one final paragraph to discuss documentation, all progress and knowledge gained from this project would be lost if there is insufficient documentation. Documentation explaining what each operation does and how to correctly use it is important for both the use and the maintenance of the component. It is also important to document how to use and deploy the component where it is needed, explain what steps must be taken prior to adding it to an application or a process to ensure it works as expected. It is also important to document what the component is supposed to do when running, as this will show some insight to the work and idea behind it to quickly locate and solve an error if it occurs.

3.2 Transport and Connection Requirements

This section is significantly smaller to the prior section, but it was considered to be relevant to include in the document. Due to the dislocated or split nature of the component we create with this project, since it includes elements needed by the resource we want to monitor, as well as parts we require to monitor the service, it is important to include some of the transport and connection requirements that have been taken into consideration to design the component:

Table 4: Transport and connection requirements of IIP-Ecosphere taken into consideration

Requirement	Summary
R14	Open and flexible connectors
R19c	Restful APIs with JSON/XML

In order to allow the connectors to be flexible, no specific host or port is specified in the component when it comes to deploy the RESTful service. This will all be managed by the Spring Cloud application from the specific device or environment, meaning that the RESTful service and the monitoring components will work equally independently of the identity of the specific process and device running them. Of course, the port can be modified using a blueprint of sorts that is provided by the Spring Cloud application, but this detail will be shown in a later section where we dive deeper into implementation details.

The last requirement considered is regarding the format we are going to give the data for its transportation. We already know that the property shown in the AAS is an "illusion", as the value has to be requested to the RESTful service. The issue is how to send data from the AAS to the service without losing any of the values. There are multiple ways and alternatives to do this, but if we have to make such a decision, the best approach is to look at the requirements from both the overall project and other components in order to be consistent with the already taken decisions. For this reason, the decided data sharing format is JSON, following the requirement R19c.

4 Architecture

In this section we will take a look at the project's architecture. We will start by showing a diagram of the architecture of the IIP-Ecosphere project so we can locate where this project is nested before going into further detail about this project's specific architecture. In order to understand the structure we start from a general overview of the complete project before moving down towards the specific elements that compose this component.

4.1 IIP-Ecosphere architecture

Since we are working in the IIP-Ecosphere project, it makes sense to first have a look at the project's architecture before continuing to the specific component we are creating to locate it within the bigger picture that is represented by this broader project. The diagrams and information shown in this subsection has been extracted from the IIP-Ecosphere platform's handbook[IH21].

Figure 1 shows the multi-layered architecture of the project. The IIP-Ecosphere project is composed of 8 horizontal layers and a 9th vertical layer. As it is common with this type of architecture, one layer can only access resources from layers that are below it and never above it, managing the inter-layer dependencies by using a hierarchical design top-to-bottom. As it is also common with this types of architectures, there is also a relaxed, vertical layer (shown on the left) that may be accessed by every other layer in the project. This vertical layer is often used to have all configuration properties in one located place.

More detailed information can, of course, be located in the appropriate documentation of the IIP-Ecosphere platform, but in order to show a minimum amount of insight, we now list the different layers with a short summary of what each layer is responsible for:

- **Support Layer:** layer responsible for realizing basic abstractions to ease functions from upper layers. This layer allows the platform to reduce unnecessary redundancy as well as promote internal conversions.
- **Transport and Connectors Layer:** layer responsible for connecting different devices among each other and with other platform services. This ensures that the appropriate protocols and formats from I4.0 are used.
- **Services layer:** layer that provides openness and extensibility throughout different types of services, focusing on AI services. These services must be configured and orchestrated in order to be useful for the application. This layer defines the basic service interfaces, provides access to data routing and manages the automatic translation of data formats along data streams.
- **Resources and Monitoring layer:** layer that manages the monitoring of services deployed in resources. This is done in terms of AAS. This layer has a monitoring component which can be deployed along with services and use the capabilities of the Transport and Connection layer.
- **Security and data protection layer:** layer that provides data to the overall security configuration (authentication, cryptography, etc.), in particular, regarding AAS and BaSyx [EF21].
- **Reusable Intelligent Services layer:** this layer is responsible for paving a way for open, reusable and extensible intelligent services. Te actual functionality of this component in the context of a running application is also defined in the platform configuration.
- **Configuration layer:** layer that contains components to manage and operate on the platform configuration.
- **Applications layer:** layer that includes a simple platform user interface (UI) relying on configuration and AI-enabled applications.

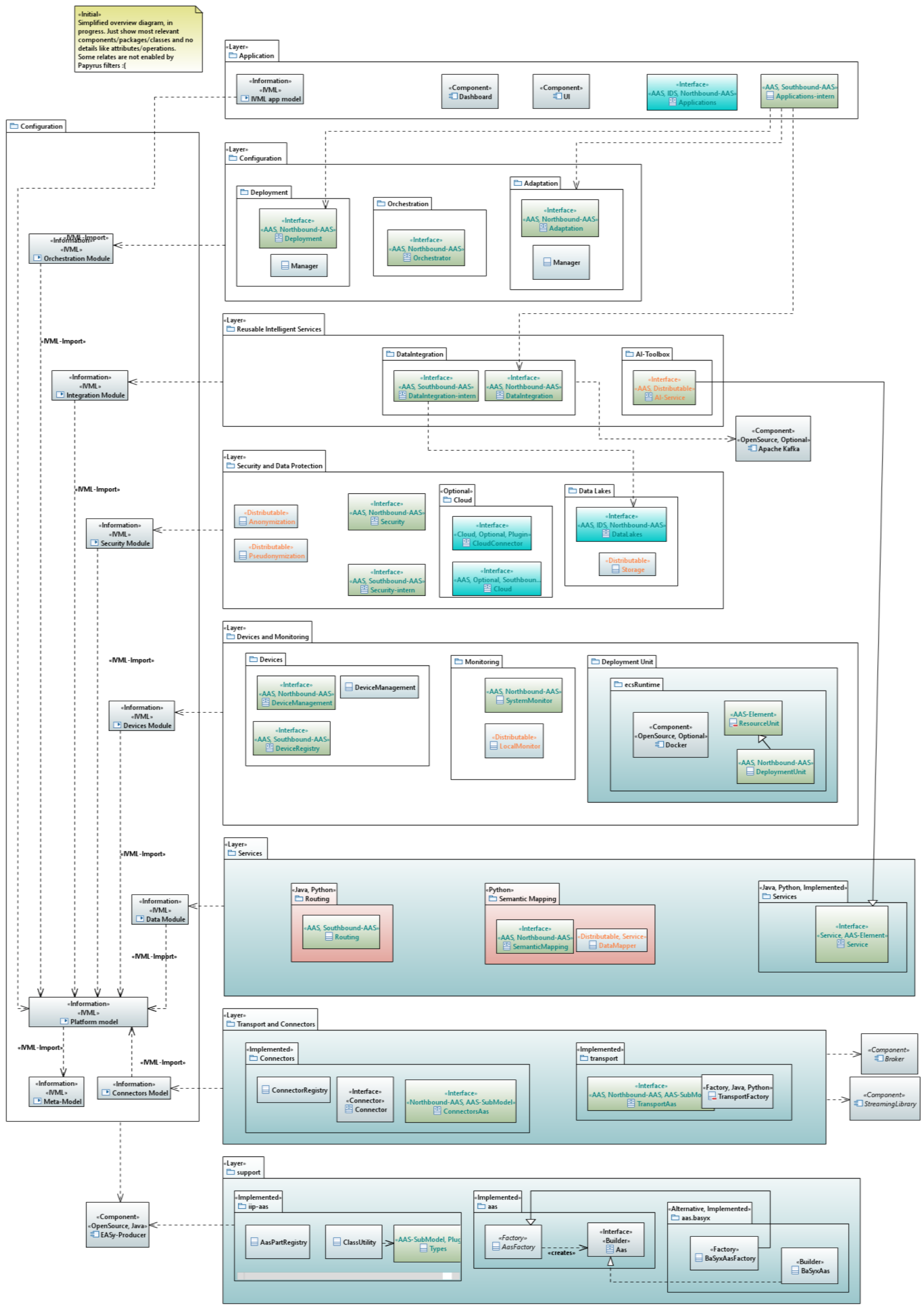

«Initial» Simplified overview diagram, in progress. Just show most relevant components/packages/classes and no details like attributes/operations. Some relates are not enabled by Papyrus filters :(
«Layer» Application
«Information» «IVML» IVML app model
«Component» Dashboard
«Component» UI
«Interface» «AAS, IDS, Northbound-AAS» Applications
«AAS, Southbound-AAS» Applications-intern
Configuration
«Layer» Configuration
Deployment
Orchestration
Adaptation
«Interface» «AAS, Northbound-AAS» Deployment
Manager
«Interface» «AAS, Northbound-AAS» Orchestrator
«Interface» «AAS, Northbound-AAS» Adaptation
«Information» «IVML» Orchestration Module
«IVML-Import»
«Layer» Reusable Intelligent Services
DataIntegration
AI-Toolbox
«Interface» «AAS, Southbound-AAS» DataIntegration-intern
«Interface» «AAS, Northbound-AAS» DataIntegration
«Interface» «AAS, Distributable» AI-Service
«Component» «OpenSource, Optional» Apache Kafka
«Information» «IVML» Integration Module
«Layer» Security and Data Protection
«Distributable» Anonymization
«Distributable» Pseudonymization
«Interface» «AAS, Northbound-AAS» Security
«Interface» «AAS, Southbound-AAS» Security-intern
«Optional» Cloud
«Interface» «Cloud, Optional, Plugin» CloudConnector
«Interface» «AAS, Optional, Southbound» Cloud
Data Lakes
«Interface» «AAS, IDS, Northbound-AAS» DataLakes
«Distributable» Storage
«Information» «IVML» Security Module
«Layer» Devices and Monitoring
Devices
«Interface» «AAS, Northbound-AAS» DeviceManagement
DeviceManagement
«Interface» «AAS, Southbound-AAS» DeviceRegistry
Monitoring
«AAS, Northbound-AAS» SystemMonitor
«Distributable» LocalMonitor
Deployment Unit
ecsRuntime
«Component» «OpenSource, Optional» Docker
«AAS-Element» ResourceUnit
«AAS, Northbound-AAS» DeploymentUnit
«Information» «IVML» Devices Module
«Layer» Services
«Java, Python» Routing
«AAS, Southbound-AAS» Routing
«Python» Semantic Mapping
«Interface» «AAS, Northbound-AAS» SemanticMapping
«Distributable, Service» DataMapper
«Java, Python, Implemented» Services
«Interface» «Service, AAS-Element» Service
«Information» «IVML» Data Module
«Information» «IVML» Platform model
«Information» «IVML» Meta-Model
«Information» «IVML» Connectors Model
«Layer» Transport and Connectors
«Implemented» Connectors
ConnectorRegistry
«Interface» «Connector» Connector
«Interface» «Northbound-AAS, AAS-SubModel» ConnectorsAas
«Implemented» transport
«Interface» «AAS, Northbound-AAS, AAS-SubModel» TransportAas
«Factory, Java, Python» TransportFactory
«Component» Broker
«Component» StreamingLibrary
«Layer» support
«Implemented» iip-aas
AasPartRegistry
ClassUtility
«AAS-SubModel, Plug» Types
«Implemented» aas
«Factory» AasFactory
«creates»
«Interface» «Builder» Aas
«Alternative, Implemented» aas.basyx
«Factory» BaSyxAasFactory
«Builder» BaSyxAas
«Component» «OpenSource, Java» EASy-Producer

Figure 1: Overall architecture of the IIP-Ecosphere platform[IH21]

With the architecture of the IIP-Ecosphere project in mind, we can now establish the location of the component created by this project within the broader project's scope. It is easy to establish that the component in question belongs to the resources and monitoring layer. Inside this layer, there are a series of subdivisions to further organize its contents. The particular subdivision where this component fits into is the environment subdivision. We will now look more in particular at the specific architecture of the project.

4.2 Project Architecture

Before diving deeper into a more detailed view of the project, we will start with an overview of the overall structure. Once again, in order to aid in better understanding the architecture, we will make use of some diagrams to visually illustrate some of the concepts and decisions taken to build the project's architecture.

The component produced by this project has been baptized under the name *Metrics Provider*. The reason behind this is that the first element created in the project, as well as the central and main element within it, is the one responsible for extracting the metrics we want to monitor and make them visible to the process or component who is responsible for carrying out the monitoring of the resource and its service. All other elements in this project revolve, in one way or another, around this particular element, either to aid it in its execution or to translate the data retrieved from it into a more manageable piece of information. Because of this significance in the project, the decision was made to name the project and component itself after it, as it is the best representative figure for the component.

As mentioned before, this project aims to provide the infrastructure to monitor a service and the resource running said service, as well as providing the required interface to comfortably access the information obtained through the monitoring. In this previous sentence, we can clearly see two significantly different parts inside the component.

The first subdivision is the one aiming to create an infrastructure for the extraction and exposure of metrics we want to monitor, including all the required elements to perform said task. The second subdivision contains elements to provide an accurate interpretation of the data produced by the first subdivision, parse it into a structure that is easier to manage and supply an interface to easily access the first subdivision. After taking this concept into account, we can clearly see that there is a sort of client-server relationship between these two divisions. The first subdivision acts like a server, supplying data upon request and lying in an otherwise "dormant" state when not in use, waiting for the other element to establish a connection and request its data. The second division acts like a client, connecting to the server to retrieve its data, and then interpreting the "raw data" it receives as a response, allowing us to parse said data into a more manageable and simple structure to use it efficiently. In figure 2, we can see the concept that we just described in a more visual manner.

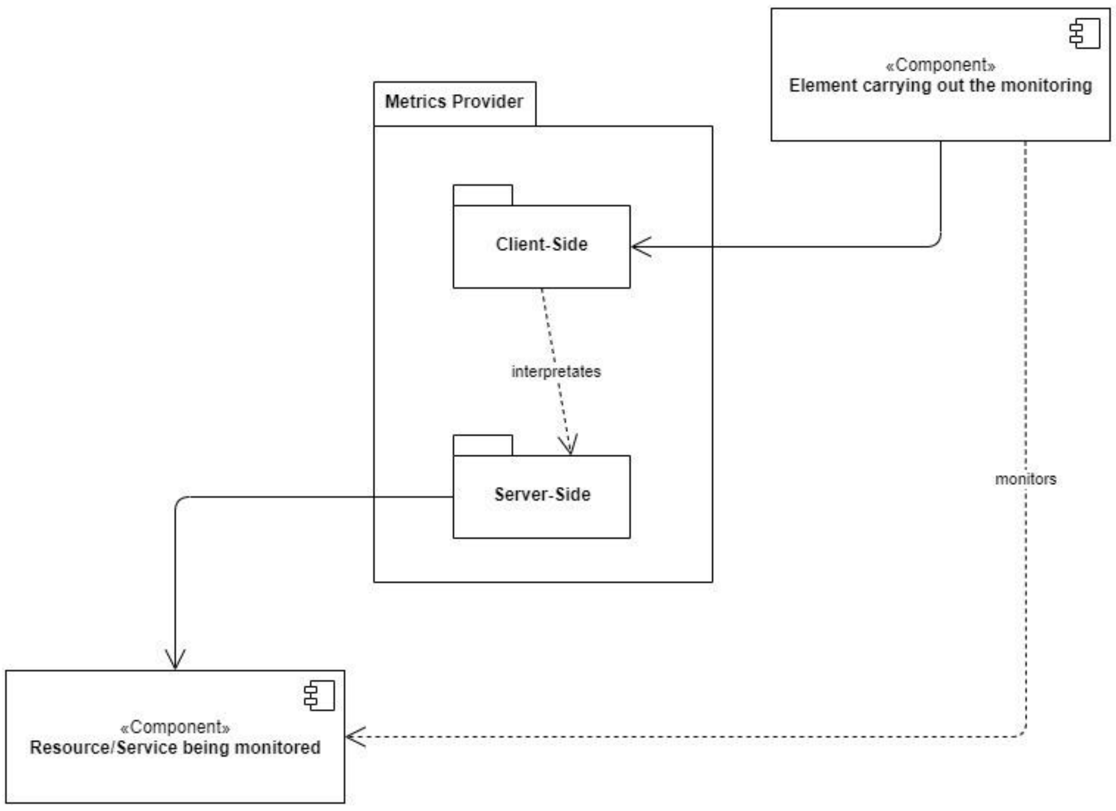


Figure 2: Conceptual model of the two component divisions in operation

With the concept introduced by figure 2 in mind, considering both subcomponents of the *Metrics Provider* component, we obtain the architecture of the project shown in figure 3. Figure 3 shows all the elements and the way they are organized within the project in a simplified way in order not to overload the diagram. We can clearly see that the component is first split into the aforementioned divisions before further subdividing and organizing the elements within. We will go into more detail in the following subsections

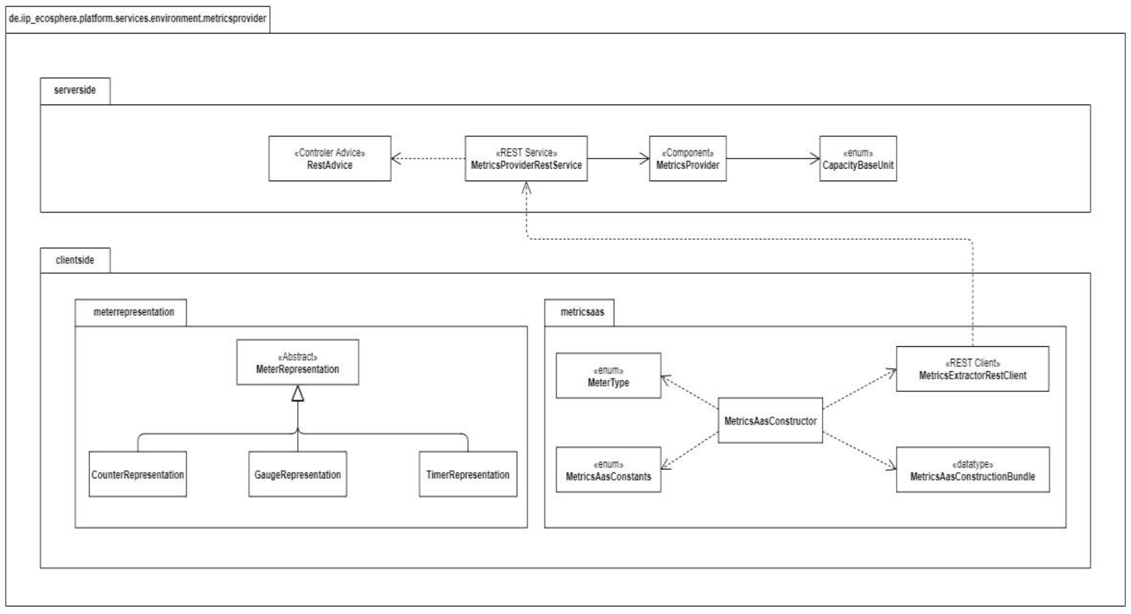


Figure 3: Simplified model of the *Metrics Provider* component

With the concepts introduced by figure 3, we can refine the conceptual idea introduced in figure 2 and provide a more detailed insight on how the component would theoretically work when in use. The individual tasks performed by each of the displayed elements will be detailed in the upcoming subsections to further illustrate the concept, but the diagram shown in figure 4 may be of assistance to have a visual idea and first impression of the roles each individual element plays when deployed.

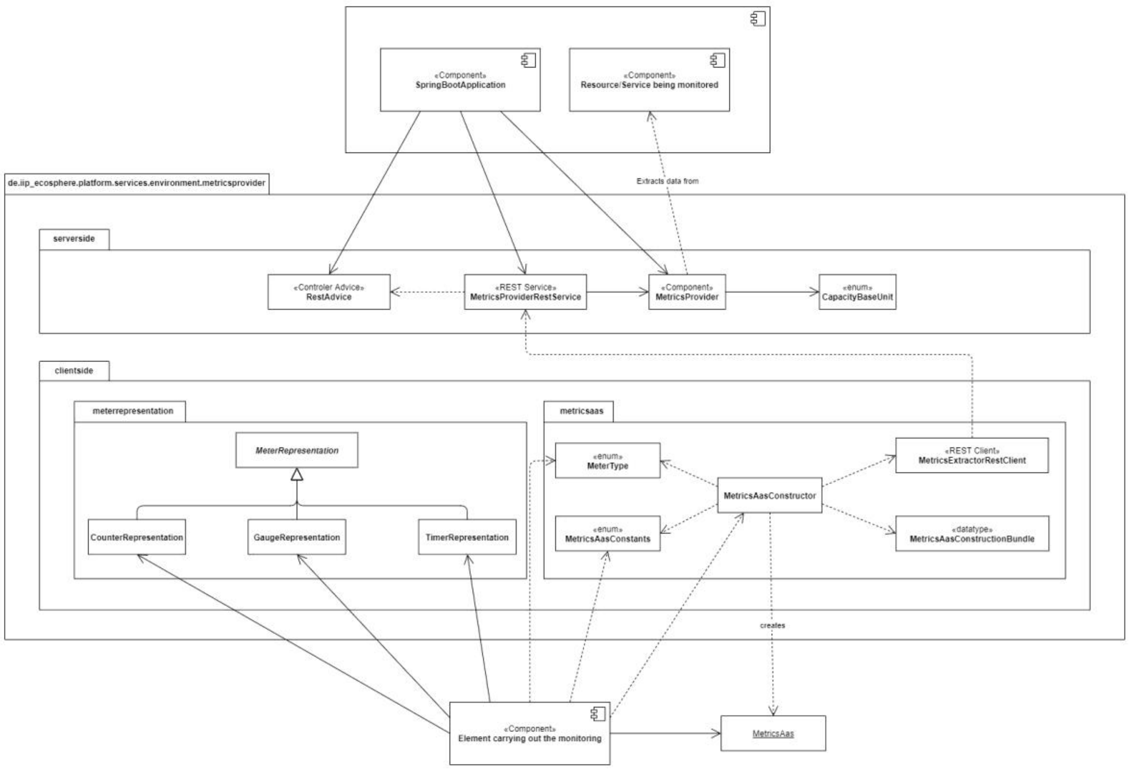

«Component» SpringBootApplication
«Component» Resource/Service being monitored
de.iip_ecosphere.platform.services.environment.metricsprovider
Extracts data from
serverside
«Controler Advice» RestAdvice
«REST Service» MetricsProviderRestService
«Component» MetricsProvider
«enum» CapacityBaseUnit
clientside
meterrepresentation
MeterRepresentation
CounterRepresentation
GaugeRepresentation
TimerRepresentation
metricsaas
«enum» MeterType
MetricsAasConstructor
«REST Client» MetricsExtractorRestClient
«enum» MetricsAasConstants
«datatype» MetricsAasConstructionBundle
creates
«Component» Element carrying out the monitoring
MetricsAas

Figure 4: Conceptual model of the individual elements from the component in operation

4.2.1 Server-side division

The first division we will look at is the one regarding the server side. As we briefly introduced earlier, this part of the component contains the required elements to extract and expose the metrics corresponding to a service and resource. As this division serves no other propose, there are no further subdivisions within this package. This package is composed by three classes and an enumeration: the *Metrics Provider* class, the *Metrics Provider REST Service* class, the *REST Advice* and the *Capacity Base Unit* enumeration. Some of the details from this classes will be explained later on when we reach the section in this document that covers the implementation, but a basic summary of the task performed by each of the elements is as follows:

- ***Metrics Provider.*** As previously stated, this is not only the central element of this package, but also of the entire component produced by this project. This element aims to extract the metric data from the resource and service we are going to monitor and expose it in a way we can read it from the client-side, following the relevant requirements from the project. As well as allowing the client-side to read the metric values, this class also provides functionality to add, modify and delete custom metrics that we might want to add to the resource both in the client side and the server side. This element is a specification of a predefined element from Spring, simply called *Component*. This can be seen in figure 5.
- ***Metrics Provider REST Service.*** This element is the one responsible for creating a RESTful service. This RESTful service will act as an interface between the client-side and the actual *Metrics Provider* that would be, otherwise, inaccessible without this element. This class serves the function to receive and process the client-side's requests to then provide the requested data that is extracted by the *Metrics provider*. It simply acts as an "gateway" between the two sides of the component. Just like the *Metrics*

Provider, this is also a specific element predefined by Spring called *Rest Controller*. In figure 5, we can see how this fits in.

- ***REST Advice.*** This element is a small class that exists to act as a simple error handler for the *REST Metrics Provider Interface*. There are some errors that can be expected to occur if there are certain conditions not met by the client-side when making a request. Other than simply returning an error with no further explanation, this element will respond with a more detailed description of the error that has been produced, showing some insight to what caused it and how the client-side can solve it. This was an element that was pretty much required by the Spring Framework in use, as it represents another predefined element from Spring, the *Rest Advice*, that is supposed to work side by side with the other element to effectively handle the errors produced by its operation.
- ***Capacity Base Unit.*** This element is an enumeration. Enumerations aim to add a restriction to a certain parameter or property from a software component. In the case of the *Capacity Base Unit*, the restriction it aims to add is the base unit for a system resource such as physical memory or disk capacity, that are measured in specific unit types. This adds a restriction to the unit in the way that only values such as "byte" or "gigabyte" are used, in opposition to a non standard base unit that the component would not consider valid. This element, as a result, adds both some aid to the client-side to know what units are valid and which ones can be requested, and some additional aid for the *Metrics Provider* itself to easily detect the unit a specific metric is shown in, and how to perform a unit conversion if needed.

This would be a summary of what each element does without going too much into implementation detail. Figure 5 shows the inheritance style or profile that this elements follow, specifying the particular type of architectural component they represent. Figure 6 shows how the different elements are related to one another and what type of association they share, as well as the directionality of said association. To keep the diagram as simple as possible, as well as omitting implementation details and decisions, this diagram only shows the names and the stereotypes of the classes without going any further into detail. A far more detailed diagram will be provided in the corresponding section.

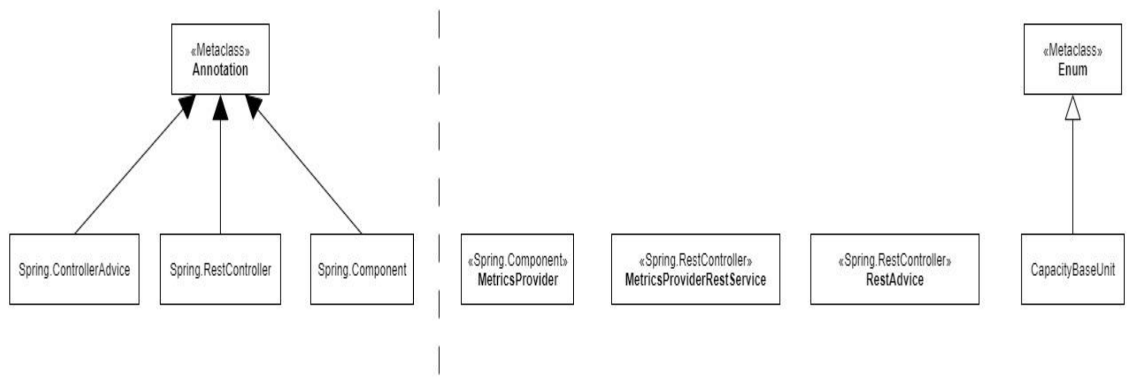

«Metaclass» Annotation
Spring.ControllerAdvice
Spring.RestController
Spring.Component
«Spring.Component» MetricsProvider
«Spring.RestController» MetricsProviderRestService
«Spring.RestController» RestAdvice
«Metaclass» Enum
CapacityBaseUnit

Figure 5: Inheritance style/profile of the server-side division

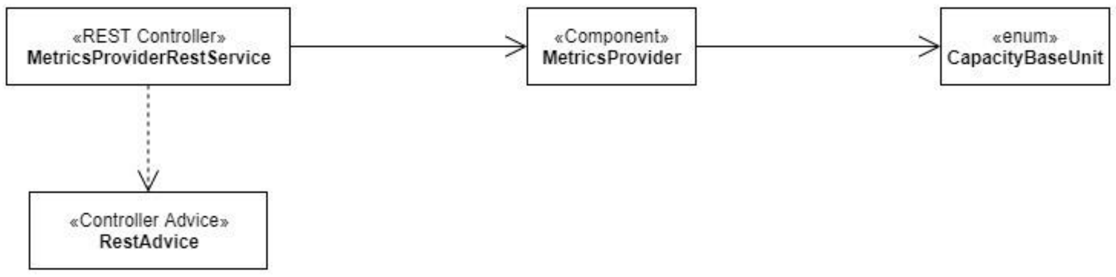

«REST Controller» MetricsProviderRestService
«Component» MetricsProvider
«enum» CapacityBaseUnit
«Controller Advice» RestAdvice

Figure 6: Simplified model of the server-side division

4.2.2 Client-side division

The client-side division is the second division the component has. This division contains the elements required to access the metrics that have been extracted by the server-side's components as well as providing a mechanism to parse the data retrieved from the server side into a more suitable and manageable structure that can be used by the application running at the client side and that is making use of this component. This would include both object representations of the meters as well as an AAS representation of the monitoring part of the asset. To have this part more organized, it is divided into two packages: one concerning the object representations of the meters, and another one containing the required tools to set up an AAS with the monitoring metrics.

The first package within this division, concerning the meter representations, is composed of four classes: the *Meter Representation* class, the *Counter Representation* class, the *Gauge Representation* class and the *Timer Representation* class. Just like before, we will detail the reasoning behind this classes in a later section, but a summary of this could be as follows:

- ***Meter Representation.*** Represents a generic meter. Mostly used to hold properties common to all the other meter types and to have common operations. It is an implementation of the *Meter* interface provided my Micrometer-API. All other meter representations are subclasses that extend this superclass.
- ***Counter Representation.*** Represents a counter meter. Provides required properties and operations to represent a counter as specified by the *Counter* interface of Micrometer-API. A counter is a type meter that allows us to count an occurrence. Counters only work in incremental values, they can never be decremented.
- ***Timer Representation.*** Represents a timer meter. Provides required properties and operations to represent a timer as specified by the *Timer* interface of Micrometer-API. A timer, as the name implies, allows us to record the time taken to perform a specific action or, if we have to manually add time from elsewhere, to manually add the time to the particular timer. A timer will know the amount of times it has been used (a counter, if you will), the maximum time that has been recorded and the total time that has been recorded.
- ***Gauge Representation.*** Represents a gauge meter. Provides required properties and operations to represent a gauge as specified by the *Gauge* interface of Micrometer-API. A gauge is simply a calculated value. Gauges are one of the most basic and useful types of meters as they can basically represent changing values. Any value that is not incremental, where we would instead use a counter, and that might vary in both positive and negative dimensions is a gauge.

The second package within the client-side division is one concerning the construction of an AAS with the metrics for the client-side. This part of the client-side, although vital to properly obtain the metrics, is independent from the meter representations, as they can both work without really using one another (in spite of this being a bit more difficult and elaborate). This package has another four classes as well as an enumeration: the *Metrics AAS Constructor* class, the *Metrics AAS Construction Bundle* class, the *Metrics AAS Constants* class, the *Metrics Extractor REST Client* class and, lastly, the *Meter Type* enumeration. This part is an important one regarding integration with the rest of the IIP-Ecosphere project and will be further detailed in the implementation section. For now, we will only present a brief overview of the different elements in this package:

- ***Metrics AAS Constructor.*** This would be one of the main classes in this package. As the name indicates, this element is responsible of creating the AAS. In reality, the client process using the *Metrics Provider* functionality will communicate solely with this component, so it acts as a sort of facade for the package. Needless to say, this component creates the AAS and provides mechanisms to extend it with custom metrics that we might want to add. In reality, this class creates the "skeleton" of the AAS, as all the functionality is provided by the *Metrics Extractor REST Client*.

- ***Metrics Extractor REST Client.*** This would be the second main element from this package. This element is the one that actually communicates with the server-side, acting as a link between the two. As we saw before, the *Metrics Provider REST service* exposes the metrics and the operations over them as a service. This component is the client that uses that service. Because of this, this class works in unison with the *Metrics AAS Constructor*: the AAS created by the *Metrics AAS Constructor* is the one that the client application will be using, but the functionality of the AAS itself is fully implemented by this other class.
- ***Meter Type enumeration.*** This enumeration was simply created to support the functionality of the AAS constructor in regards to adding custom metrics. We will see the specifics in detail when we reach the implementation details, but in short, by using this enumeration, the user can specify what type of custom metric they want to map into the AAS.
- ***Metrics AAS Constants.*** This class is also a support class for the AAS constructor, this one containing names and tags used for the system and Micrometer metrics. We will see the idea behind it, once again, in the appropriate section, as the reason for this class existing separately from the *Metrics AAS Constructor* class is an implementation decision.
- ***Metrics AAS Construction Bundle.*** This is yet another support class, but slightly closer to the structure of the AAS. As the name implies, this element represents a bundle that groups together several components that are required to build the AAS, allowing for a more comfortable and uniform access to said components. Its target is once again to support the construction of the AAS as well as simplifying the process as much as possible. The *Metrics AAS Constructor* will expect an instance of this class to be provided in order to successfully build the AAS.

This would be a summary of what each element does, ignoring further implementation details for now. Figure 7 shows the inheritance style or profile that this subdivision follows, similarly to the previous section, also showing the particular type of architectural component they represent. Figure 8 shows how the different elements are related to one another and what type of association they share, as well as the directionality. To keep the diagram simple, it only shows the names and stereotypes of the classes. A more detailed diagram will be provided in the corresponding section.

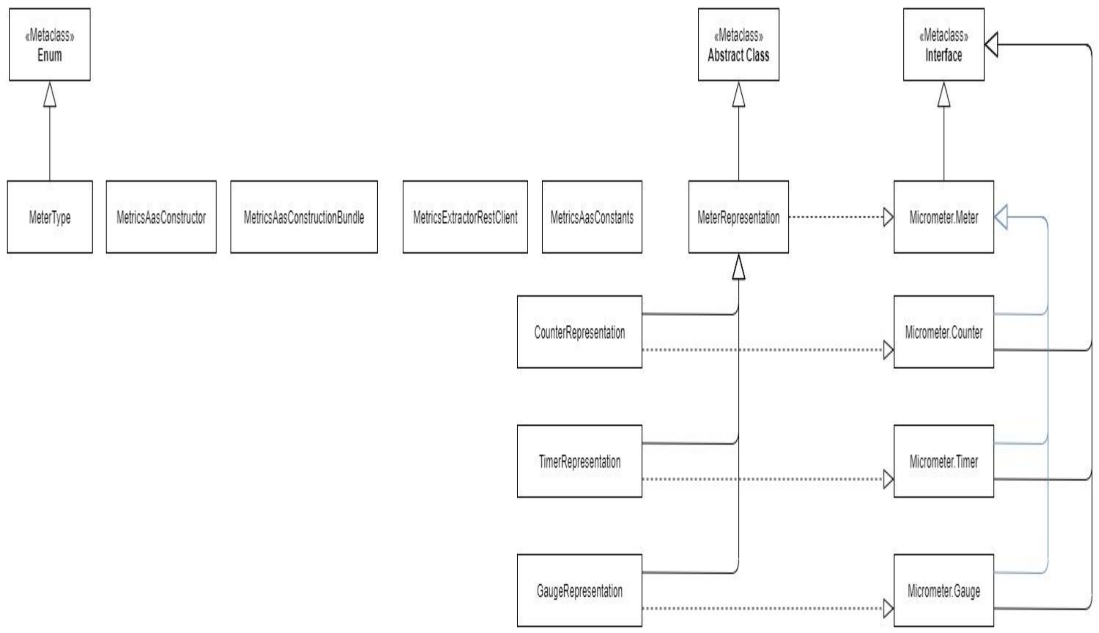


Figure 7: Inheritance style/profile of the client-side division

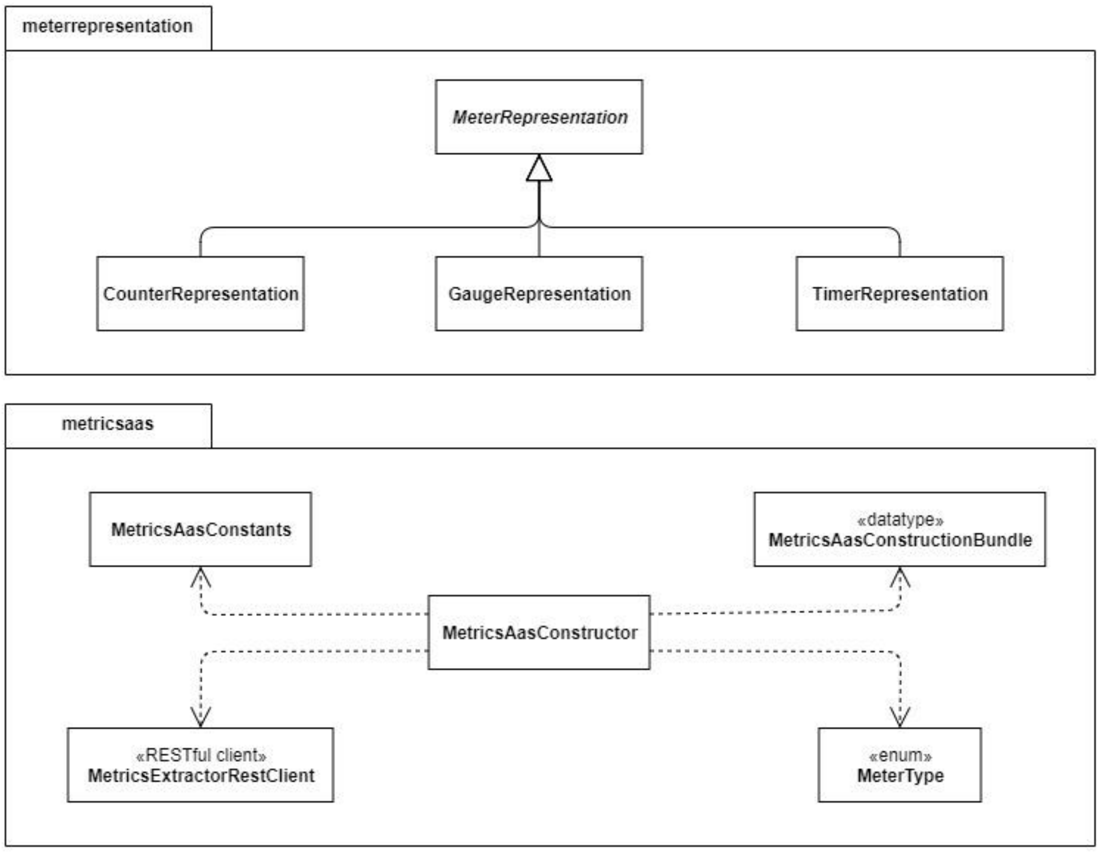


Figure 8: Simplified model of the client-side division

5 Implementation

In this section, we will go more into detail in regards to the project's implementation and the implementation decisions. This decisions are greatly influenced by the requirements, but not solely by them. Some of the decisions, as we shall see soon, were made due to limitations in the current technology as well as due to attempting to create an approach that could simplify or make an operation more secure in one way or another. We will start discussing some general decisions taken for the entire project before going into more specific decisions regarding each of the two divisions within the project.

5.1 General implementation of the project

Due to both our experience with Java and due to the fact that most of the IIP-Ecosphere's platform is currently written in Java, it seemed like a good idea to choose Java as programming language for the project. To further back this idea, the AAS implementation from IIP-Ecosphere and Eclipse BaSyx both are implemented in Java and, as a result, when we studied these technologies at the beginning of the project, we were already looking into them with this language. To add yet another reason to use Java, the idea behind the creation of a digital twin and, more in particular, an asset to be accessed through an AAS fits very well into the idea that inspired Object Oriented Programming [WP21]. Because of this, Java seemed a more than suitable option.

To further specify what type of Java project we are using, we decided to use a Maven project. Maven is an open source project management tool based on the concept of a project object model (POM). Maven manages the project's build, reporting and documentation from this central piece of information [ASF02]. This means that any dependencies our project has with other open source libraries are managed by maven, downloading and installing the required version of the library directly from the Central Maven Repository. This is very comfortable and safe when it comes to managing this dependencies, saving a lot of time when it comes to updating the library versions we require and not having to manually download the libraries, helping us save time. We will see the dependencies each of the different sides has when we detail them.

To add further reasons to using Maven, the IIP-Ecosphere platform is also available in a Maven repository, meaning that we can use the already existing components from the IIP-Ecosphere project, such as the AAS implementation and the adaptations made to Spring Cloud Stream to fit the requirements. Both of these are important to be able to build a prototype to test out the *Metrics Provider* Component to ensure it is working the way it is supposed to, as well as being able to add a functionality related to the AAS that can be used by an application built using the IIP-Ecosphere implementation of it.

In the following subsections we will see the specific implementation and implementation decisions of the two main blocks of the project. The order we will follow to explain them will be a chronological approach. The reason behind this decision is simply due to the fact that the component itself was incrementally built, starting with the core components that are needed to extract the metric data. Everything else was added later on to simplify future work and implementation of elements using the *Metrics Provider* component. With this approach to writing the following subsections, the reader will hopefully be able to understand the train of thought followed when creating the component as well as further understanding how the system works internally as each component was created as a result of wanting to extend the functionality of the previous.

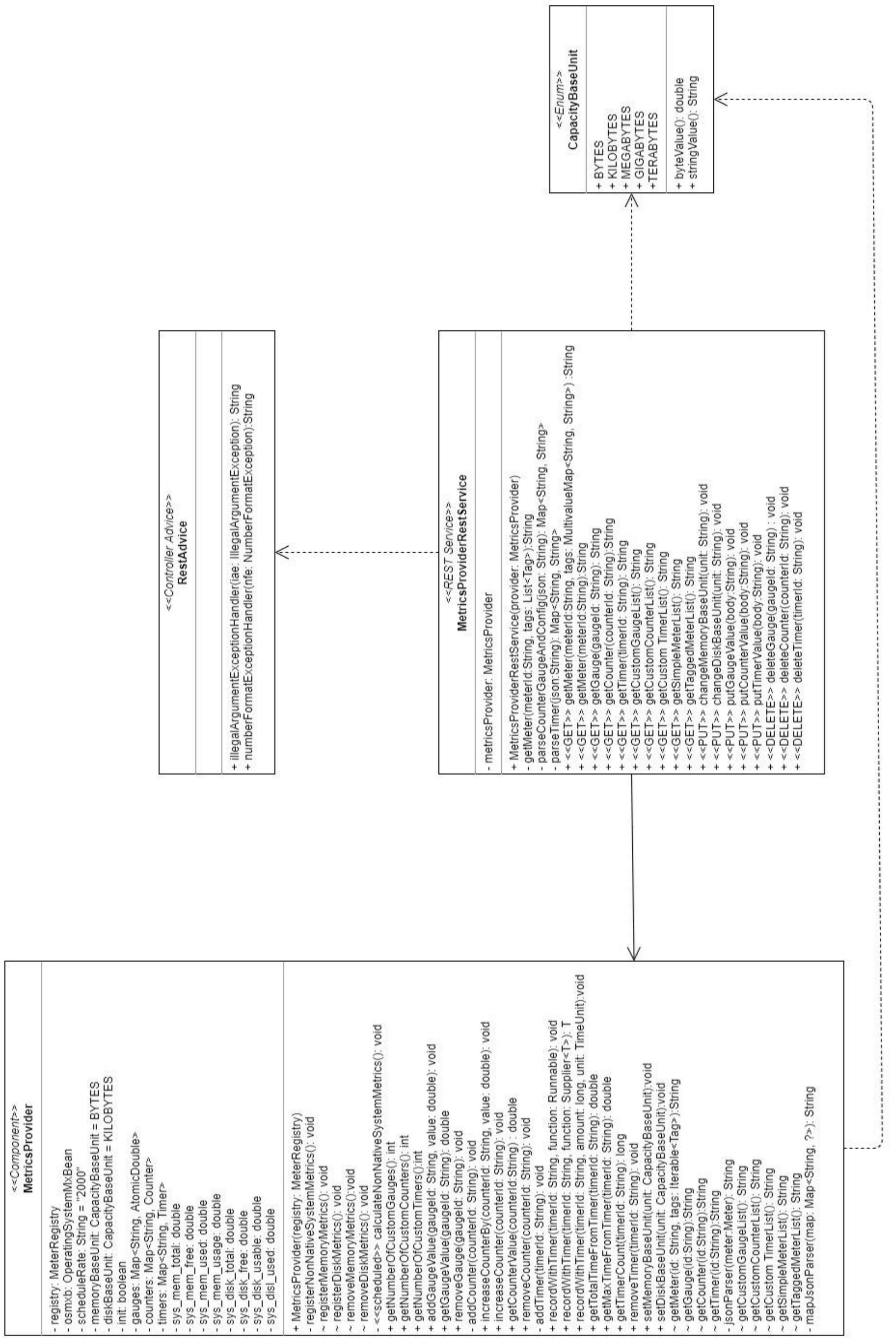
5.2 Implementation of the server-side

The server side was the very first part to be implemented for this component, more in particular the *Metrics Provider* class which, after all, gives a name to this component. As mentioned in the previous paragraph, this is the element that provides the core functionality to the entire component. If the requirements for deploying this component are satisfied, this component is enough to carry out the monitoring. It would, of course, require a lot of effort to do so, as we would have to create all the infrastructure to support an HTTP client and an AAS manually, as well as finding a suitable way of interpreting the data. We would also have the problem that this relationship would be one-sided, meaning we would only be able to read the metrics and not to modify or add custom metrics of our own. In spite of this issues, it can be done and, in fact, it was done at the beginning to inspire the changes that followed.

The *Metrics Provider* class was inspired by a small prototype created when we were first researching and understanding how the different tools in the project worked, more in particular Micrometer-API. During this phase of the project, we took to explore the internet trying to find examples of Micrometer-API applications. One of these examples was creating a Spring Application with a Micrometer component, which seemed to fit right into the projects restrictions to begin with. This example can be found on in the bibliography as it was an important asset to help starting up the project [MH20].

The example takes advantage of the integration of Micrometer-API into Spring Boot. The starting point is a Spring Boot Application that is executed to start putting the Spring Framework into action. There is a class labeled as "component" that will be created automatically by the Spring Boot Application and will have its dependencies injected. This class was a first version of the *Metrics Provider*. The injection creates an instance of a Micrometer-API class called *Meter Registry*. Upon creation, this *Meter Registry* will already register some of the system's metrics. By simply instantiating this class, we would already have the *Meter Registry* running. By adding a few more lines to the application's configuration file (called *application.yml*), we could ask the Spring Boot Application to expose this registry on an endpoint that could be accessed via HTTP. The example then proceeded to use an application called "Grafana"[GL21] to extract the metrics exposed on the endpoint to create graphs to be displayed on the screen. We would, of course, not delve deeper into this part, as Grafana is not relevant to our project, but if this application was able to use the metrics to create graphs, it was a clear indicator that we could do the same to represent the metrics using an AAS in a similar fashion.

Figure 9: Detailed class diagram of the server-side



<<Component>>
MetricsProvider
- registry: MeterRegistry
- osmxb: OperatingSystemMxBean
- scheduleRate: String = "2000"
- memoryBaseUnit: CapacityBaseUnit = BYTES
- diskBaseUnit: CapacityBaseUnit = KILOBYTES
- init: boolean
- gauges: Map<String, AtomicDouble>
- counters: Map<String, Counter>
- timers: Map<String, Timer>
- sys_mem_total: double
- sys_mem_free: double
- sys_mem_used: double
- sys_mem_usage: double
- sys_disk_total: double
- sys_disk_free: double
- sys_disk_usable: double
- sys_disl_used: double
+ MetricsProvider(registry: MeterRegistry)
- registerNonNativeSystemMetrics(): void
~ registerMemoryMetrics(): void
~ registerDiskMetrics(): void
~ removeMemoryMetrics():void
~ removeDiskMetrics(): void
- <<scheduled>> calculateNonNativeSystemMetrics(): void
+ getNumberOfCustomGauges(): int
+ getNumberOfCustomCounters(): int
+ getNumberOfCustomTimers():int
+ addGaugeValue(gaugeId: String, value: double): void
+ getGaugeValue(gaugeId: String): double
+ removeGauge(gaugeId: String): void
- addCounter(counterId: String): void
+ increaseCounterBy(counterId: String, value: double): void
+ increaseCounter(counterId: String): void
+ getCounterValue(counterId:String) : double
+ removeCounter(counterId: String): void
- addTimer(timerId: String): void
+ recordWithTimer(timerId: String, function: Runnable): void
+ recordWithTimer(timerId: String, function: Supplier<T>): T
+ recordWithTimer(timerId: String, amount: long, unit: TimeUnit):void
+ getTotalTimeFromTimer(timerId: String): double
+ getMaxTimeFromTimer(timerId: String): double
+ getTimerCount(timerId: String): long
+ removeTimer(timerId: String): void
+ setMemoryBaseUnit(unit: CapacityBaseUnit):void
+ setDiskBaseUnit(unit: CapacityBaseUnit):void
~ getMeter(id: String, tags: Iterable<Tag>):String
~ getGauge(id:Sring):String
~ getCounter(id:String):String
~ getTimer(id:String):String
- jsonParser(meter:Meter): String
~ getCustomGaugeList(): String
~ getCustomCounterList(): String
~ getCustom TimerList(): String
~ getSimpleMeterList(): String
~ getTaggedMeterList(): String
- mapJsonParser(map: Map<String, ?>): String
<<Controller Advice>>
RestAdvice
+ illegalArgumentExceptionHandler(iae: IllegalArgumentException): String
+ numberFormatExceptionHandler(nfe: NumberFormatException):String
<<REST Service>>
MetricsProviderRestService
- metricsProvider: MetricsProvider
+ MetricsProviderRestService(provider: MetricsProvider)
- getMeter(meterId:String, tags: List<Tag>):String
- parseCounterGaugeAndConfig(json: String): Map<String, String>
- parseTimer(json:String): Map<String, String>
+ <<GET>> getMeter(meterId:String, tags: MultivalueMap<String, String>) :String
+ <<GET>> getMeter(meterId:String):String
+ <<GET>> getGauge(gaugeId: String): String
+ <<GET>> getCounter(counterId: String):String
+ <<GET>> getTimer(timerId: String): String
+ <<GET>> getCustomGaugeList(): String
+ <<GET>> getCustomCounterList(): String
+ <<GET>> getCustom TimerList(): String
+ <<GET>> getSimpleMeterList(): String
+ <<GET>> getTaggedMeterList(): String
+ <<PUT>> changeMemoryBaseUnit(unit: String): void
+ <<PUT>> changeDiskBaseUnit(unit: String): void
+ <<PUT>> putGaugeValue(body:String): void
+ <<PUT>> putCounterValue(body:String): void
+ <<PUT>> putTimerValue(body:String): void
+ <<DELETE>> deleteGauge(gaugeId: String) : void
+ <<DELETE>> deleteCounter(counterId: String): void
+ <<DELETE>> deleteTimer(timerId: String): void
<<Enum>>
CapacityBaseUnit
+ BYTES
+ KILOBYTES
+ MEGABYTES
+ GIGABYTES
+TERABYTES
+ byteValue(): double
+ stringValue(): String

5.2.1 *Metrics Provider* implementation and the *Capacity Base Unit* enumeration

The *Metrics Provider* class was initially created as a "clone" of the class used in the previous example. Of course, this was not enough to suit our needs, but it would be a suitable starting point. Because of this we already have a few elements that remain from that initial point. First of all, the *Metrics Provider* is annotated as a Component so that the Spring Boot Application can locate it and treat it as a component, successfully injecting the dependencies we need, which brings us to the constructor, which receives the Micrometer-API *Meter Registry* instance created by Spring Boot as an argument. This is saved as an attribute, as all the important operations will require using this attribute in some way.

We also added the Configuration Properties annotation. This was done to allow the server-side to define configuration properties for the metrics provider directly from the *application.yml* file, which acts like a blueprint of sorts. As of now there are three properties that can be configured using this mechanism: the schedule rate, the memory base unit and the disk capacity base unit. The way this can be done is shown in the text below extracted from the aforementioned *application.*yml file. There will be a full example of this file in the appendix.

```
metricsprovider:
    schedulerrate: 3000
    memorybaseunit: kilobytes
    diskbaseunit: megabytes
```

The schedule rate, as we will see later on, is required for a scheduled task that runs in the *Metrics Provider*. This is the rate of invocation of the scheduled operation in milliseconds. This value is defaulted to two seconds in case it is not specified, due to the fact that 2 seconds should be enough to allow the component to "rest" after execution, but it could very likely be lowered to a much faster rate. The memory and disk capacity base units are created to allow the monitoring application to specify the base unit for the system metrics regarding physical memory and disk capacity. These are, by default, extracted as bytes, which can be difficult to manage if the value is too high. To avoid this, we created these two configuration properties. These two configuration properties can be modified at runtime, which will also allow the client-side application to customize this if needed. The scheduler rate, on the other hand, must be a final constant value, so this can only be modified in the *application.yml* file before execution.

To add some constraint to this mechanism, we created the ***Capacity Base Unit*** enumeration. This enumeration contains the valid base units that can be used, which include: bytes, kilobytes, megabytes, gigabytes and terabytes. Apart from allowing to safely parse the property indicated in the *application.yml* file into a valid value, this enumeration also adds a few useful operations. The first provides a value in bytes of the selected base unit, so that the extracted value can accurately be converted to the selected value. Additionally, there is another method that creates a String to add to the meter description, indicating the base unit following the Micrometer-API standards (a lowercase string with the corresponding base unit).

Returning once again to the *Metrics Provider*, we can also find three maps created as attributes of the class, one for each type of meter. This was done to store custom metrics when they are created. The decision of choosing a Map instead of another type of Collection like a list or an Array is due to the fact that every meter is identified using a URN or name. By using a map, we can access the custom metric by using that same name, allowing a uniform access as well as removing the dependency that would appear should we choose to use an index such as a number instead.

The Gauges' map does stand out as it doesn't map a *Gauge* object but an *Atomic Double* instead. There is a reason for this. Micrometer-API defines Gauges as values that are calculated upon request. If we create a *Gauge* instance, then this value cannot be modified. This is actually acknowledged by Micrometer-API, that added the possibility or registering a number as a Gauge. The registry acknowledges that number as a Gauge, providing the value when requested, while also allowing to modify the value. As to why an *Atomic Double* was used, after some initial experimentation and trial and error, it was determined that the number in question had to be an instance of a *java.util.concurrent* number or a subclass. This was because any other (for example, *Double*) would not register correctly and would always render a null or zero value when requested. A gauge returns a double value, so it was decided to use *Atomic Double* to maintain the type of number in use.

The class contains some methods to perform the basic CRUD[WP21] operations on the custom metrics. It is important to mention that the create and update operations were unified into a single one. The reason behind this is that the custom metrics can be accessed (and probably will be accessed) by a Spring Cloud Stream. If we separate these two operations, the Spring Cloud Stream would first have to check if the value exists, create it if it doesn't exist and/or update it if it does exist. This adds some extra work required on the server-side application using the Metrics Provider Component. We want to reduce that overhead as much as possible, so the create and update operation are unified into a single operation that runs the necessary checks without requiring the server-side application to do that.

After running some initial tests, it was discovered that the *Meter Registry* from Micrometer-API did not register some interesting metrics, namely the physical memory values and the disk capacity values. After snooping around the famous software forum, Stack Overflow[SE21], a method of obtaining this values was discovered. This involved using a *Operating System MX Bean*, which is the management interface for the Operating System where the Java Virtual Machine is running [O20], to obtain the metrics. This metrics where then registered onto the registry using the Micrometer-API naming convention (separating the words with dots). After some further tests however, this was proven to be somewhat of a problem efficiency-wise. As mentioned previously, a gauge is calculated upon request, meaning that every time the value was requested, all the calls to the Operative System (OS) had to be performed, which made the retrieval roughly between 2 or 3 times more costly than the other metrics. To solve this issue, the scheduled operation was created. The scheduled operation calculates the values on a separate thread every certain amount of time (the scheduler rate property previously mentioned) and stores the values in attributes that are registered as gauges. With this, the "real time" aspect of these measurements is lost to a "soft time" aspect, but the efficiency gained is worth this loss, especially considering that we can decide how much we are willing to sacrifice using the configurable property.

Now that we have introduced the scheduled operation, we have to explain a strange condition in it that initializes these added metrics to begin with. One would expect them to be registered by the constructor instead of this scheduled method, but there is a catch. Spring Boot first initializes the components and then retrieves the configuration properties. This means that, at the time the constructor is called, we do not have the configuration properties ready and, as a result, cannot register these metrics. This is the reason for this somewhat "ugly" addition to this operation.

To finish explaining this class, there are some methods marked as *protected*. These methods are marked that way to allow access only to the *Metrics Provider REST Service*. These methods are required to retrieve the data in a JSON format as well as modifying the configuration properties using the REST Service, as the system metrics registered in the scheduler have to first be de-registered and the re-registered again in order for the unit change to take effect. The JSON parser methods also rise a few eyebrows. These methods are strange as there already are plenty of JSON libraries that allow the creation of a JSON Object without having to write it manually as these methods do. We were, however, left with no choice. These classes run inside a framework that, as of now, still has a JSON Object parser that is "too green" to actually be able to parse a JSON Object. For this reason, it has to be done manually. We could say this methods were added as a quick-fix for the current state of the framework. If the framework updated this issue in the future, these methods could then provide a proper JSON object instead, reducing the complexity of the code considerably.

5.2.2 Implementation of the *Metrics Provider REST Service* and the *REST Advice*

After the *Metrics Provider* class was implemented, we could now access the metrics using an endpoint provided by the Spring Boot Actuator that runs within the Spring Boot Application. This was a good starting point to retrieve the metrics, but we now required a way to modify these values remotely. This would allow us to register all custom metrics within a single *Metrics Provider* if needed, as well as also letting a client-side application to modify the configuration properties that allow this modification. Seeing that the preferred method of exposing these metrics by the Spring Boot Actuator is by using an endpoint accessible by HTTP requests, the most reasonable approach was to create a REST service that supported both reading and modifying operations on the custom meters. This REST service would also end up taking over the original endpoint so the *application.yml* file can be simplified as well as having a single access point for the HTTP requests, independently of their intentions.

The *Metrics Provider REST Service* class implements this RESTful service mentioned in the above paragraph. As we could have guessed, the Spring Framework already has a framework implementation for a REST service, annotated as *REST Controller*. With this annotation, this class will now be instantiated by the Spring Boot Application automatically, just like the *Metrics Provider*. As a matter of facts, a reference to the instance of the *Metrics Provider* created by the Spring Boot Application is injected into this one through the constructor. With this connection between both instances, we can now access the *Metrics Provider*'s metrics using the class implementing the REST Service. This class, therefore, acts as a link between the *Metrics Provider* and the "outside world".

This service, initially including solely the functionality regarding the custom meters, has an endpoint for each of the type of meter. For each meter, an HTTP client can retrieve a list of the meters of that specific type, modify the value or add a new custom meter to the *Metrics Provider* or delete a custom meter. This is all done using the appropriate HTTP request type to the corresponding endpoints. The update or create operation works just like the one in the *Metrics Provider*, unifying the create and update operations. The JSON object expected, however, can be a bit puzzling at first. The format of said JSON object as well as its reason to be is explained by the different *Meter Representations* found in the client-side subsection.

Apart from the custom metrics, the REST service also allows the client-side application to modify the base units for the memory and disk capacity system metrics. To do so, this methods expect a very simple JSON Object containing a single attribute called "unit". This attribute has a String value attached that is equal to one of the valid units present in the *Capacity Base Unit* enumeration. This methods will then deregister and re-register the corresponding metrics to successfully update the values.

Once again, we can observe some strange operations performed to parse a JSON Object manually. Once again, this is due to the limitations of the library's current state. In this case we use a map to read the JSON and obtain its values manually. The best option would be to use an actual JSON library, once this issue has been solved in a future update, but for now this quick-fix implementation allows the component to work.

To finish with this class, there are another two endpoints created: *simple-meter* and *tagged-meter*. With these two endpoints, the */actuator/metrics* endpoint used by the Spring Boot Application to expose the metrics could be removed from the *application.yml* file and allow all the operations to be performed through the REST service. The idea was, initially, to have a single endpoint called *meters*, however, we once again encountered a limitation that made this impossible. Due to the way the REST service (or rather, REST controller) is implemented by the Spring Framework, we either have a request that receives at least one request parameter or that doesn't receive any. This was an issue as some of the metrics registered in Micrometer's *Meter Registry* have tags that are requested using request parameters. Because of this, and after several attempts made to unify both endpoints, there was no other option than having an endpoint for meters with tags (*tagged-meter*) and another one for meters without tags (*simple-meter*).

To finish off this part of the component, we will mention the *REST Advice* class. This class was created following the implementation standard portrayed by the Spring Framework's approach to a REST service. As well as a *REST Controller* a REST service created with the Spring Framework should also have a *Controller Advice*. The function served by this class is to implement an error handler for the controller, in our case, the *REST Controller*. If the client-side application sends a request that is not valid, this handler will capture the produced exception and reply with an error response containing the message of the exception that was triggered. For example, if the user tries to modify a custom gauge, but the JSON object is malformed, this class will respond with a *401 HTTP Response* and a message explaining what was not valid.

This class is expected to handle two exceptions: *IllegalArgumentException*s and *NumberFormatExceptions*. The first type of exception can be produced in multiple places if the request's body has an invalid field or, in different words, an illegal value in one of its attributes. The *NumberFormatException* can be triggered when attempting to parse the numbers sent through HTTP as a String into the number they represent.

Further information about the REST Service provided by these two classes can be found in the appendix.

5.2.3 Dependencies of the server-side

To finish this section, we will talk about the dependencies that the server-side has. This information is represented graphically in figure 9, to be able to see this information in one glance.

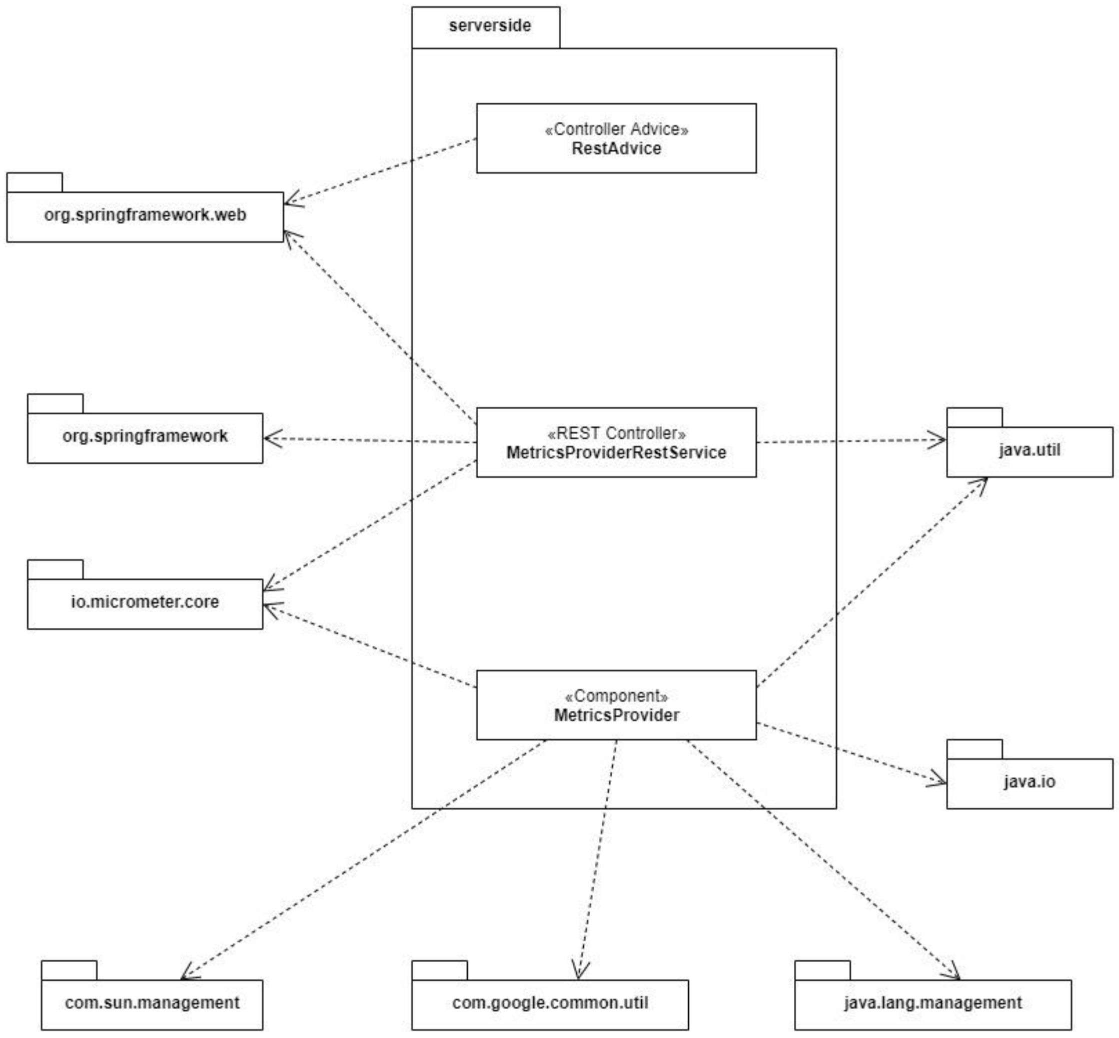


Figure 10: Dependency diagram of the server-side

Starting the diagram from top to bottom, the *REST Advice* only has dependencies with the *org.springframework.web*. This library includes the annotations required by the Spring Framework to correctly indicate that this class is a controller advice as well as indicating which exception is handled by each of the different methods. We also require this annotations to be able to define the response status and to define that the response of the method is delivered as a response body. Finally, we also need the enumeration *HttpStatus* to use it alongside the corresponding annotation to indicate the status code for the response (in this case all are client errors, but with a few differences).

The next class, the *Metrics Provider REST Service*, also has that dependency with *org.springframework.web* for similar reasons as the advice. The annotations that mark that this is a REST Controller, as well as defining the HTTP status and indicating that the result of the operations (if it has any) is sent as a response body are all done in the same way as for the advice. There are some further dependencies with this library which involve the annotations that are used to determine what path and HTTP request type (GET, PUT, DELETE, POST) is handled by each method as well as a pair of annotations indicating if the input parameters of the method are present in the request's body, the request's path or if it is a request parameter.

This class also has a dependency with the *io.micrometer.core* library that is required to parse the request parameters in a request of a tagged meter into a tag object readable by Micrometer-API. A last dependency involving java's *java.util* library is also established mainly to be able to use the *Array List* class to dynamically store the tags from the previously mentioned tagged meter request, and the *concurrent Time Unit* class to be able to record time with a custom timer if the update of a custom timer is requested. This library is also currently in use to manually parse a JSON Object into a map to extract the values. Needless to say, in a hopefully near future, when JSON is supported by the framework's virtual servers, this last dependency should be changed for a dependency with the *javax.json* library to parse the JSON Objects the way they are meant to be parsed.

The last class we haven't covered is *Metrics Provider*. This class has a clear dependency with the *io.micrometer.core* library as it used the Micrometer-API *Meter Registry* as well as the different Meters that it can have (Gauge, Timer, Meter and Counter). This library is also required to retrieve the measurements recorded by the meters, read the tags for the tagged metrics when we wish to request them and handle the exception that may occur if the search for a requested metric fails (mostly due to it not existing in the registry). Apart from the Micrometer-API library, we also require the *org.springframework* library containing the annotations we need to add to mark this class as a Spring Component, enable the scheduled operation, and import the configuration properties from the *application.yml* file.

Apart from these two libraries, we will require the *com.sun.management*, *java.lang.*management and *java.io* libraries to be able to use the *Operating System MX Bean* required to extract the memory and disk metrics and the *File* object to read the disk metrics, the *com.google.common.util* library to define an *Atomic Double* we need for the custom gauges; and the *java.util* library to define and use the maps where we store the custom meters, to have access to the *Time Unit* enumeration for time recording and be able to define an *Supplier* reference for the time recording methods that require this reference to a method we what to record.

5.3 Implementation of the client-side

After the server-side was implemented and it was verified that it worked, the next logical step to make was to implement a way to easily access the information exposed by the server-side in a comfortable and convenient manner. With the server-side working as intended, any REST Client or, even simpler, any HTTP client could access the information exposed by the server-side. Of course, the problem with this approach was that both the AAS and HTTP client had to be created in each of the client-side applications and find their own way of parsing the JSON Object containing the requested data. This would mean a lot of repetition of the same type of work as well as the possibility of two client-side applications taking different approaches to the same problem and, as a result, coming up with different solutions that, although do the same thing, might be somehow incompatible and cause conflicts further on.

To avoid this issue, it was decided to work on this second aspect of the monitoring process as soon as the metrics had been exposed, even prior to the existence of a proper REST service. The work done by the client-side was divided into two big groups: the meter representations and the metrics AAS. The first one to be worked on was the meter representations, created before the REST service existed to parse the meters exposed on the Spring Boot Actuator endpoint into object representations that could be easily handled by the client-side application. The second one, created after the REST service was working, is a factory of sorts that allows the client-side application to create the AAS directly as well as add any custom meters they might want to include, so all the client-side applications have the exact same type of monitoring AAS.

Just like with the *Metrics Provider*, this work was inspired by another prototype that was provided from the IIP-Ecosphere project at the beginning of this project's work, that was used to get familiar with the concept of AAS and the implementation of AAS done by the IIP-Ecosphere project. The use of a REST client was later introduced inspired partly by previous university work done in the field of REST services, this time including a simple Jersey[EF18] client to access a REST service. The idea, that would at the end be the one used, was to map the different meters as AAS properties, and internally define the get operation as an HTTP get request using a Jersey client. If this worked, it could later be extended to the use of REST to modify custom meters in a similar fashion.

With this in mind, we will delve deeper into the details of the client-side of the component, starting with the meter representations and slowly making our way towards the metrics AAS package.

5.3.1 Meter representations

This package is composed of four simple classes and its name is quite self explanatory. The data from the *Metrics Provider* is exposed as a JSON Object in a rather complicated manner, as it can be seen in the JSON Objects found in the appendix, so the use of this raw data by itself without further work seemed to be unadvisable, to say the least. It was clear that we required some sort of Object representation of the extracted metric so we could use it and read it as such in a comfortable way. The first idea was, naturally, to look into Micrometer-API, as the meters we were using were theirs to begin with. We found a partial solution to our issue within the library: all of the meters provided by Micrometer-API were represented using an interface. There were four particular interfaces: *Meter*, *Counter*, *Gauge* and *Timer*; one for each of the specific meter types, as well as a generic superclass containing common operations to all of them. With this realization, we decided that we would use these interfaces in the client-side to represent the retrieved meters.

The issue we encountered is due to Java not allowing, for obvious reasons, the instantiation of an interface directly. We need to instantiate a class that implements that interface. Micrometer-API beautifully uses the protection provided by the interface mechanism by never providing the implementing class and always providing the object masked by the interface instead. Although this is a very good and correct approach from a software point of view, we depended on the factories provided by Micrometer-API to instantiate an object the way it was done normally, like on the server-side, for example. This was instantly rejected as the factories provided by the Micrometer-API only allow the creation of a new meter and not one with values. With the *Counters* and *Gauges*, this wasn't much of an issue, to be honest: we could simply create a new counter and gauge and set the value to the retrieved value. The big problem came with the *Timer*, which internally stores a series of different variables that cannot be directly modified. As the issue with the Timer was encountered, we decided that the best approach would be to create a set of classes implementing the interfaces so we could create our own meter objects the way we needed to create them, with the retrieved values.

After taking what we saw in the previous paragraph into account, it is very important to note that these classes were implemented with the intention to represent a meter retrieved from the *Metrics Provider* and, although they can be used in the client side to obtain and update values, it is strongly recommended to refrain from doing so unless strictly necessary. The reason behind this is, since the real meters are in the server-side, the update might be "dangerous". The meters retrieved from the *Metrics Provider* might be modified while we are reading them, as they are still being used to monitor the resource meanwhile. This means that, when requesting an update, it is quite possible that these values have already changed when the update request arrives, and said update will always affect the current state of the meter, not the one we initially read. There are, of course, exceptions to this. If we are using custom meters created exclusively by client-side applications to monitor, for example, the time taken to retrieve a specific set of values, and this timer is never modified by the *Metrics Provider* at any moment, then it is perfectly safe to request updates. An example will be provided when we delve into how to use the client-side of the component.

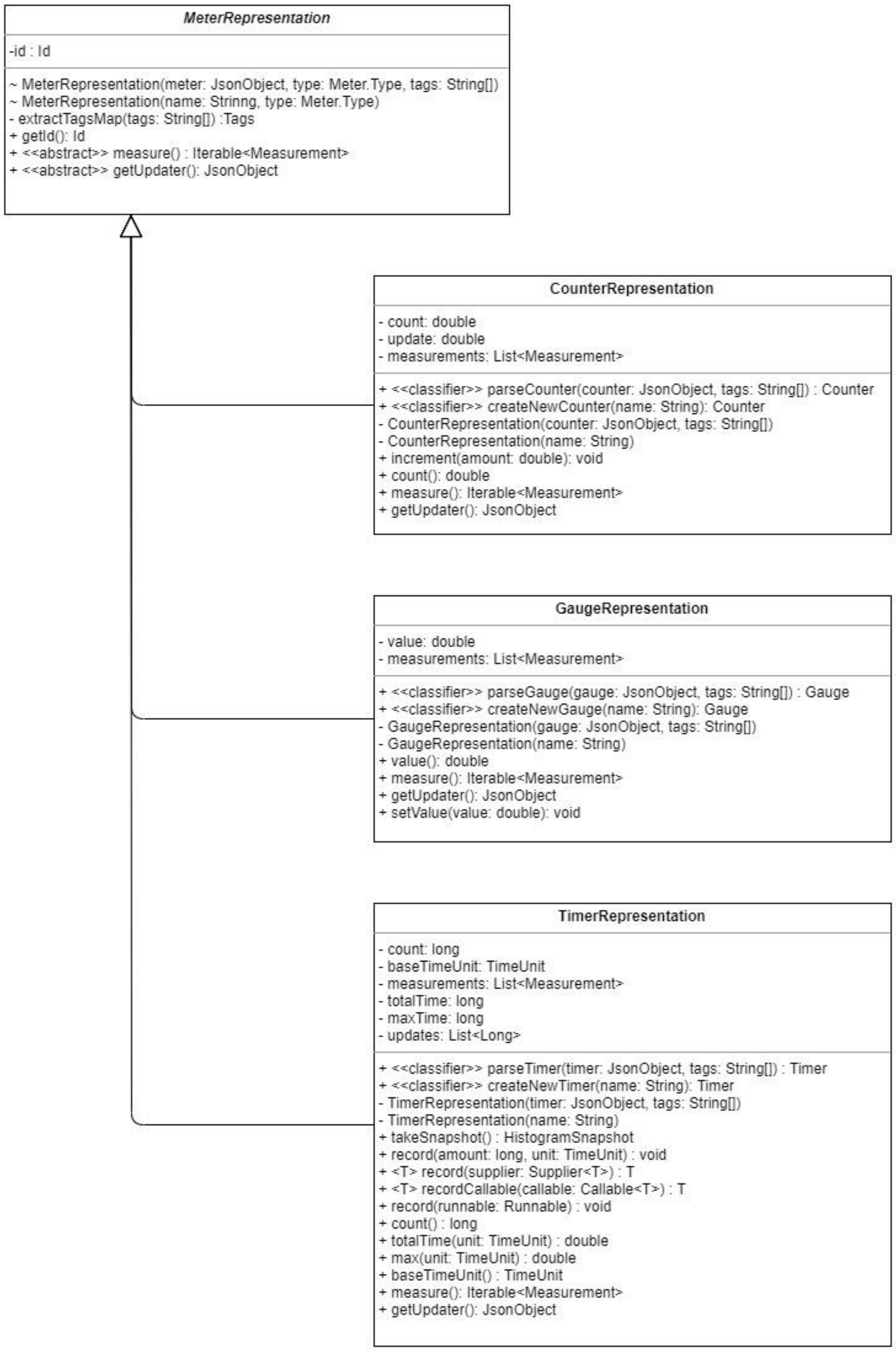

MeterRepresentation
-id : Id
~ MeterRepresentation(meter: JsonObject, type: Meter.Type, tags: String[])
~ MeterRepresentation(name: Strinng, type: Meter.Type)
- extractTagsMap(tags: String[]) :Tags
+ getId(): Id
+ <<abstract>> measure() : Iterable<Measurement>
+ <<abstract>> getUpdater(): JsonObject
CounterRepresentation
- count: double
- update: double
- measurements: List<Measurement>
+ <<classifier>> parseCounter(counter: JsonObject, tags: String[]) : Counter
+ <<classifier>> createNewCounter(name: String): Counter
- CounterRepresentation(counter: JsonObject, tags: String[])
- CounterRepresentation(name: String)
+ increment(amount: double): void
+ count(): double
+ measure(): Iterable<Measurement>
+ getUpdater(): JsonObject
GaugeRepresentation
- value: double
- measurements: List<Measurement>
+ <<classifier>> parseGauge(gauge: JsonObject, tags: String[]) : Gauge
+ <<classifier>> createNewGauge(name: String): Gauge
- GaugeRepresentation(gauge: JsonObject, tags: String[])
- GaugeRepresentation(name: String)
+ value(): double
+ measure(): Iterable<Measurement>
+ getUpdater(): JsonObject
+ setValue(value: double): void
TimerRepresentation
- count: long
- baseTimeUnit: TimeUnit
- measurements: List<Measurement>
- totalTime: long
- maxTime: long
- updates: List<Long>
+ <<classifier>> parseTimer(timer: JsonObject, tags: String[]) : Timer
+ <<classifier>> createNewTimer(name: String): Timer
- TimerRepresentation(timer: JsonObject, tags: String[])
- TimerRepresentation(name: String)
+ takeSnapshot() : HistogramSnapshot
+ record(amount: long, unit: TimeUnit) : void
+ <T> record(supplier: Supplier<T>) : T
+ <T> recordCallable(callable: Callable<T>) : T
+ record(runnable: Runnable) : void
+ count() : long
+ totalTime(unit: TimeUnit) : double
+ max(unit: TimeUnit) : double
+ baseTimeUnit() : TimeUnit
+ measure(): Iterable<Measurement>
+ getUpdater(): JsonObject

Figure 11: Detailed class diagram for the meter representation package

5.3.1.1 Implementation of the *Meter Representation*

The first class we will talk about is *Meter Representation*. This class is an abstract class, as all of the meters have a specific type and cannot just simply be a "meter". We do, however, require the existence of this class as there are a series of common variables and operations that all the meters have, especially regarding the ID. The constructors of the *Meter Representation* are protected as the should only be accessed by the subclasses to correctly add the ID, which includes the URN of the meter, the base unit name of the value it records and the tags if it has any. About the tags we must add that, after some tests that were initially run with a REST Client, we discovered that the JSON Object itself does not store the tags when requested. For this reason, we must indicate the tags used to retrieve the value as a parameter, which are expected to be in their HTTP request format and, thus, need to be parsed into the correct Object representation to be added to the ID. This is simple as each of the tags has a simple format composed of a set of key-value pairs separated by a two dots. The second constructor is to simply create a new *Meter* Representation with just the name and the type of meter, as there can be meters created in the client-side instead of the server-side. This class will provide the functionality required to retrieve the ID. There is a method regarding the retrieval of the measurements is left as abstract as each specific meter uses a different statistic and might, in case of the *Timer*, render multiple measurements.

We also added an abstract method specific to our meter representations to declare the need of having a function that retrieves a JSON object with the required information to update the meters using the REST Service. Unlike conventional REST services, we cannot update the custom meter by parsing the entire Object into a JSON Object and sending it to the REST service. This is, due to each of the meters having a peculiarity that made this process impossible. We will discuss these peculiarities in each of the meter representations as we will have to go over the method in each of the meter representations.

5.3.1.2 Implementation of the *Counter Representation*

As the name implies, the *Counter Representation* implements the *Counter* interface by representing a retrieved counter meter. The constructor of this class is private, as the client-side application should always use the *parse counter* method to create an instance (which in term masks it with the *Counter* interface. When the JSON Object representing a *Counter* is retrieved, the *Counter Representation* stores the count value as an attribute and sets the *Measurement's* statistic to *COUNT*. This class implements all the required methods from the *Counter* interface, allowing the user to increment the counter value and read it as if it were an actual counter and not the representation of a counter retrieved from the *Metrics Provider*.

The peculiarity of the counter is that the value can only be updated by incrementing the current value, meaning that the value cannot be directly set. For these reason, there is an attribute that will update in the same way as the counter that is initialized with a zero value. When the JSON Object updater is requested, a JSON Object is created containing two attributes: the name of the counter, and the increment it has had in the client-side. If an PUT request is sent with this update, the server-side will use the name to locate the custom counter and update its value by incrementing it the amount indicated as increment value.

If the client-side is creating a new probe that is not present in the server-side, they will have to use the *create new counter* method. This will create a completely new counter to be used in the client-side and in order to have a valid updater to push later into the server-side.

5.3.1.3 Implementation of the *Gauge Representation*

This class implements the *Gauge* interface by proving a representation of a retrieved gauge meter. It was called *Gauge Representation* for this reason. Just like with the *Counter Representation*, the constructor is private to ensure that the *parse gauge* method is called instead, masking this representation under the *Gauge* interface.

The *Gauge Representation* will store the value of the gauge found in the JSON Object and will set the *Measurement's* statistic to *VALUE*. Just like before, this class implements all the required methods from the corresponding interface, in this case, *Gauge*. In the case of a gauge, the interface only specifies that the retrieval of the value is required. As we explained earlier in this document, gauges are supposed to calculate the value, so are usually attached to an operation rather than a numeric value. Unable to know what is used to calculate the value, instead, we have added a *set value* method in the *Gauge Representation* class to allow the client-side to modify the value the same as it is done in the server-side.

The gauge's peculiarity is that one: not really being able to update the value. Of course, once this problem is taken care of by adding the *set value* method to the *Gauge Representation*, we could simply parse the entire *Gauge* and send it to the REST service to update the value there as well. However, as both the timer and the counter's peculiarities make that impossible for those two other meter types, it was decided that it was best to keep the same format of updating JSON Object for the *Gauge Representation* as well for consistency purposes. Once this JSON Object is received by the server-side, similarly to the counters, the REST service looks for the custom gauge and updates the value to the one indicated by the JSON Object.

Just like with the counter, if the client-side is creating a new probe that is not present in the server-side, they will have to use the *create new gauge* method, which will create a new gauge to be used in the client-side and in order to have a valid updater to push later into the server-side.

5.3.1.4 Implementation of the *Timer Representation*

The final meter type is the timer. The *Timer* interface from Micrometer-API is, by far, the most complex of all the different meters. We used the *Timer Representation* to implement this interface in our component. Like our other meter representations, we need to use the *parse timer* method to obtain an instance of this class masked by the *Timer* interface. Just like the other classes before it, the constructor is private.

From the JSON Object we retrieve, we set the count and base time unit directly from the JSON Object. After some digging into the way the *Timer* from Micrometer-API works using the *Time Unit* enumeration and paying attention to the method that allows us to record an amount of time directly by specifying a long value and a time unit, it was decided that the best way to store this data and allow the client-side to use the timer without losing any information was to store the maximum time and total time as a long value with the smallest unit possible, in this case, nanoseconds. With this, whenever we require the time measurements, we can retrieve them by specifying the unit, just like the *Timer* interface requires, and not risk losing excessive information on the way.

Unlike the other meters, a timer has three *Measurements*. The first measurement is the number of times this timer was used, and has a long value with the *COUNT* statistic. The second one is the total time recorded. This value is a double value represented in the timer's base unit with the *TOTAL_TIME* statistic. The last measurement is the maximum time recorded by the timer, which is mapped into a *Measurement* using the *MAX* statistic.

The *Timer Representation* will allow the client-side to use the object as a timer, so all the methods required by the Timer interface are implemented to provide this functionality. The *take snapshot* method is the only one that has been created using a basic *Histogram Snapshot* from the Micrometer-API library, including only the basic information required. As a result, the histogram provided is probably not the best option, but this method was never intended to be used by this component or the client-side, so a basic implementation was produced. It is also important to note, as it will be shown in the corresponding section, that the test done to this method is very basic. As well as a *Histogram Snapshot*, the client-side can use this class to retrieve the count of times the timer has been used, the base time unit used by the timer, and the total and maximum times recorded by the timer after specifying the time unit we want that data in.

This class will also use the basic Micrometer-API timers to record a callable, a supplier or a runnable. All of these methods require using a method also required by the *Timer* interface which involves recording a specific amount of time into the timer by specifying the amount and the time unit it has been recorded in. This means that all recording operations will, eventually, end up using this particular method to save the time and update the values. This proved to be key to be able to store the recordings performed by this timer while in the client side. Every time a recording is made, it is added to a dynamic list in nanoseconds and stored as an attribute. The reason behind this is the timer's peculiarity, that was the one that caused the most trouble due to its complexity to solve.

As we mentioned before, the individual values stored in a timer (count, maximum time and total time) cannot be set separately, they can only be done by recording an amount of time with the timer that will, in turn, update the counter and maximum recorded time as the recorded time is added to the total amount or recorded time. As we saw in the previous paragraph, once we discovered the common method used by all the recording methods, we decided to do the same when it came to updating the "real" timer in the *Metrics Provider* via REST. As we said, all the recordings are stored in nanoseconds within a dynamic list whenever the recording methods are called. When we ask for the JSON Object used to update, the first attribute will, of course be the name of the timer, but the second attribute will be a JSON Array created as a result of the aforementioned dynamic list. This JSON Object, when received by the REST Service will trigger the service to retrieve the timer and proceed to record each of the amounts indicated by the JSON Array. As inefficient as this seems, this seemed to be the only plausible solution that could be used to solve this issue. Of course, it is safe to assume that the update of this particular type of meter is the most costly, just like the use of the timer is the most costly of them all as defined by the Micrometer-API[PS21]. The use of this timers outside the *Metrics Provider* is, as a result, strongly unadvisable, even more than any of the other meter representations.

Similarly to the other two meters, we also include another classifier method called *create new timer*, allowing us to create a brand new timer in the client-side to be later pushed into the server side.

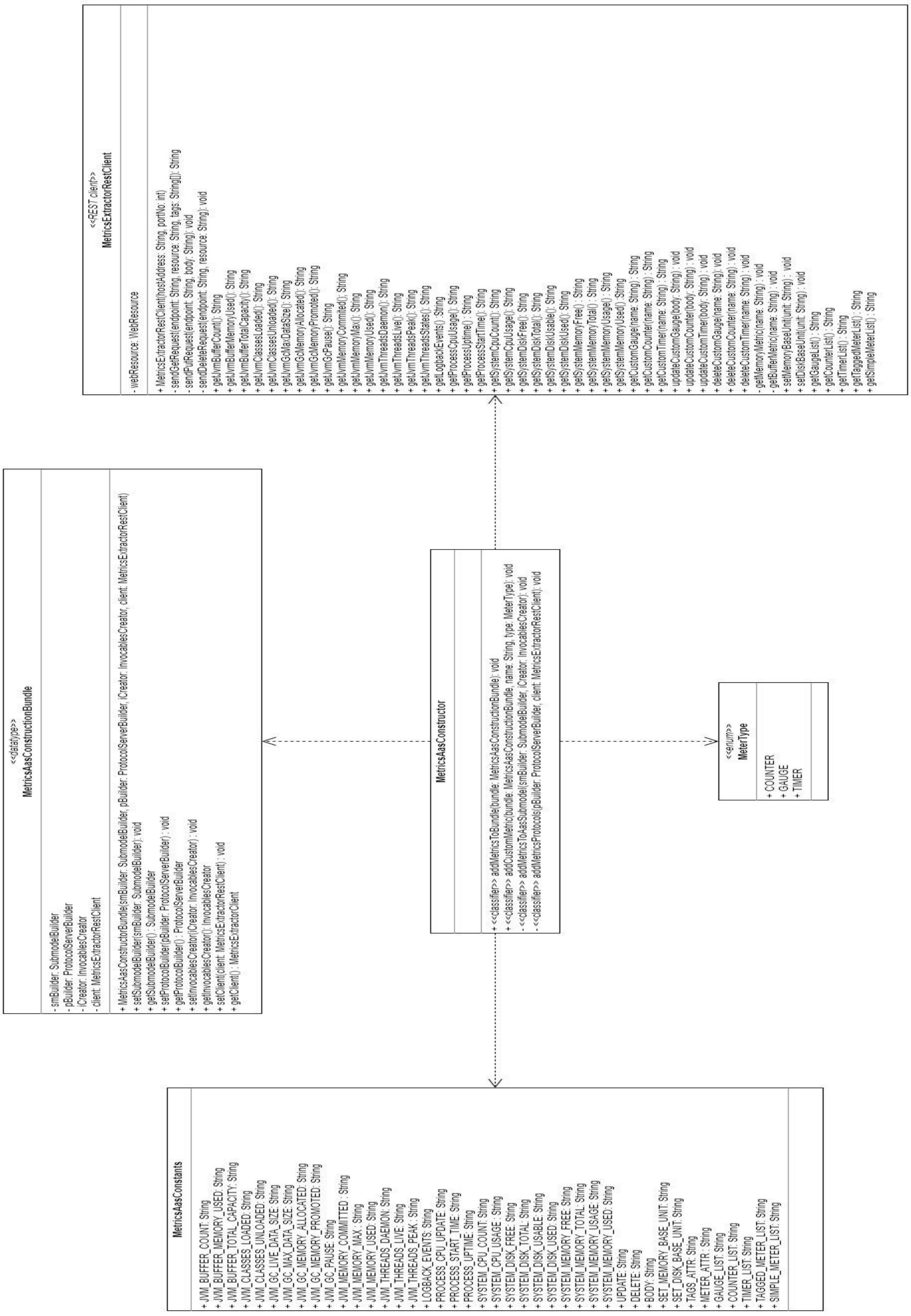
5.3.2 Metrics AAS

This second package within the client-side division of the Metrics Provider component is composed of four classes and an enumeration. Once we have the data from the *Metrics Provider* exposed on an endpoint and controlled by a REST service, we need to retrieve this data. Of course, the best way to retrieve the data from a REST service is using an HTTP RESTful client. Following the IIP-Ecosphere's requirements of using an AAS, the sensible decision here is to map the metrics of the service we are monitoring as properties of the asset and then retrieve said properties using the RESTful client. This does seem like the best approach as, in spite of all the underlying operations to retrieve the meter's value, we can create the illusion that its simple retrieving a simple property like it would read any other property mapped into an AAS.

Before the implementation of this package, all this effort of creating a RESTful client and an AAS mapping the metrics as properties was all done by the client-side. This was tested out using a small prototype initially created to test the meter representations. However, after several executions and changes, the idea of creating a template of sorts that already provided this functionality that was so obviously required for the metrics provider component became clear. Based on this previously created prototype, this package was created from scratch to have a "fresh start" to this template. The goal was originally to create two classes: one containing the definition for the AAS, which we would call *Metrics AAS Constructor*; and a second class that would implement the RESTful client, as a result containing the functionality required by the AAS to retrieve the data.

As these two classes were implemented, further implementation decisions were made as new issues and limitations arose. Some other decisions were made as, due to the work carried out on this classes, some features that could be interesting to have also from the client-side perspective started to become visible. The end result would be this package, with double the amount of classes as originally planned as well as an enumeration. Although one of the classes (the

Metrics AAS Constants class) is more of an enumeration than a class, we will explain the different elements contained in this package in the subsequent sub-subsections.



<<REST client>>
MetricsExtractorRestClient
- webResource: WebResource
+ MetricsExtractorRestClient(hostAddress: String, portNo: int)
- sendGetRequest(endpoint: String, resource: String, tags: String[]): String
- sendPutRequest(endpoint: String, body: String): void
- sendDeleteRequest(endpoint: String, resource: String): void
+ getJvmBufferCount(): String
+ getJvmBufferMemoryUsed(): String
+ getJvmBufferTotalCapacity(): String
+ getJvmClassesLoaded(): String
+ getJvmClassesUnloaded(): String
+ getJvmGcMaxDataSize(): String
+ getJvmGcMemoryAllocated(): String
+ getJvmGcMemoryPromoted(): String
+ getJvmGcPause(): String
+ getJvmMemoryCommited(); String
+ getJvmMemoryMax(): String
+ getJvmMemoryUsed(): String
+ getJvmThreadsDaemon(): String
+ getJvmThreadsLive(): String
+ getJvmThreadsPeak(): String
+ getJvmThreadsStates(): String
+ getLogbackEvents() :String
+ getProcessCpuUsage(): String
+ getProcessUptime() : String
+ getProcessStartTime(): String
+ getSystemCpuCount(): String
+ getSystemCpuUsage(): String
+ getSystemDiskFree(): String
+ getSystemDiskTotal(): String
+ getSystemDiskUsable(): String
+ getSystemDiskUsed(): String
+ getSystemMemoryFree() : String
+ getSystemMemoryTotal() : String
+ getSystemMemoryUsage() : String
+ getSystemMemoryUsed() :String
+ getCustomGauge(name: String) : String
+ getCustomCounter(name: String) : String
+ getCustomTimer(name: String) : String
+ updateCustomGauge(body: String) : void
+ updateCustomCounter(body: String) : void
+ updateCustomTimer(body: String) : void
+ deleteCustomGauge(name: String): void
+ deleteCustomCounter(name: String) : void
+ deleteCustomTimer(name: String) : void
- getMemoryMetric(name: String) : void
- getBufferMetric(name: String) : void
+ setMemoryBaseUnit(unit: String) : void
+ setDiskBaseUnit(unit: String) : void
+ getGaugeList() : String
+ getCounterList() : String
+ getTimerList() : String
+ getTaggedMeterList() : String
+ getSimpleMeterList() : String
<<datatype>>
MetricsAasConstructionBundle
- smBuilder: SubmodelBuilder
- pBuilder: ProtocolServerBuilder
- iCreator: InvocablesCreator
- client: MetricsExtractorRestClient
+ MetricsAasConstructorBundle(smBuilder: SubmodelBuilder, pBuilder: ProtocolServerBuilder, iCreator: InvocablesCreator, client: MetricsExtractorRestClient)
+ setSubmodelBuilder(smBuilder: SubmodelBuilder): void
+ getSubmodelBuilder() : SubmodelBuilder
+ setProtocolBuilder(pBuilder: ProtocolServerBuilder) : void
+ getProtocolBuilder() : ProtocolServerBuilder
+ setInvocablesCreator(iCreator: InvocablesCreator) : void
+ getInvocablesCreator(): InvocablesCreator
+ setClient(client: MetricsExtractorRestClient) : void
+ getClient() : MetricsExtractorClient
MetricsAasConstructor
+ <<classifier>> addMetricsToBundle(bundle: MetricsAasConstructionBundle): void
+ <<classifier>> addCustomMetric(bundle: MetricsAasConstructionBundle, name: String, type: MeterType): void
- <<classifier>> addMetricsToAasSubmodel(smBuilder: SubmodelBuilder, iCreator: InvocablesCreator): void
- <<classifier>> addMetricsProtocols(pBuilder: ProtocolServerBuilder, client: MetricsExtractorRestClient): void
<<enum>>
MeterType
+ COUNTER
+ GAUGE
+ TIMER
MetricsAasConstants
+ JVM_BUFFER_COUNT: String
+ JVM_BUFFER_MEMORY_USED: String
+ JVM_BUFFER_TOTAL_CAPACITY: String
+ JVM_CLASSES_LOADED: String
+ JVM_CLASSES_UNLOADED: String
+ JVM_GC_LIVE_DATA_SIZE: String
+ JVM_GC_MAX_DATA_SIZE: String
+ JVM_GC_MEMORY_ALLOCATED: String
+ JVM_GC_MEMORY_PROMOTED: String
+ JVM_GC_PAUSE: String
+ JVM_MEMORY_COMMITTED : String
+ JVM_MEMORY_MAX : String
+ JVM_MEMORY_USED: String
+ JVM_THREADS_DAEMON: String
+ JVM_THREADS_LIVE: String
+ JVM_THREADS_PEAK : String
+ LOGBACK_EVENTS: String
+ PROCESS_CPU_UPDATE: String
+ PROCESS_START_TIME: String
+ PROCESS_UPTIME: String
+ SYSTEM_CPU_COUNT: String
+ SYSTEM_CPU_USAGE : String
+ SYSTEM_DISK_FREE: String
+ SYSTEM_DISK_TOTAL: String
+ SYSTEM_DISK_USABLE: String
+ SYSTEM_DISK_USED: String
+ SYSTEM_MEMORY_FREE: String
+ SYSTEM_MEMORY_TOTAL: String
+ SYSTEM_MEMORY_USAGE: String
+ SYSTEM_MEMORY_USED: String
+ UPDATE: String
+ DELETE: String
+ BODY: String
+ SET_MEMORY_BASE_UNIT: String
+ SET_DISK_BASE_UNIT: String
+ TAGS_ATTR: String
+ METER_ATTR : String
+ GAUGE_LIST: String
+ COUNTER_LIST: String
+ TIMER_LIST: String
+ TAGGED_METER_LIST: String
+ SIMPLE_METER_LIST: String

Figure 12: Detailed class diagram for the metrics AAS package

5.3.2.1 Implementation of the *Metrics AAS Constructor* and use of the *Meter Type* enumeration

This is the first of the two classes that were initially intended to be a part of this package. The idea inspiring this class is to have a factory or a constructor of some kind that adds the meters exposed by the *Metrics Provider* into an AAS. This would ensure that all applications using the client-side of this component worked in the exact same manner, as well as reducing the workload of this client-side applications, which would otherwise have to know all the details behind retrieving the metrics and having to implement the entire retrieval operation.

The first detail to take into consideration is that all the methods provided by this class are of classifier scope. This means that the class does not need to be instantiated. There was quite some thought put into this decision as there seems to be some elements that could easily fit into the idea of instance attributes. A different approach to this was taken however, after considering the idea that perhaps we would be interested in adding this properties and operations into an already existing AAS, instead of creating a brand new AAS from scratch with only the metrics. For this reason, it was decided that this class would be stateless and, as such, not have any attributes or instances for the matter.

The first and most important method is the *add metrics to bundle* method. This method expects to receive a *Metrics AAS Construction Bundle*, a data-type created during the implementation of this class that we will explain after this section. This method has two parts, which were separated into private methods for a clearer vision as well as for maintenance purposes. The first part consists on adding the metrics to the submodel of the AAS that will administer the asset we are monitoring. Although there were different possibilities to this, considering that the system metrics are read-only data, we chose to map metrics as properties. This would mean that the state of the resource we are monitoring can be read as any other type of property. Needless to say, all of these properties are read-only and cannot be modified, so there is no possibility of setting the property to a different value.

An important consideration taken here is that the return type of these properties will be a String. The IIP-Ecosphere Framework for creating AASs is very comfortable indeed, but requires the programmer to define the return type of the method, which would be more beneficial than detrimental in most cases, but sets an important limitation in this aspect. The ideal would be to return an already parsed and ready to go meter representation of the corresponding type upon request. However, as this specific type is not one of the options defined by the BaSyx AAS specs for building AASs, we had to switch to the previous state of this object, which would be an unparsed version of it in JSON format. This again, proved to be an issue for two reasons: the first reason being that JSON is still not one of the valid accepted types for the IIP-Ecosphere Framework, which as of now does the exacts same thing we did in this project; and the second reason being that the JSON library used in the AAS server is still not fully built, so attempting to use a JSON library to perform operations within the AAS will cause it to crash. After this, we have no other choice than to return the raw String representing a JSON that, in turn, represents the meter we have retrieved.

Apart from these properties, there will be two operations added to allow the modification of the configurable properties of the *Metrics Provider*, namely the memory and disk capacity base units. Just like with the properties, even though the ideal would be to send a JSON Object with the new unit within its body, we have to first parse it into a String to be able to send it to the server. It is an additional operation that has to be carried out, unfortunately, but its small enough to overlook it for now in the current state of the overall project. This sets the path to an update in the future that allows the use of JSON in this method. For now, the String will suffice.

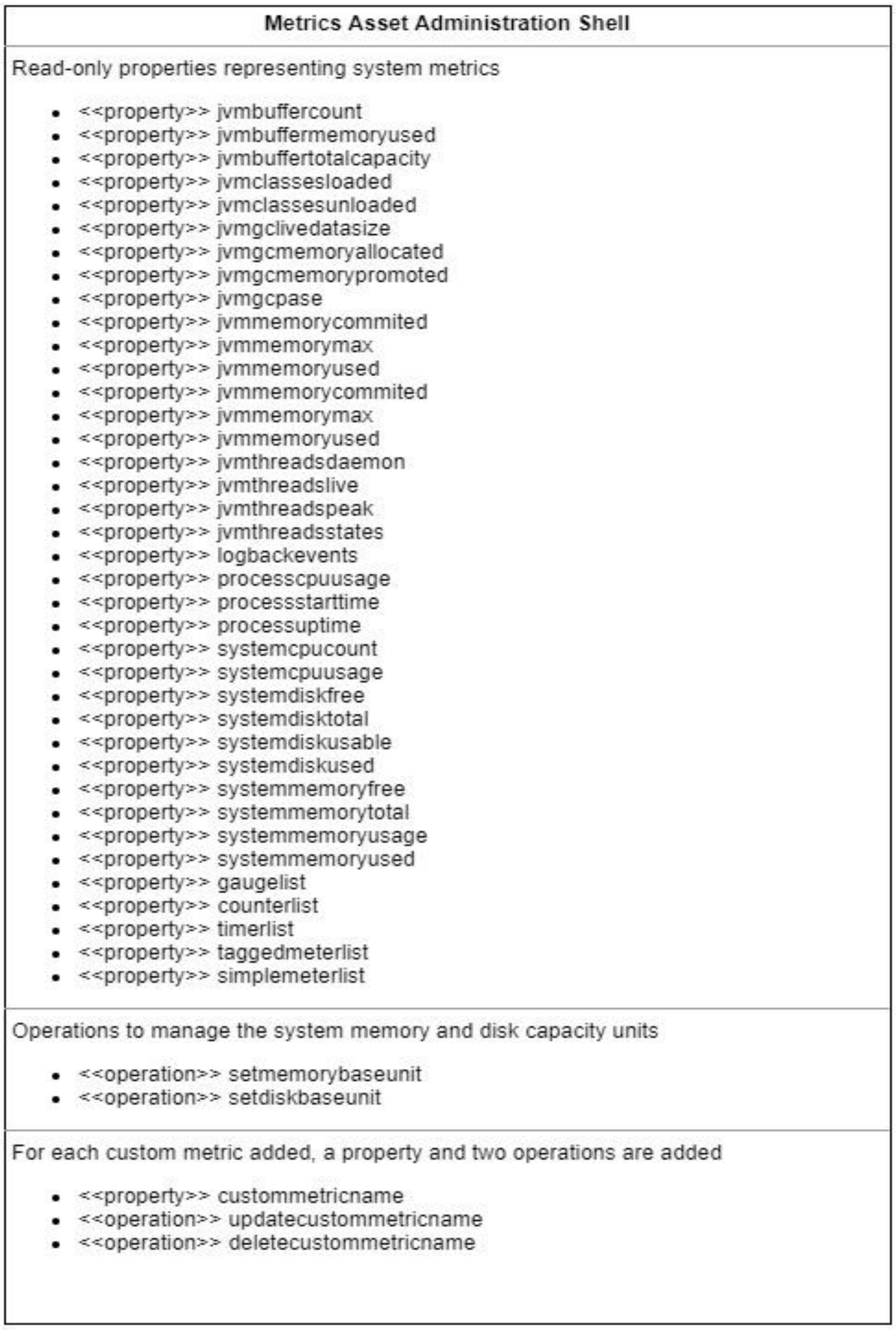
The second part of the *add metrics to bundle* method involves using a *Protocol Server Builder*. This is a builder created by the IIP-Ecosphere platform that allows us to create a protocol server that contains the required implementation for the properties and operations defined in the submodel. This element is required to add the operations that are called whenever the properties and operations mapped in the first part of the method are used. The functionality itself, as we already mentioned, lies within the RESTful client, but this method links the specific method to be used to each of the properties and operations created in the first part.

The final method located in this class allows the client-side application to define a custom meter. We can see two circumstances were this could be needed. The first one is that there is a custom meter created on the server-side that we want to monitor. That metric will have a URN identifying it. If we define a custom metric with the same URN and type as the metric created in the server-side, the RESTful client will retrieve the corresponding metric the same as the system metrics were retrieved. The second possibility is that we want to measure something on the client-side and store it in the *Metrics Provider* for future reference or so that other client-side applications interacting with the same *Metrics Provider* can access this values. Either case works the same on the client-side.

After specifying a valid name and type for the meter, a new property with that same name will be added to the AAS. Additionally, two operations will be added, one to update and one to delete. The naming convention used for this two operations is simply adding the words "update" and "delete" in front of the name supplied. Although the modification could be implemented as a setter for the property, we decided to keep all properties consistent as a read-only property and create the two operations instead. It is also a good idea taking into account that the JSON Object retrieved from the property and sent from the operations are structurally different as we previously described in the meter representation section.

Precisely due to how the different meters are updated using a different JSON Object structure, there are three different endpoints in the REST Service as we saw before. In order to help determine which endpoint we are trying to interact with, the *Meter Type* enumeration was created. This is a very simple enumeration containing only three primitives: *COUNTER*, *GAUGE* and *TIMER*. This way, the client-side can indicate what specific type of meter they are attempting to map under that name in a precise and safe manner, in turn, allowing the RESTful client to know which endpoint to connect to.

In order to aid in understanding the structure of the AAS, we present figure 13. In this small diagram, we have divided the contents of the AAS in three parts. The first one are properties mapping system metrics collected by the *Meter Registry* automatically as well as the system metrics extracted by the *Metrics Provider*, the second one are the operations used to change the system memory and disk unit, and finally a third section showing the property and two operations created when adding a custom meter.


Metrics Asset Administration Shell

Read-only properties representing system metrics

- <<property>> jvmbuffercount
- <<property>> jvmbuffermemoryused
- <<property>> jvmbuffertotalcapacity
- <<property>> jvmclassesloaded
- <<property>> jvmclassesunloaded
- <<property>> jvmgclivedatasize
- <<property>> jvmgcmemoryallocated
- <<property>> jvmgcmemorypromoted
- <<property>> jvmgcpase
- <<property>> jvmmemorycommited
- <<property>> jvmmemorymax
- <<property>> jvmmemoryused
- <<property>> jvmmemorycommited
- <<property>> jvmmemorymax
- <<property>> jvmmemoryused
- <<property>> jvmthreadsdaemon
- <<property>> jvmthreadslive
- <<property>> jvmthreadspeak
- <<property>> jvmthreadsstates
- <<property>> logbackevents
- <<property>> processcpuusage
- <<property>> processstarttime
- <<property>> processuptime
- <<property>> systemcpucount
- <<property>> systemcpuusage
- <<property>> systemdiskfree
- <<property>> systemdisktotal
- <<property>> systemdiskusable
- <<property>> systemdiskused
- <<property>> systemmemoryfree
- <<property>> systemmemorytotal
- <<property>> systemmemoryusage
- <<property>> systemmemoryused
- <<property>> gaugelist
- <<property>> counterlist
- <<property>> timerlist
- <<property>> taggedmeterlist
- <<property>> simplemeterlist

Operations to manage the system memory and disk capacity units

- <<operation>> setmemorybaseunit
- <<operation>> setdiskbaseunit

For each custom metric added, a property and two operations are added

- <<property>> custommetricname
- <<operation>> updatecustommetricname
- <<operation>> deletecustommetricname

Figure 13: Small diagram representing the AAS' structure

5.3.2.2 Implementation of the *Metrics Extractor REST Client*

This is the second most important class from this package. Just as we saw in the previous section, the *Metrics AAS Constructor* provides a "skeleton" of sorts for the AAS we intend to create to map the metrics. This acts as a facade that will be what the client-side application will be using. However, this "skeleton" is useless if the operations don't do anything. That is where the *Metrics Extractor REST Client* comes in.

The design used for this class follows the structure of a Jersey[EF18] REST Client. Unlike the *Metrics AAS Constructor*, this class does require instantiation. This is because, we need to build the URI of the server to be able to send the requests. All the possible endpoints are saved as constant values at the beginning of the class both to simplify the code below and to have an easy access to them in case they are modified in future updates, following the Java principle of "code once". We also included the different tags that can be expected from the tagged metrics for the same reason.

The class constructor requires a String representing the IP address of the resource we want to monitor, as well as an integer representing the port number. With these two parameters, the constructor will build a web resource containing the URI of the resource we are monitoring. This will be used to send all the different HTTP requests to the resource.

After the constructor, there are three methods that are used to send the different types of HTTP requests to the *Metrics Provider REST Service*. Due to the possibility of there being tags, the GET request is the most complex, but not excessively so. These three methods work similarly. We need to specify the endpoint we want to use as there are multiple possibilities, allowing us to have a single method to use with any endpoint. After the endpoint, the parameters are different depending on the request type. The *send GET request* method receives the name of the REST resource, in our case the name of the meter it is trying to retrieve, and a list of tags if it has any. The *send PUT request* receives the JSON Object with the information required to update the meter (as a String due to the aforementioned issues with JSON in the AAS environment) and the *send DELETE request* method receives the name of the REST resource we want to delete.

As we have seen in the previous paragraph, we have a method to send three of the four different types of requests that can be sent using HTTP. We can send GET, PUT and DELETE, but there is no method to send a POST request. This might seem strange as we already said that the client-side can create a custom meter, so it would make sense to use this request type to create it. The reasoning behind this is the way the *Metrics Provider* works in relation to the Spring Cloud Stream that is being used in the server-side. As we mentioned in the previous section, to reduce the operation overhead in the server-side's streaming application, the create and update operations for custom meters was unified into a single method. This incidentally creates a black-box of sorts in a way that there is no way for the machines to actually know whether the custom meter exists or not, requiring more checking if we decided to add the POST command, which could really only be used once with each meter, requiring more work to make than necessary.

For this reason, the following concept was introduced: instead of creating or updating an individual element of the map containing the custom metrics, we can consider this as updating the entire map object. This would, in theoretical terms, make using a PUT command instead of a POST command semantically valid in the eyes of REST, since we are indeed updating a resource, in spite of this meaning creating a smaller resource within. This might, at first, seem a bit dodgy, but we must remember that the same situation is happening in the exact same way in the server-side, so by maintaining a single update function that performs both the update and create operation, we continue respecting the operation consistency we had laid out.

After this three methods are introduced, we have a set of many different methods that use them to perform their operations. We could have made the three request operations public and used them for this as well, but it would imply a lot more coding as well as knowing the exact endpoints in the *Metrics AAS Constructor* class, so we decided it was best to have a single method per operation in the *Metrics Extractor REST Client*, which would not only simplify the work for the *Metrics AAS Constructor*, but also make it more readable.

The tagged metrics are, a bit special. The reason is that they are a collection of different meters of the same type (gauge, counter or timer) that are all grouped under a single name. The only thing that defines them and makes them different are the tags. Creating a single method for each of the different meters grouped together seemed a little bit too excessive, as the AAS would grow considerably. It also seemed strange to do so as, after all, in spite of being different meters, they are all referring to different values from the same resource. For example, if we take a buffer, there is a value for the direct memory and for the mapped memory, all from the same buffer. After some consideration, since we are working with JSON Objects regardless, it did seem like a plausible solution to create a JSON Array containing the different meters. After some thought and, in combination with what we had already created with the meter representations, we decided to create a JSON Array that would contain a series of JSON

Objects, one per meter extracted under that name. The JSON Objects would have two attributes, the first one being the tags used to extract that particular metric as it was what identified the metric to begin with. The second one is the meter object itself, which would consist of the specific JSON Object retrieved from the *Metrics Provider*. The JSON Object representing the retrieval of this tagged metrics can be found in the appendix.

We once again must note the limitation with JSON. In spite of being a separate class from the *Metrics AAS Constructor*, this methods will be embedded and used by the AAS, meaning that we cannot use the JSON functionalities here either for the time being. As a result, we once again find ourselves manually writing the JSON Objects and JSON Arrays ourselves where required.

As well as the JSON Object restrictions, there are also some methods that look interesting as we dive down, methods that receive and return Objects and that, when looking at the way they are implemented, we clearly see they actually only use a single argument of String type and return null. At first glance these methods seem ridiculous and out of place, so we found it necessary to provide some explanation to these peculiar methods. It is, once again, related to the AAS. The IIP-Ecosphere implementation of AAS supports adding operations dynamically, which is what is done here. Although we must define when building the project both the parameter type and the return type for them to be checked by the AAS's mechanisms , the signature of the method we have to introduce is still a generic one. This is related to the BaSyx restrictions that IIP-Ecosphere wraps. These signature defines that the method must receive an Object array as input (which will contain the appropriate parameters masked as Objects) and will return an Object (which will actually be the corresponding return type). In our case, nothing is returned on either of the methods, so we simply return "null".

The last methods that this class includes were added later on just to respect the REST standards a bit better as well as providing some extra functionality that could be interesting to have at some point. This functionality is to request a list of all the different metrics exposed on each of the endpoints. With this methods, we can know what custom meters of each type we have as well as knowing which of the system's metrics are exposed at each endpoint, allowing us to foresee if the returned JSON value is an object (meaning that it is a metric exposed on the *simple-meter* endpoint) or if we will be retrieving a JSON Array (which would, in turn, mean that its exposed on the *tagged-meter* endpoint).

5.3.2.3 Implementation of the support classes: *Metrics AAS Constants* and *Metrics AAS Construction Bundle*

After having a look at the two main classes from this package that are responsible for doing the work required to successfully build the AAS, we are going to have a look at these two support classes created in order to help both the *Metrics AAS Constructor* and the client-side application using these services.

The first class worth mentioning is the *Metrics AAS Constants*. This is, according to the Java notation, a class; but if we want to follow a more strict UML notation, this element would most likely be considered an enumeration. This is because, in spite of the Java notation that was used to avoid the implications that declaring the file as enumeration would have (which in java implies adding a series of methods that we are not interested in having for this particular enumeration), this class only contains constant String values. There are no attributes that define a state and there are no operations.

Initial prototypes used to test out the *Metrics Provider* component had all these values in the class that would act as an AAS creator, the equivalent to our *Metrics AAS Constructor*. At first, these values were part of that class. However, two facts soon became obvious. The first fact is that there are a lot of constant values which would significantly enlarge the size of the class, making it more obscure when it came to maintenance. The second fact, which was actually the one that fueled this decision, was that these values are not only used in the *Metrics AAS Constructor*. The constant values in this class are the URNs or names used by the AAS to identify the properties and operations that map the *Metrics Provider*'s meters and operations. This means that the *Metrics AAS Constructor* needs to have access to them to create the AAS, but the client application using the AAS also needs to have the exact same tags to be able to read the properties and invoke the operations. For this reason mainly, it was then decided to have a separate class containing only the values, avoiding the need to import the *Metrics AAS Constructor* class into the client using the AAS.

The constants saved here are, as previously mentioned, the names given to the different properties that mask the meters. There are tags for each of the system meters, including both simple meters and tagged meters. Additionally, we also added the prefix used for the update and delete operations for our custom meters. The way these operations are created is by concatenating the words "update" and "delete" to the custom meter's name. To showcase a simple example, if we create a custom counter called "mycounter", the property "mycounter" will be created mapping that meter, and we will also have the operations "updatemycounter" and "deletemycounter" added. The client using the AAS can use the tags in the same way if they want to avoid errors caused my misspelling a word.

One last type of constant value saved in this class are the names given to the attributes of the JSON Objects created when retrieving a tagged meter from the *Metrics Provider*. It is always a bit dodgy to hardcode the names of the attributes every time we want to retrieve them as there is always room to commit mistakes. By adding them to this class as constants, we can be sure we are safely retrieving the values.

In regards to the *Metrics AAS Construction Bundle*, this support class can easily be considered a "data type" in UML notation. The need for this class came as we were implementing the *Metrics AAS Constructor* class. We wanted to build a class that would allow the addition of system metrics as well as custom metrics to an AAS. In order to do this, we required a submodel builder and an invocables creator. Both of these elements come from the IIP-Ecosphere platform, the first one is a useful tool to build a submodel for the AAS and the second is to declare and define the how a property can be set or retrieved and what signature the operations have. After adding the operations, we would then return the sub-model builder. Now, if we want to add the implementation of the properties and operations introduced before, we would have to use the corresponding *Protocol Server Builder* and the appropriate instance of the *Metrics Extractor REST Client* with the *Metrics Provider REST Service*'s endpoint. The operations would be added and the *Protocol Server Builder* returned. If we wanted to stick to not using this data-type, we would first have to construct the submodel, then the protocol server. In between, there is room to make many mistakes like sending the wrong instance of the *Protocol Server Builder*, or sending the two at wrong times; which would result in runtime exceptions and errors. To avoid this, the best way was to do it all in one go. This was a problem because Java only allows us to return a single type using a function. The solution to this issue was clear: create a bundle that groups the submodel builder, the invocables creator, the protocol server builder and the RESTful client all in the same place and use the bundle as a whole for all operations.

This bundle is simply a data-type. It has a constructor that requests all four elements to be added. There is a setter and a getter for each of the four attributes and that is it. The way it is intended to be used is creating the bundle on the client-side application implementing the AAS server including all the elements needed into the bundle all at once. With this bundle created, all operations over any of the elements should be done using the getter operations of the bundle, to ensure that the components stay packed together. With this in mind, the *Metrics AAS Construction Bundle* is, not only a data-type that allows the AAS server application to keep all elements it needs for the AAS neatly together, but also a bundle used as in-out arguments for the *Metrics AAS Constructor* class, which works around the limitation of having a single return type in Java functions.

5.3.3 Dependencies of the client-side

We will finish this section the same way we finished the previous one: discussing the dependencies that the client-side has. Although some of the dependencies are shared among the client-side and the server-side, some are incompatible between each other. This was one of the reasons to separate the client-side and the server-side to begin with. The problem, in particular, lies within the version of the Tomcat[ASF21] server used by Spring (which is the one used in the server-side) and the one used by BaSyx (running in the client side). These incompatibilities are caused between the client-side and the server-side applications, and not by the component itself, so it is safe to work with the component on either as long as they don't mix.

Just like in the previous section regarding the server-side dependencies, we will present a diagram, seen in figure 10, with a graphic representation of the dependencies to see them at first glance.

We once again start from top to bottom to follow the diagram's content. We first encounter the meter representation package with the four classes it contains. This dependencies are easily explained as they all use the same dependencies and for the same reasons. All the classes use the *java.util* library to be able to create the dynamic lists required for saving the measurements required by the Micrometer-API interface *Meter*. The *Timer Representation* additionally needs this library to be able to define callables and suppliers, as well as having access to the *Time Unit* enumeration it uses for the base unit and conversions.

All of the classes from the meter representation package also use the *javax.json* library. As this classes run out of the AAS server's scope, particularly in the client using the AAS server's services, using JSON is possible here and, being the preferred option as continuously insisted upon in this document, there were no doubts on whether to use it or not.

To finish with the meter representation package, as one might expect, we require to use the *io.micrometer.core* library so we can have access to the different methods and operations we require from the Micrometer-API as well as being able to access the interfaces that we are attempting to implement with this classes. Thanks to this imports, not only can we freely use this classes as accurate representations of the interfaces they implement, but we can also use them as more than simple dummy representations. This is most visible with the *Timer Representation* class which, after all, has a method returning a *Histogram Snapshot*, which is a class from Micrometer-API, and that also uses the sample timers in Micrometer-API to record methods if needed.

Moving on to the next package, the metrics AAS package, we have surprisingly few dependencies. This is because the dependencies are already handled by the IIP-Ecosphere's library for the most part, meaning that all the parts involving AAS will simply use the IIP-Ecosphere Framework and avoid the dependencies. The only class that doesn't use these dependencies is the *Metrics Extractor REST Client*.

The *Metrics Extractor REST Client* uses the *javax.ws.rs.core* library to obtain a URI builder needed to create the web resource, to define the media type expected to be sent or returned through the HTTP request-response loop and to obtain the maps needed to correctly send the requests for the tagged meters. As well as that library, as we previously stated in one of the above subsections, this class is a RESTful Client implemented using a Jersey[EF18] REST client. For this reason, we include the *com.sun.jersey.api* library, which will provide the functionality we need to send the HTTP requests, get the responses and instantiate the web resource representing the REST service.

Last, but not least, we have the *Metrics AAS Constructor* and *Metrics AAS Construction Bundle*, both having a dependency with the *de.iip_ecosphere.platform.support.ass* library belonging to the IIP-Ecosphere framework. The bundle simply requires this to be able to store the instances

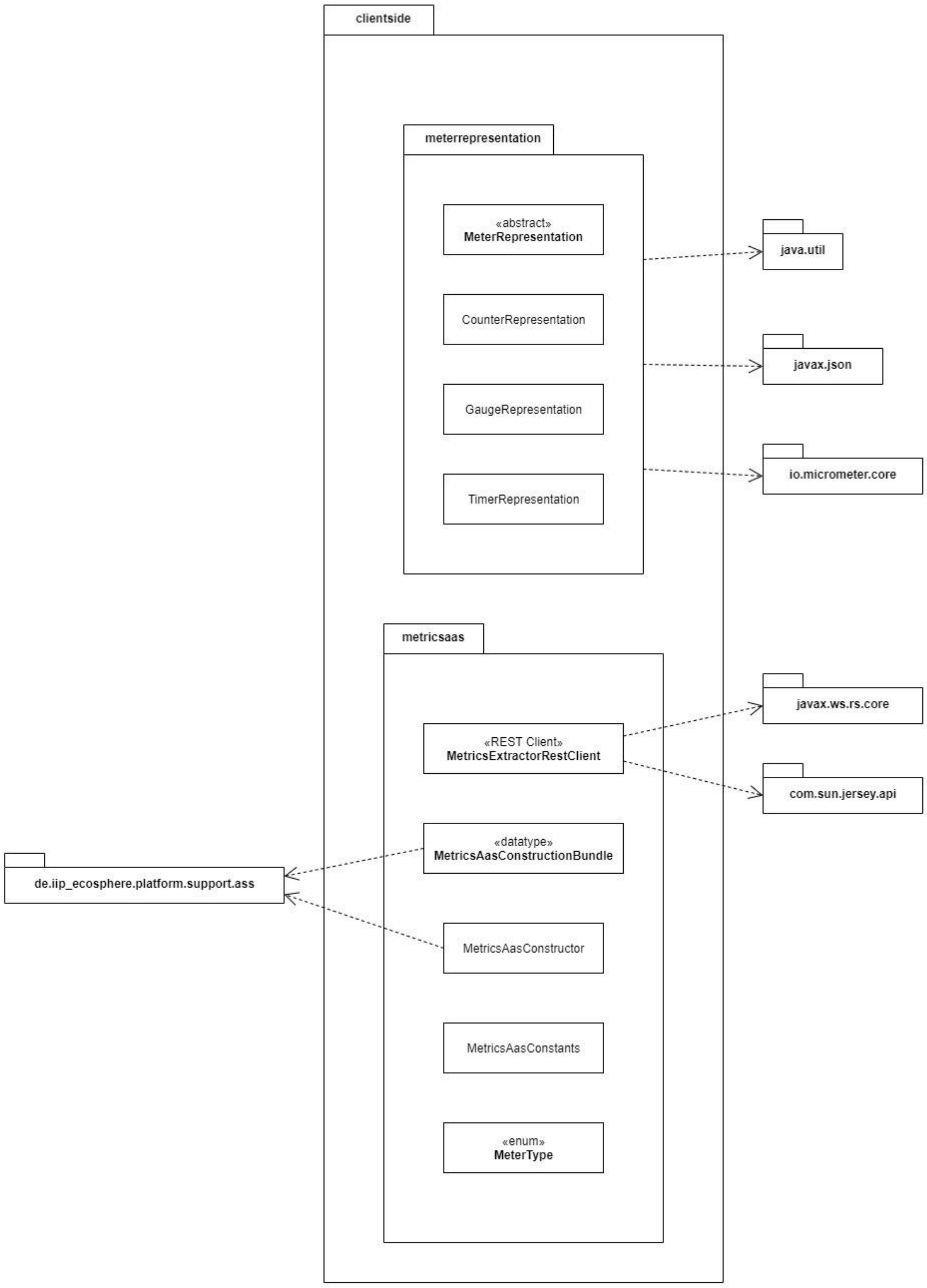


Figure 14: Dependency diagram of the client-side

of the submodel and protocol server builders and the invocables creator. The *Metrics AAS Constructor* needs this library to actually create the AAS's submodel and use the protocol server builder's capabilities to link the properties and operations from the submodel to the corresponding operations of the RESTful Client.

6 Evaluation and validation

In this section we discuss how the project's validity was evaluated and validated. Evaluation and validation are of great importance when it comes to any type of project, including software projects. An idea or a concept can be very appealing and very interesting, but if the product that was supplied as a result of the project is faulty, then it was a waste of time and resources to carry out the project to begin with. That is why, testing and evaluation has been something deeply embedded in this project from the very beginning. An early focus on testing and validation has allowed us to detect errors and bugs early in development, giving us plenty of time and space to solve them before they were even a problem.

Evaluation was also of great importance to understand how to proceed in the project's development. As we mentioned at the beginning of this document, many of the technologies used for this project are still in development and, as a result, there has been a lot of experimentation based on a trial-and-error system to achieve the goal we wanted to achieve before creating a stable version to add to the component. Many of the ideas that added new functionality have also arisen from the prototypes created to test the component. This would include ideas on how to do something more efficiently or how to make something easier and more uniform for a certain task. The entirety of the client-side part of the component was, in fact, somehow inspired by one of the prototypes used to test the client-side.

We have divided this section into three subsections, one for each type of evaluation and validation done for the project.

6.1 Funtional tests

This types of tests are the base for testing any good software project. This tests aim to assert that the functionality that a particular operation is said to have is, indeed there. This tests aim to ensure the correct result for an operation both in expected and unexpected conditions, to make sure that the operation also responds as intended in case of an error or invalid invocation.

Since the language of choice for the project is Java, we used the *JUnit* library to build unit tests. A unit test is a testing method in which individual units, in the case of Java, a method, are tested to determine whether they are fit for use or not[WP21]. Unit tests must include both valid and invalid uses of the specific method we are testing to know and ensure that they behave exactly the way we are expecting them to work.

Most of the elements created in this project, have a unit test attached to it that tests if the functionality is as expected. The classes involving the use of REST and HTTP were easier to test using instrumentation tests so, as a result, the *Metrics Provider REST* Service and all the classes involved in the metrics AAS package from the server side are tested using said instrumentation tests.

Unit tests follow a certain standard. Since the tests are supposed to be independent from execution order, each test case should be tested separately. This is because the unit test runner typically mixes up the order of the tests to ensure the order is irrelevant. Each method will, therefore have a test battery against each of the methods with both valid and invalid cases. We will now include the test battery for each of the classes and their operations. To better understand the notation used in the following tables, we provide a simple key in table 4.

Table 5: Unit Test battery key

Method	Case	Description	Expected result
Name of the method being tested	+1	Description of a valid use of the method	Expected result
	-1	Description of an invalid use of the method	Expected error response

To make it easier to distinguish each of the elements being tested, we will divide this section once again to reflect both the server-side and the client-side separately as well as further subsections for any special cases that have to be taken into consideration. Apart from the test, by using the tools provided by Eclipse IDE[MO20], we also run a coverage analysis to ensure that the different possibilities were correctly explored.

6.1.1 Server-side unit tests

Table 6: Capacity Base Unit Test Battery

Method	Case	Description	Expected result
bytevalue()	+1	Unit is BYTES	1.0
	+2	Unit is KILOBYTES	1024.0
	+3	Unit is MEGABYTES	1048576.0
	+4	Unit is GIGABYTES	1073741824.0
	+5	Unit is TERABYTES	1099511627776.0
stringValue()	+1	Unit is BYTES	bytes
	+2	Unit is KILOBYTES	kilobytes
	+3	Unit is MEGABYTES	megabytes
	+4	Unit is GIGABYTES	gigabytes
	+5	Unit is TERABYTES	terabytes
valueOf(String)	+1	Argument is "bytes"	BYTES
	+2	Argument is "kilobytes"	KILOBYTES
	+3	Argument is "megabytes"	MEGABYTES
	+4	Argument is "gigabytes"	GIGABYTES
	+5	Argument is "terabytes"	TERABYTES
	-1	Argument is null	Null Pointer Exception
	-2	Argument is a random String	*IllegalArgumentException*

Table 5 shows the test battery used to test the *Capacity Base Unit* enumeration. This enumeration is tested due to the fact that it has methods. The coverage analysis shows that this test covers 100% of the enumeration.

6.1.1.1 Metrics Provider Test

This specific test does require a bit more explanation as it is a mixture between a unit test and an instrumentation test. The reason for this is that this component is "alive" during its execution, and certain methods behave differently depending on the state of the component. Because of this, not all tests can be executed at a random moment. More in particular, we are talking about the tests regarding the modification and deletion of custom metrics. This methods behave differently depending on whether a requested metric exists or not to perform the various operations. The same request can, as a result, yield different results depending on the moment it was invoked. This dependency on the state has to be somehow managed. In order to do this, this methods were tested in one big block that executes the different operations through the different possible states the component can be in, ensuring its correct execution in each of the states. We will, as a result, include a table for the unit tests that do exist for this class as well as a table that acts as a step by step list of these instrumentation tests of sorts.

Table 7: Capacity Base Unit Test Battery

Method	Case	Description	Expected result
Initialization	+1	Valid Meter Registry	Metrics Provider instance created

	-1	Argument is NULL	*IllegalArgumentException*
setMemoryBaseUnit() & *getMemoryBaseUnit*()	+1	Change memory base unit to Kilobytes	*getMemoryBaseUnit*() will first retrieve a non NULL value and, after the update, retrieves KILOBYTES
	-1	Argument is NULL	*IllegalArgumentException*
setDiskBaseUnit() & *getDiskBaseUnit*()	+1	Change disk base unit to Megabytes	*getDiskBaseUnit*() will first retrieve a non NULL value, after the update, retrieves MEGABYTES
	-1	Argument is NULL	*IllegalArgumentException*
Gauge CRUD operations	1. The number of custom gauges is 0 2. Removing an inexistent gauge throws an *IllegalArgumentException* 3. Removing NULL throws an *IllegalArgumentException* 4. Adding a value to NULL throws an *IllegalArgumentException* 5. Adding a new gauge increases the number of custom gauges to 1 and the value retrieved is the same one we set it to 6. If we add a value to the same gauge, the number of gauges is still 1 and the value has changed to the new value 7. If we add a negative value to the gauge, the number of gauges is still 1 and the value has changed to the indicated value 8. If we request the value of an inexistent gauge, the value returned is 0 9. If we request the value of NULL, the value returned is 0 10. If we remove the custom gauge, the number of gauges is 0 11. Requesting the value of the gauge will return 0 12. Trying to remove the same gauge again throws an *IllegalArgumentException*		
Counter CRUD operations	1. The number of custom counters is 0 2. Removing an inexistent counter throws an *IllegalArgumentException* 3. Removing NULL throws an *IllegalArgumentException* 4. Increasing NULL throws an *IllegalArgumentException* 5. Increasing NULL by an amount throws an *IllegalArgumentException* 6. Increasing an inexistent counter increases the number of counters to 1 and the value of the counter is 1.0 7. Increasing an inexistent counter by an amount increases the number of counters to 2 and the value of the counter is the one we have indicated 8. Increasing the counter by an amount does not alter the number of counters (still 2), and the value of the counter is the sum of the previous value and the indicated value 9. Increasing a counter by a negative value throws an *IllegalArgumentException*, does not change the number of counters and does not alter the value of the counter 10. Requesting an inexistent counter value returns 0 11. Requesting the counter value of NULL returns 0		

	12. Removing a counter reduces the number of counters to 1 13. Removing the same counter again throws an *IllegalArgumentException*
Timer CRUD operations	1. The number of custom timers is 0 2. Removing an inexistent timer throws an *IllegalArgumentException* 3. Removing NULL throws an *IllegalArgumentException* 4. Recording a runnable with NULL throws an *IllegalArgumentException* 5. Recording a NULL runnable throws an *IllegalArgumentException* 6. Recording a Supplier with NULL throws an *IllegalArgumentException* 7. Recording a NULL supplier throws an *IllegalArgumentException* 8. Recording a one second runnable with an inexistent timer increases the number of timers to 1. The max time of the timer and the total time of the timer are 1.0. The usage count is 1. 9. Recording a three second runnable with that timer does not alter the number of timers (1). The max time of the timer is 3.0 and the total time 4.0. The usage count is 2. 10. Recording a two second runnable with that timer does not alter the number of timers (1). The max time is still 3.0, the total time is 6.0, the usage count is 3. 11. Recording a one second supplier with an inexistent timer increases the number of timers to 2. The max time of the timer and the total time of the timer are 1.0. The usage count is 1. 12. Recording a three second supplier with that timer does not alter the number of timers (2). The max time of the timer is 3.0 and the total time 4.0. The usage count is 2. 13. Recording a two second supplier with that timer does not alter the number of timers (2). The max time is still 3.0, the total time is 6.0, the usage count is 3. 14. Recording 1000 milliseconds with an inexistent timer increases the number of timers to 3. The max time of the timer and the total time of the timer are 1.0. The usage count is 1. 15. Recording 3000 milliseconds with that timer does not alter the number of timers (3). The max time of the timer is 3.0 and the total time 4.0. The usage count is 2. 16. Recording 2000 milliseconds with that timer does not alter the number of timers (3). The max time is still 3.0, the total time is 6.0, the usage count is 3. 17. Recording negative time with that timer throws an *IllegalArgumentException* 18. Recording a valid time with NULL as time unit throws an *IllegalArgumentException* 19. Requesting the total time from an inexistent timer returns 0.0 20. Requesting the total time of NULL returns 0.0 21. Requesting the max time of an inexistent timer returns 0.0 22. Requesting the max time of NULL returns 0.0 23. Requesting the usage count of an inexistent timer returns 0

	24. Requesting the usage count of NULL returns 0 25. Removing a timer decreases the number of timers to 2 26. Removing the same timer again throws an *IllegalArgumentException*

This tests cover 37.3% of the *Metrics Provider* class. This, under normal circumstances, is not a good coverage value, however, as we introduced at the beginning of this subsection, this class is a bit difficult to test using unit tests. Using the coverage tool, we determined that all the lines of code that were skipped by the unit tests were due to not having a Spring Boot Application, which is responsible for starting the schedule, which involves a big percentage of code, as well as all the code created to support the REST Service. This functionality is, of course, tested elsewhere, by using prototypes. The prototypes were also run using the coverage tool from Eclipse, so we could accurately ensure that the missed code was tested with the prototype.

6.1.2 Client-side unit tests

Table 8: Meter Representation test battery

Method	**Case**	**Description**	**Expected result**
Initialization with JSON	+1	Valid JSON and no tags	The Meter is created and the ID is correct (name, base unit, description and tags)
	+2	Valid JSON two tags	The Meter is created and the ID is correct
	+3	Valid JSON with no base unit value	The Meter is created and the ID is correct
	+4	Valid JSON with no description value	The Meter is created and the ID is correct
	-1	Argument is NULL	*IllegalArgumentException*
	-2	JSON is missing description attribute	*IllegalArgumentException*
	-3	Valid JSON but tag is missing the key	*IllegalArgumentException*
	-4	Valid JSON but tag is missing the value	*IllegalArgumentException*
	-5	Valid JSON but the tag is a random String	*IllegalArgumentException*
	-6	JSON does not have a name attribute	*IllegalArgumentException*
	-7	JSON does not have a name value	*IllegalArgumentException*
	-8	JSON does not have a base unit attribute	*IllegalArgumentException*
Initialization with name	+1	Valid name	The Meter is created and the ID is correct (name, base unit, description and tags)
	-1	Name is Null	*IllegalArgumentException*
	-2	Name is an empty String	*IllegalArgumentException*

Table 7 shows the test battery used to test the *Meter Representation* class. There is only one method that can really be tested in this class as most of the functionality is covered by the subclasses as this is an abstract class. Even though the tests are centered on a specific method (the constructor), the *get* ID operation is also tested as it is used to verify that the ID is correctly established by the constructor. The coverage analysis shows that this test covers 100% of this class.

Table 9: Counter Representation test battery

Method	**Case**	**Description**	**Expected result**
Initialization	+1	Valid JSON and no tags	The Counter is created. the ID is correct and the count value is the one in the JSON Object
	+2	Valid JSON two tags	The Counter is created. the ID is correct and the count value is the one in the JSON Object
	-1	JSON has two measurements	*IllegalArgumentException*
	-2	Measurement has the wrong statistic	*IllegalArgumentException*
increment(double)	+1	Incrementing the value by 1.0	The count value is the one in the JSON + 1.0
	+2	Incrementing the value by 0.0	The count value is the one in the JSON
	-1	Incrementing the value by -1.0	*IllegalArgumentException*
getUpdater()	+1	Increment the counter and request an updater	A JSON Object that follows the expected format and contains the value we incremented as "increment" value
measure()	+1	Request the measurements of the counter	There is a single measurement with the COUNT statistic and the statistic matches the count value of the counter
Initialization with name	+1	Valid name	The Counter is created. the ID is correct and the count value is 0, and the measurement is correct

Table 8 shows the test battery used to test the *Counter Representation* class. All the methods from this class are tested. The test for the initialization is more relaxed as some cases have already been covered by the *Meter Representation* test. The method returning the count value of the counter is also tested within the initialization tests as it is used to further ensure the initialization is correct. There are no invalid cases for said method, so no further testing of it is required. The coverage analysis shows that this test covers 100% of this class.

Table 10: Gauge Representation test battery

Method	**Case**	**Description**	**Expected result**
Initialization	+1	Valid JSON and no tags	The Gauge is created. the ID is correct and the value is the one in the JSON Object
	+2	Valid JSON two tags	The Gauge is created. the ID is correct and the value is the one in the JSON Object

	-1	JSON has two measurements	*IllegalArgumentException*
	-2	Measurement has the wrong statistic	*IllegalArgumentException*
getUpdater()	+1	Update the Gauge Value and request an updater	A JSON Object that follows the expected format and contains the value we indicated as "value" value
measure()	+1	Request the measurements of the gauge	There is a single measurement with the VALUE statistic and the statistic matches the value of the gauge
Initialization with name	+1	Valid name	The Gauge is created. the ID is correct and the value is 0, and the measurement is correct

Table 9 shows the test battery used to test the *Gauge Representation* class. All the methods from this class are tested. The test for the initialization is more relaxed as some cases have already been covered by the *Meter Representation* test. The method returning the value of the counter is also tested within the initialization tests as it is used to further ensure the initialization is correct. There are no invalid cases for said method, so no further testing of it is required. The same thing occurs with the *set value* method that we added to be able to update the gauge's value. This method is, in turn, tested in the *get updater* test and, having no invalid cases, no further testing of this method is required either. The coverage analysis shows that this test covers 100% of this class.

Table 11: Timer Representation test battery

Method	**Case**	**Description**	**Expected result**
Initialization	+1	Valid JSON and no tags	The Timer is created. the ID is correct and the counter, the total time and the max time are the ones in the JSON Object
	+2	Valid JSON two tags	The Timer is created. the ID is correct and the counter, the total time and the max time are the ones in the JSON Object
	-1	JSON has invalid base unit	*IllegalArgumentException*
	-2	JSON has two measurements	*IllegalArgumentException*
	-3	JSON has four measurements	*IllegalArgumentException*
	-4	One of the measurements has a VALUE statistic	*IllegalArgumentException*
	-5	There are two measurements with COUNT statistics	*IllegalArgumentException*
	-6	There are two measurements with TOTAL_TIME statistics	*IllegalArgumentException*

	-7	There are two measurements with MAX statistics	*IllegalArgumentException*
takeSnapshot()	+1	Request the histogram snapshot	The method does not return NULL
record(long, TimeUnit)	1. The initial count, total time and max time are 0 2. Recording 0 seconds increases the count by one but no other unit is altered 3. Recording 1 second increases the count by one, the total time and max time increases to 1.0 4. Recording -1 seconds increases the count by one, the total time is decreased to 0.0 and the max time remains being 1.0		
record(Supplier)	1. The initial count, total time and max time are 0 2. Recording a one second supplier increases the count to 1 and both the max time and total time recorded are 1.0 3. Recording a three second supplier increases the count to 2. The total time is now 4.0 and the max time is 3.0 4. Recording a two second supplier increases the count to 3. The total time is now 6.0, the max time is still 3.0		
recordCallable(Callable)	1. The initial count, total time and max time are 0 2. Recording a one second callable increases the count to 1 and both the max time and total time recorded are 1.0 3. Recording a three second callable increases the count to 2. The total time is now 4.0 and the max time is 3.0 4. Recording a two second callable increases the count to 3. The total time is now 6.0, the max time is still 3.0		
record(Runnable)	1. The initial count, total time and max time are 0 2. Recording a one second runnable increases the count to 1 and both the max time and total time recorded are 1.0 3. Recording a three second runnable increases the count to 2. The total time is now 4.0 and the max time is 3.0 4. Recording a two second runnable increases the count to 3. The total time is now 6.0, the max time is still 3.0		
totalTime(TimeUnit)	Requesting the total time in all possible time units renders the correct value in the corresponding unit		
max(TimeUnit)	Requesting the maximum time in all possible time units renders the correct value in the corresponding unit		
getUpdater()	+1	Record a long value, a supplier, a callable a runnable and request an updater	A JSON Object that follows the expected format and contains a JSON Array under the attribute "recordings" that contains the 4 recorded values in nanoseconds
measure()	+1	Request the measurements of the timer	There are three measurements. The COUNT measurement corresponds with the timer count, the TOTAL_TIME measurement corresponds to the total time recorded and the MAX measurement

			corresponds to the maximum time recorded
Initialization with name	+1	Valid name	The Timer is created. the ID is correct and the counter, the total time and the max time are zero, and the measurements are correct

Table 10 shows the test battery used to test the *Timer Representation* class. All the methods from this class are tested. The test for the initialization is more relaxed as some cases have already been covered by the *Meter Representation* test. The method returning the value of the count, max time, total time and base unit are also tested within the initialization tests as it is used to further ensure the initialization is correct. The total time and maximum times are also tested separately to ensure that the time unit is correctly converted from one unit to the other. The coverage analysis shows that this test covers 100% of this class.

Table 12: Metrics AAS Construction Bundle test battery

Method	**Case**	**Description**	**Expected result**
Initialization	+1	Everything is ok	The bundle is created and the attributes are the same ones used as arguments
	-1	The submodel builder is NULL	*IllegalArgumentException*
	-2	The Protocol Server Builder is NULL	*IllegalArgumentException*
	-3	The invocables creator is NULL	*IllegalArgumentException*
	-4	The Metrics Extractor REST Client is NULL	*IllegalArgumentException*
setSubmodelBuilder(SubmodelBuilder)	+1	Change the submodel builder	The submodel builder changes correctly
setInvocablesCreator(InvocablesCreator)	+1	Change the invocables creator	The invocables creator changes correctly
setClient(MetricsExtractorRestClient)	+1	Change the client	The client changes correctly
setProtocolBuilder(ProtocolServerBuilder)	+1	Change the protocol builder	The protocol builder changes correctly

Table 11 shows the test battery used to test the *Metrics AAS Construction Bundle* class. This is a simple data-type that only has setters and getters for each of the four attributes it contains. All the cases are covered with this test. The constructor internally calls the setters to unify the illegal argument check, so we no longer have to check it with the setter tests. Similarly, we use the getter operations to test the initialization and the setters, so we don't need separate test cases to check them. Coverage analysis of this class shows that 100% of the class is covered by this test.

Table 13: Metrics Extractor REST Client test battery

Method	**Case**	**Description**	**Expected result**
Initialization	+1	Arguments are valid	Client is created
	-1	Host is NULL	*IllegalArgumentException*
	-2	Host string is empty	*IllegalArgumentException*
	-3	Port number is negative	*IllegalArgumentException*

Table 11 shows the test battery used to test the *Metrics Extractor REST Client*. It can only cover the constructor as, similarly to the *Metrics Provider REST Service*, the functionality that this class provides is REST based and, as a result, requires to be tested using the integration tests. The only thing this unit test can check is if the initialization is correct. Because of this, the unit test only covers 3.5% of the class. It is an extremely low percentage of coverage, but, just like the main functionality can only be covered with the instrumentation tests, the best way to cover the constructor is with unit tests.

6.2 Integration tests

Once the functionality of the component that could be tested using unit tests was successfully verified, we could check how it worked in operation. This was an important test to carry out as some of the functionality, such as the *Metrics Provider REST Service*, as well as the elements in the client-side's metrics AAS package could not be tested any other way. This tests involve the creation of a prototype for each of the two parts the component has: one to use the server-side and one to use the client-side. Although this prototypes were simple, they were enough to correctly assert how the component worked under normal circumstances and allow us to cover the code we had not been able to check with the normal unit tests.

In this subsections, we will have a look at how each of the prototypes was implemented and how it works, as well as what functionality each of them is designed to test. The server-side prototype was, of course, the first one to be created and used, as the client-side needs a server-side to connect to and work. Once the server-side was yielding the expected results, this second prototype was built and run. The coverage analysis were not run until the both prototypes were working correctly. Thanks to this tests, as well as the coverage analysis, we were able to detect some errors and some missing elements that were added after the tests.

6.2.1 Server-side prototype

The server-side prototype is based on a testing prototype created in the IIP-Ecosphere that was used to test a simple Spring Stream Cloud service that uses a messaging broker. The original prototype contains three classes: *Test With Broker*, *Test* and *Configuration*, as well as some noteworthy resources such as a zipped broker server configuration and the *application.yml* file that Spring Cloud Stream uses. This prototype was executed by running the *Test With Broker* class. This class first extracts the broker configuration and adds some program arguments that are required by the Spring Cloud application. The prototype then deployed the broker and run the Spring Cloud application. The prototype originally created and deployed a MQTT[MO20] server, but some early testing we did revealed some compatibility issues between the server version and the Java version, so the server was changed to an AMQP[O14] server as MQTT had some stability issues on this version.

The original prototype's Spring Cloud Stream application consisted of a simple stream application with two beans: a supplier bean and a consumer bean. These beans worked together following a publish-subscribe style of messaging. One of the beans, the supplier, created a string of data (in this simple prototype, it was literally a String containing the word "DATA"). This would then be published to the broker server. The consumer bean would retrieve the data, following the destructive reading style of a subscriber and would process the data received (in this case, printing it to standard output). This class had a configuration class attached to it using the appropriate Spring Cloud Stream annotation, which allowed the Spring Cloud Stream Application to extract some configuration parameters from the *application.yml* file and use them as configuration options. By default, the Spring Cloud Application runs on an endless loop. To avoid this, the configuration included a limit of messages to be ingested by the consumer. When the number was reached, the application would safely shut down.

This working prototype was modified to suit our needs and allow us to test the functionality of our component in a semi-realistic manner, running the component similarly to how it would run in a real case scenario. This modifications included deleting the *Configuration* class, as we no longer needed it, and changing some code from the *Test* class, containing the Spring Cloud Application. The *Test With Broker* class, that starts up the broker server, is not modified in the slightest.

In order to work with the *Metrics Provider* component, we have to add the following annotations for the class, as well as maintaining the ones that might already be there. Said annotations are:

- **Component Scan**: this annotation tells the Spring Boot Application (which we can consider the Spring Cloud Stream Application's super-type) where the component is located. The argument it receives is a String with the name of the server-side package of the *Metrics Provider* component.
- **Enable Scheduling:** this annotation is required to enable scheduling operations in the Spring Boot Application. This will mean that it will create a separate thread to run scheduled operations, which we require to run the scheduled operation that extracts the system metrics.
- **Import**: this annotation allows us to indicate that we want to import the *Metrics Provider* class into the beans running in the Spring Cloud Stream Application, which will allow us to inject the dependency into the beans and access the custom metrics programmatically to modify them.

Apart from this annotations, we also modify the supplier and consumer so they don't depend on the *Configuration* file that was deleted. The supplier now instead creates a random number and uses this value to modify a custom counter, a custom gauge and records this entire operation using a custom timer. The first iteration it will also create some custom metrics to be used for the client-side testing prototype regarding REST modification, as it would be near impossible to modify constantly changing values. The random number is then published onto the broker. The consumer now retrieves that value and uses it to modify its own custom metrics while also timing the operation. The application will run for about 5 minutes before shutting down safely.

The final additions that have to be made to the server-side prototype are in the resources directory, in particular the *application.yml* file. Here, we can add the configurable properties for the *Metrics Provider* component by adding them under the tag *metricsprovider*. There is an example of what this file looks like in the appendix. Originally, there also had to be added a series of lines to ask the Spring Boot Actuator to expose the metrics, but this is now done by the REST Service, so it is no longer required. This exposes the REST Service in the default port 8080, but it can be modified using the appropriate property (also in the example). In our case, we will be using the 8080 port and will set the schedule to run every 3 seconds, use kilobytes as memory unit and megabytes as disk capacity unit.

6.2.2 Client-side prototype

The client-side prototype is a lot more complex than the server-side prototype as it is the one that actually carries out the monitoring, so it is responsible to actively send the requests. This prototype was also based on an example prototype from the IIP-Ecosphere that we looked at when researching about AAS. The prototype was originally intended for another student that was working on AI integration using AAS. It was a very crud and simple prototype in an early stage, but more than enough to understand how the IIP-Ecosphere implementation of AAS worked. Although it is based from an already existing prototype, the code was written from scratch, meaning that this was actually created by us and not copied and modified from the IIP-Ecosphere tests.

This prototype originally only tested the REST Service and the *Meter Representations*, meaning that it was responsible for creating both a RESTful client and an AAS supporting the monitoring apart from testing it. As the project progressed and it became clear that something needed to be done to standardize the AAS creation and data retrieval, a great chunk of what was originally part of this prototype was integrated into the component in the metrics AAS package of the client-side. The job regarding the creation of an AAS that supported monitoring as well as the RESTful client functionality became part of the component, simplifying the client-side prototype a lot. Due to this, if we look at the client-side prototype, we will see that all three classes that are a part of it have the suffix "V2" to signify they are a second version of the originally created client-side prototype.

The classes we want to focus on here are just two of the three classes: the *AAS Server (v2)* and the *AAS Jersey Client (v2)*. The third class, called *Time Recorder (v2)* is the one we used to carry out the performance tests, so it will be explained in the following subsection, allowing us to focus on the other two classes here.

The first class we will look at is the *AAS Server (v2)* class. This class is the first one that has to be run on the client-side prototype. Attempting to alter the order will, understandably, result in failure. As the name implies, this class is the one responsible for setting up and deploying an AAS Server. Without this AAS Server, we cannot have an AAS. This class starts by creating a couple of Server Addresses and endpoints for the AAS Server base and the AAS's registry. Once the endpoints are created, we build the AAS Builder using the *AAS Factory* provided by the IIP-Ecosphere framework. This AAS Builder will give us the submodel builder and the invocables creator we need for the component to work. The *AAS Factory* also provides the protocol server builder we need and we create an instance to the *Metrics Extractor REST Client*. We put all these four component into a *Metrics AAS Construction Bundle* and we are ready to use the component.

One of the aims we had with the component's signature was to allow the modification of an already created submodel. For this reason, to ensure it works correctly, we add a few properties to the submodel (a name, a version and a description) and provide their implementation in the protocol server builder prior to using the component. We now call the *add metrics to bundle* method from the *Metrics AAS Constructor* that adds all system metrics to the submodel. We proceed to add some custom metrics (corresponding to the ones created in the server-side for the consumer and supplier). This would be the way we are expecting the component to be used in a real scenario.

To check for errors, a few illegal operations are added surrounded by try-catch clauses to ensure that the errors are actually being thrown and caught correctly by the application. Once this is tested, we build the different elements and deploy the servers. The AAS Servers stay running for 2 minutes and then shut down safely. This was done in this manner so that both the normal test and the performance test could be launched.

The *AAS Jersey Client (v2)* class is very long, but in concept is quite simple. This class is responsible for running the actual test to ensure that the AAS and REST service as well as, of course, the RESTful client are running correctly. The class starts by retrieving the AAS from the *AAS Factory* the way an actual client of the AAS would do in operation. The submodel is retrieved and we proceed to extract the different data, displaying it on standard output. First we ensure that the data that was added by the *AAS Server* class was not modified by the component. After this has been checked, we retrieve the properties regarding the list of meters, including custom counters, gauges and timers, as well as both simple and tagged system metrics. After this, we retrieve each and every one of the properties, parsing them into the corresponding meter representation and printing them on the standard output using the parsed version instead of the JSON version for more clarity.

After all the properties are retrieved, we run some final tests that assert that the REST service works correctly. We retrieve, modify and delete some custom metrics that are not constantly changing in the server-side to assert that everything functions as intended. We also attempt to carry out illegal operations by sending invalid arguments to the REST service to test its reaction in this cases, and ensure that the *REST Advice* is correctly handling the errors.

Finally, this test modifies the original memory and disk capacity units using the REST Service before finishing and shutting down. In the upcoming section where we go over the test coverage, we will see how this somewhat simple tests allowed us to cover the functionality of the code in a very effective way.

6.3 Performance tests

Once we know that the component we are creating works correctly, another important question to ask ourselves is if the component itself is feasible. It may happen that the component works as expected in regards to produced results, but if the efficiency of the component is not good, we cannot really use the component, regardless of its successful execution of the task at hand. We already mentioned in the requirements section that there are a series of quality requirements in the IIP-Ecosphere that have to be taken into account. We have to bear in mind that the system we are monitoring could be critical for a certain production line, and that a failure we were not able to diagnose in time could be costly. For this reason, time efficiency is already something to take into account.

To analyze the performance of the system, there was an additional element added to the client-side, as the performance we are actually interested in is how long it takes to retrieve a value being monitored. This is what prompted us to create the aforementioned *Time Recorder (v2)* class. With this class we intend to throw a series of tests that would allow us to estimate the time required to retrieve the metrics and parse them into their corresponding meter representations. After running some initial tests, it became clear that the time required to parse the retrieved meter into an object is only a few microseconds, as it does not even reach one millisecond to even be detected by the *Time Recorder* application. When the second version was created, this was switched to nanoseconds to at least get the recording we could show on a graph.

After creating a few control variables, the application runs a loop a certain amount of times, which is the variable we will use for the "X" axis of the graphs. For each loop, all non tagged metrics (including the custom ones) are retrieved using the AAS. We decided not to retrieve the tagged ones as they would probably require more time to process as they are actually multiple meters being retrieved. This would alter the results, providing wrong measurements. Because of this, we only used those metrics that could be directly parsed into a meter representation. The time required to retrieve the meters is measured, then the same is done to calculate the parsing time. Both are combined to know the total time.

After all the loops have been done, the application calculates the mean value and prints it on standard output. This measurements were then collected and added to a spreadsheet, which we used to analyze the data and show the results. For each of the values for the loop, we repeated the execution three times for more accurate results. We recorded for 25, 50, 75, 100, 125, 150, 175 and 200 loops. The graphs from figures 11 to 18 show the resulting graphs from this study. We first present the retrieval times for the different meter types and then move over to the parse times.

To sum up what we explained in the previous paragraphs, the performance tests consists of an application that requests the system properties that don't have tags and parses them into the corresponding *Meter Representation*. Both the time taken to retrieve the meters and to parse them are recorded. All of this is done from the client side as the server-side already has to be running for it to work and the time it takes to set up depends on the Spring Boot Application rather than the component itself. Each meter type was recorded separately as they have differences from one another. Each loop consists on requesting all counters, all gauges and all timers, meaning that the full set of requests is performed rather than only requesting one type. The reason why it was done in this manner was because it is more similar to what would actually happen in a real case. In figure 15, we show a diagram explaining the test's flow.

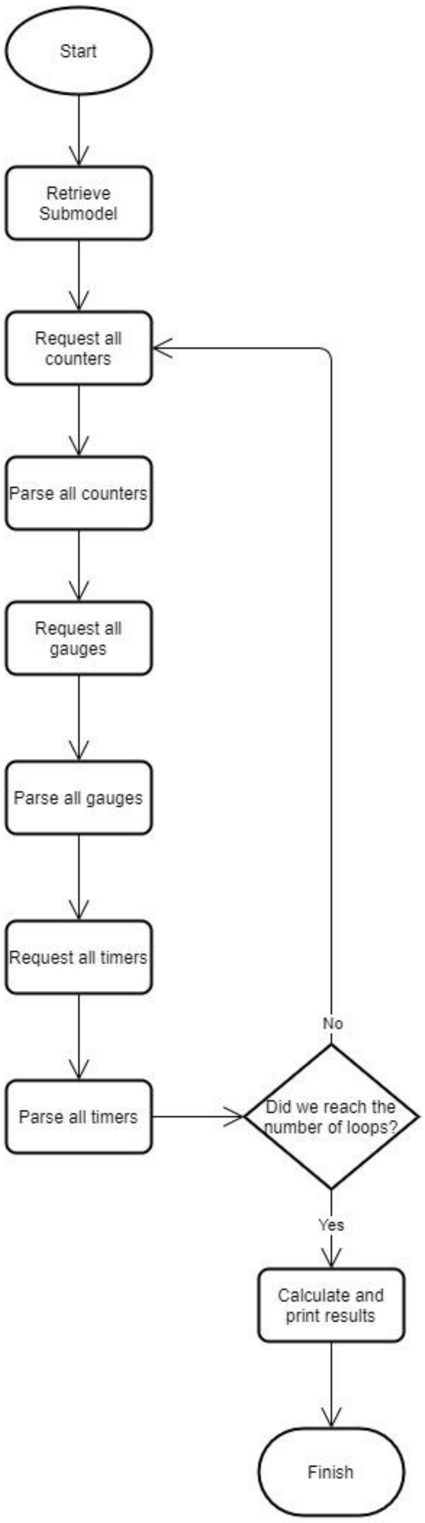


Figure 15: Flow diagram of the test

One final detail to be taken into account is that these tests were run on the same device, which affects the retrieval times for HTTP requests as the request doesn't need to be sent through a network, which would of course affect the time considerably. The device in question is a Lenovo Z50-70 laptop, running Windows 10 with an Intel-i7 1.80GHz processor and 16 RAM memory. The Java JRE version used was JavaSE-1.8 (jdk1.8.0_144).

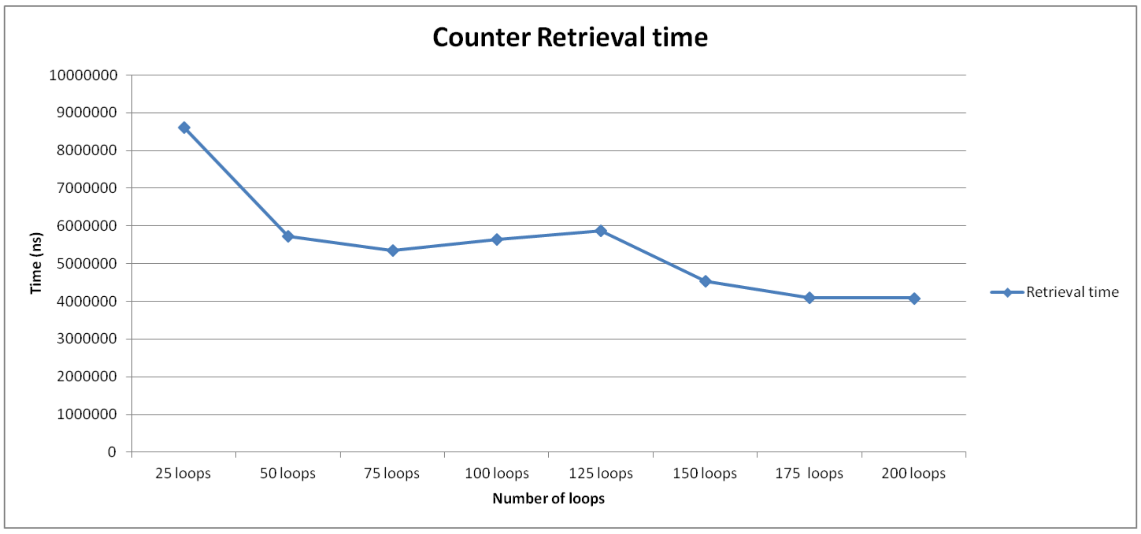


Figure 16: Graph depicting the time required to retrieve a counter in nanoseconds

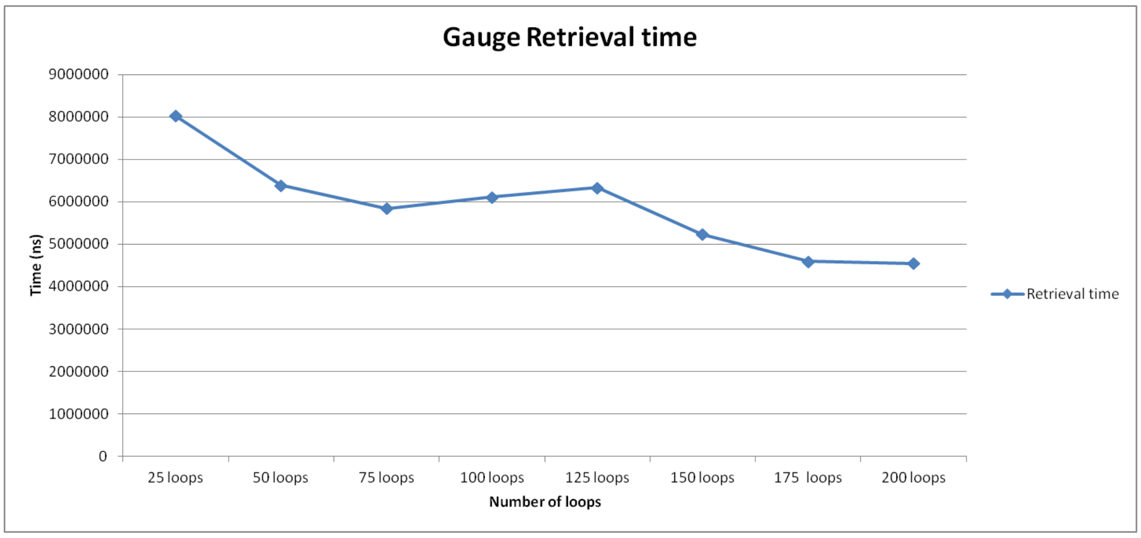


Figure 17: Graph depicting the time required to retrieve a gauge in nanoseconds

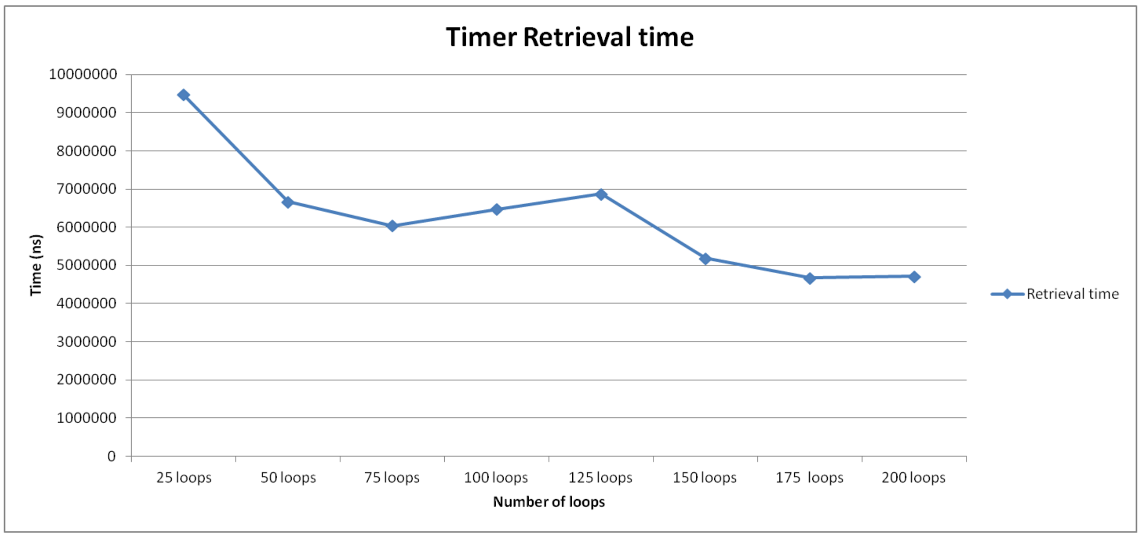


Figure 18: Graph depicting the time required to retrieve a timer in nanoseconds

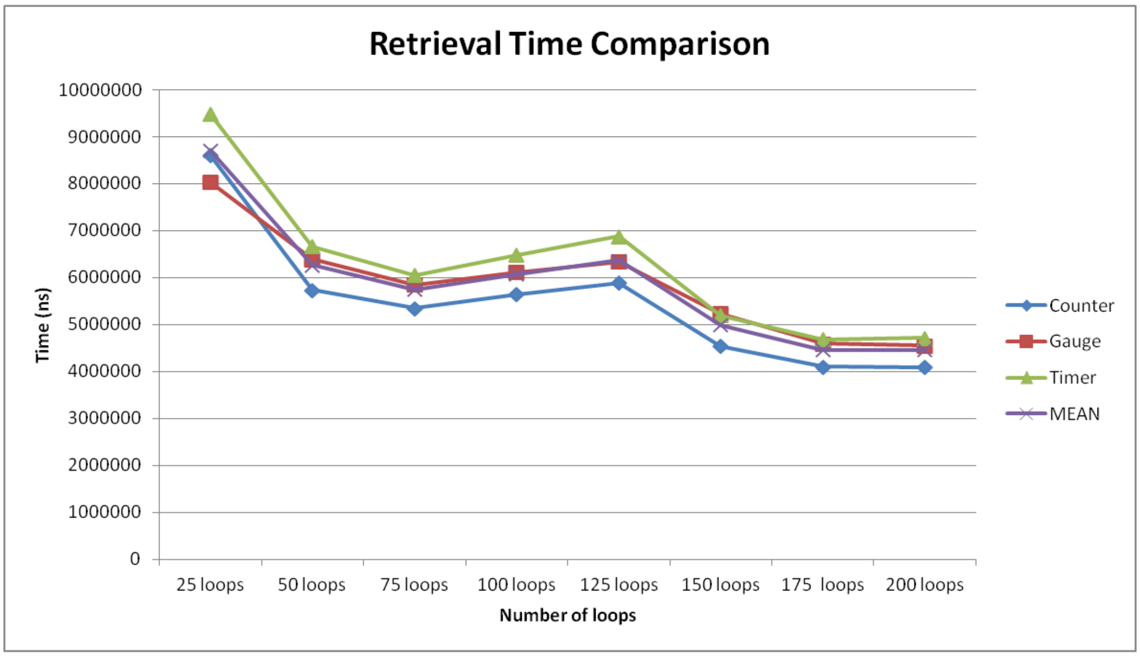


Figure 19: Graph depicting the time required to retrieve the different meters in nanoseconds to compare the retrieval times

The graphs from figures 16 to 19 show the different retrieval times for the different meter types. We can see that with few loops, the response times are quite elevated, ranging somewhere between 8ms and 9.5ms. However, this times go down as the number of loops increase, more or less staying stable between 4ms and 5ms when we reach 150 loops. Apart from this, we can see that all the different meters take, roughly, the same time to be retrieved, with no real significant difference between them. The timer takes a bit more than the rest due to size and the counter takes less for the same reason. The gauge, in spite of having the same size as a counter, takes longer because, as we already discussed, it calculates the value on request. The current state of the component makes it so that only a simple getter is used to calculate said value, thus keeping the times in harmony with the other meters. Prior to adding the scheduled operation we talked about in the *Metrics Provider*, the gauge typically took around 10ms to 11ms in the later loops, meaning it took about double the amount of time due to all the system calls that are required to retrieve the memory and disk capacity metrics. The results prove that the efficiency increase by sacrificing the real time in favor of semi-real time measurements for this set of metrics was worth the trouble. As a reminder, the rate can be changed to increase the frequency of the schedule, getting a more accurate reading, but the efficiency will slowly be affected.

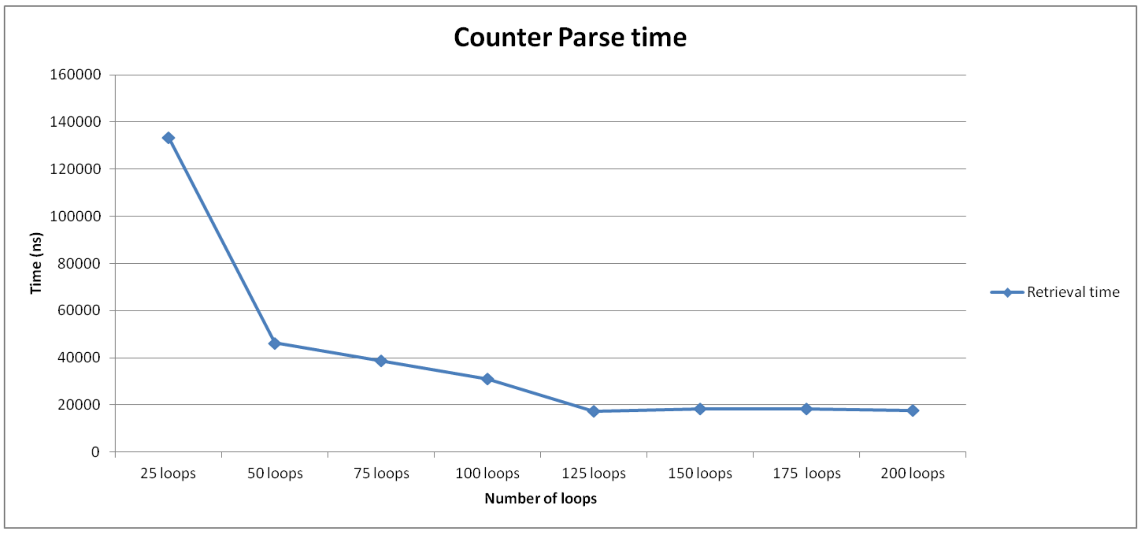


Figure 20: Graph depicting the time required to parse a counter in nanoseconds

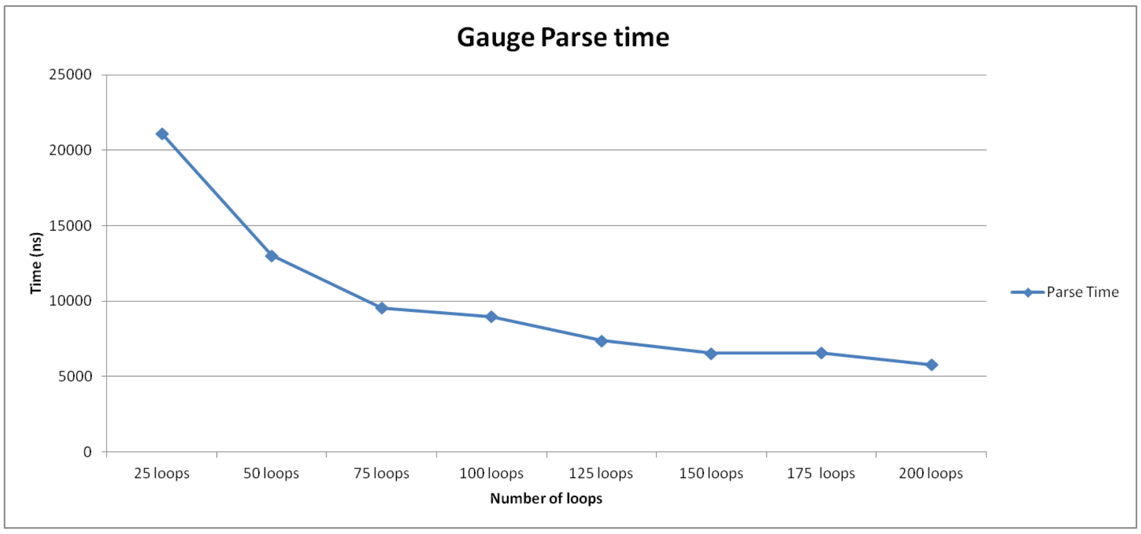


Figure 21: Graph depicting the time required to parse a gauge in nanoseconds

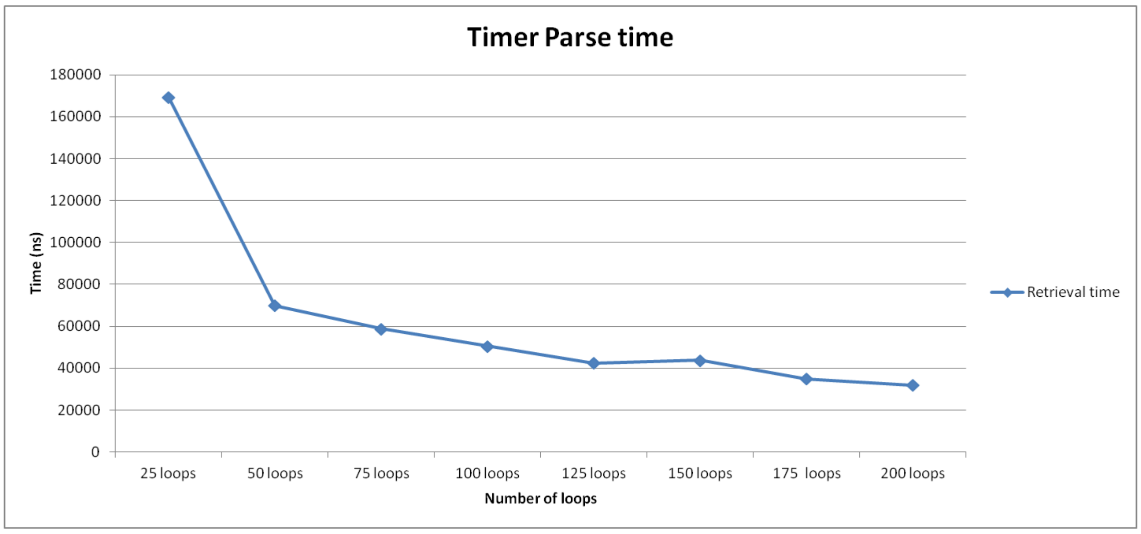


Figure 22: Graph depicting the time required to parse a timer in nanoseconds

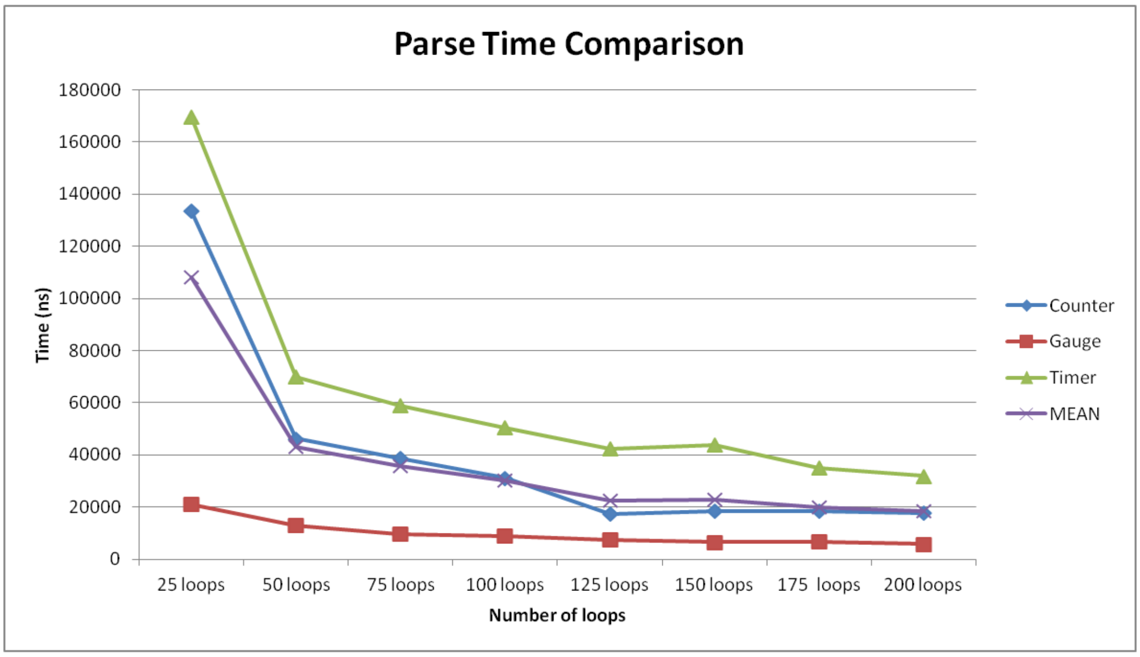


Figure 23: Graph depicting the time required to parse a timer in nanoseconds

The graphs in figures 20 to 23 show the times taken to parse the retrieved meter in String format to the corresponding *Meter Representation*. We can clearly see that the measurement from 25 loops is much higher, meaning that there is still a detectable overhead from the start up of the process. For this reason, we will ignore the results from the 25 loops, as they don't really show the times we want to focus on.

As we saw, due to the limitations we have, the meters have to be retrieved as Strings. This Strings are then parsed into JSON Objects and then parsed into a *Meter Representation*. The biggest factor for this times is, by far, the size of the meter. The bigger the meter, the longer it takes to parse it. Gauges are, when parsed into a *Gauge Representation* the simplest and smallest of all three meter types. We can clearly see this, as the gauges have never taken more than 20µs, staying at an almost steady rate of about 10 to 5µs to parse.

The counter, although equal in size in its JSON representation, does have more attributes that have to be taken care of when parsed into a *Counter Representation*. This makes it take significantly longer, taking somewhere in between 45µs and 20µs to be parsed.

The final meter is the Timer, which is the biggest of the three and, as a result, takes a lot more time to be parsed. The average time to parse a timer retrieved from the *Metrics Provider* to a *Timer Representation* is between 70µs to 30µs. Taking about 50% more than the other two meters combined. This is something we expected from the beginning as it was the most troublesome and large meter to create a representation of. Apart from that, we also know that it is the heaviest and most resource consuming *Meter Representation*, both size- and time-wise, so this was the most expected results.

Overall, the times taken for both the retrieval and the parsing are exactly what we had hypothesized and expected before even running the tests. After the modification introduced by the scheduled operation, all meters were supposed to be retrieved in the same time approximately, which we managed to achieve successfully. The parse times, based on sizes as we suspected, were also clear to constraint, having the gauge taking the less time and the timer taking the most.

With this results in mind, however, just as we said, the most critical process regarding time efficiency is clearly the retrieval time, which also makes sense as it involves HTTP communication, serialization and de-serialization of data and all the underlying communication protocols and dependencies. Even in this tests that, as previously described, ignore the time required to send the request through a network, the time taken to parse is insignificant to efficiency impact, as it doesn't even reach 1ms. As a matter of fact, it rarely surpasses 0.5ms, with only the timer getting close to reaching that mark. We will now finish this section by having a look at the test coverage.

6.4 Test coverage

Although we did mention the coverage of specific classes for some of the performed unit tests, as we saw with the *Metrics Provider* test, there is a lot of uncovered code that required the integration tests to run. If we run all of the unit tests created for the component, the total coverage of the code is 27.9% This is roughly a quarter of the component, showcasing that there is the need of running different tests to cover the rest. That is where the integration tests come in.

Luckily, we can run the integration tests with coverage as well, allowing us to obtain the statistic we require. Interestingly enough, although it theoretically should not be like this, running the server-side prototype with coverage measurements changed the way the garbage collector worked, causing it to have different tags and, as a result, causing an error on the client side. It was a good finding, as this error could easily be solved in the REST client by adding a few more checks to request the correct version of the Garbage Collector's timer, which was the meter in question that got altered by this change in execution. This, unfortunately meant that we had to change our execution pattern a little bit to correctly ensure that all the lines were being covered, as the coverage tool itself was what was causing this disruption.

In order to solve this issue, we run the prototypes with coverage in two turns. The first consisted on running both server- and client-side prototypes with coverage. After this execution, we rerun the project but only record the coverage of the client-side, so that the server-side behaves the way it is supposed to. After both executions have terminated, we merge their coverage to obtain the total coverage combining both executions. With this, we can accurately obtain the total coverage of the integration tests. This coverage was of 91.9% of the code, that included all that the unit tests could not cover.

To ensure that everything was correctly covered, using the same method as before, we merged together the coverage results of the unit tests and integration tests, what gives us a total coverage of 99.9%. This is not the ideal 100% that one might expect, but there are very clear explanations as to why this happens and that, in reality, the tests cover the functionality of the component without any problem.

The *Metrics AAS Constructor* considers itself 99.7% covered because, as the class is never instantiated for containing only classifier scope methods, it counts the implicit constructor method provided by Java as not covered. There is also a switch that considers that one of its branches is missed because the default branch is never covered. This default branch is impossible to cover as it implies a certain parameter to not coincide with any of the cases, which is impossible because the switch covers all the possible cases of the enumeration that that parameter belongs too. The last case could be set as default, but it wouldn't be very correct semantically speaking, so it is better to leave it as it is, by adding this explanation to justify why the tool considers this option "not covered", even though it was never an option to begin with.

Excluding this class, everything else is covered 100%, so we can be confident in saying that the component has been correctly tested, covering all possible branches in the code successfully.

7 Integrating the Metrics Provider Component

Now that we have discussed how the *Metrics Provider* component works, we will end by explaining how we can use the component's functionality within the IIP-Ecosphere's scope. The component can be treated like any other java component. A possibility would be to obtain it as a JAR file and import it as a library into the software project where we want to use it, however, with the existence of other tools, such as Maven[ASF02] or Gradle[G21], this is not recommended. The recommendation is to use a Maven Project, as it is both comfortable to manage all dependencies in the same way as well as keeping them up to date. Additionally, as we mentioned earlier, this project has been carried out using Maven, so we are more certain of its correctness if we use Maven.

As of now, we only know that this component will be a part of the *services.environment* library from the IIP-Ecosphere platform, but we still don't know the exact name that said library will have, so we will not be able to include it. In the examples found in the appendix regarding the POM files, due to not knowing the name, we called the artifact representing the component *de.iip-ecosphere.platform.monitoring.metricsprovider* in the current prototypes, and this will be the name shown in the appendix's examples. To import the component, we only need to include this in the POM file. In order for the component to work, however, there are a set of pre-requisites that are expected to be present in the project depending on what side of the monitoring is being carried out. We will now discuss what pre-requisites are expected on each of the two sides of in order for the component to work correctly.

7.1 Integrating the server-side of the component

As a reminder, the server-side of the component is the one responsible for collecting the resource's metrics and providing functionality to read them as well as allowing to create and manage custom metrics. This is the part of the component that will be running in the resource being monitored.

Despite this being the key part of the component, we do not require an excessive amount of additional dependencies for it to work correctly. Apart from the artifact representing the component, we will also have to include the Spring Boot Starter Web and Spring Boot Actuator libraries. These two components of the Spring Cloud Stream framework are the ones responsible for starting the Stream Cloud Application, that will boot up the different elements required for the monitoring and inject any dependencies they require, such as the *Meter Registry* where the meters will be registered, or importing the properties we might have added to the *application.yml* file. The Spring Boot Actuator is the one responsible for enabling the endpoints that will allow the REST service to work properly.

With this imported, we can create the Spring Stream Application we wish to use in our resource, adding the functionality we require for our particular task. We can proceed to create an reference to the *Metrics Provider* within the application's class if we want to add some custom probes on the server-side. This could be interesting if we want to time a particular operation, monitor a specific value with a gauge or count the number of times a particular element is called. The instance in question must not be manually initialized, instead, we will use the "Autowired" annotation so that the Spring Boot Application adds the correct instance through dependency injection. There is an example of this in the appendix, where the server-side prototype's main class is showcased with this elements and an example of usage in a simple stream.

In order for the component to be initialized and to be able to use it, however, we must add three additional annotations to the class. The first one is a "component scan" annotation with the full name of the package containing the *Metrics Provider* server-side, which will tell the Spring Framework where the component can be found for it to automatically start it and inject the dependencies it requires. Currently, this is:

```
de.iip_ecosphere.platform.services.environment.metricsprovider.serverside
```

We will also have to include the "enable scheduling" annotation for the system metrics that Micrometer-API does not automatically add to be added, and the "import" annotation with the *Metrics Provider* class as an argument to be able to import it into the application if we want to add custom probes. If no custom probes need to be added, this last annotation can be omitted.

As we mentioned earlier, we can use the *application.yml* file to add some properties as a blueprint. We can change the port the actuator uses to expose the REST service by adding the *server.port* property and we can change the configurable properties from the *Metrics Provider* in the same way by editing the values of *metricsprovider.schedulerrate*, *metricsprovider.memorybaseunit* and *metricsprovider.diskbaseunit*. All of this can be seen in an example of the *application.yml* file, corresponding to the file from the server-side's prototype used for testing, that can be found in the appendix.

7.2 Integrating the client-side of the component

The client-side of the component, as we previously stated, is the one responsible for allowing the metrics from the server-side to be collected so the monitoring can take place. This part of the component, as a result, is the one expected to be running on the resource that is carrying out the monitoring.

This part requires a few more dependencies for it to work optimally. Apart from importing the component, the Maven POM file will have to include all the libraries provided by the IIP-Ecosphere that are required to create an AAS. In particular, we are talking of the "services", "support AAS" and "support AAS BaSyx" libraries from the IIP-Ecosphere. This, just like with the Spring libraries in the server-side, is a requirement from the IIP-Ecosphere, so it doesn't really add anything new to use the core functionality that allows to retrieve the metrics from the *Metrics Provider*. We do, however, need to add a few other dependencies if we want to use the *Meter Representations*. We will, firs of all, require adding the "Javax JSON API" and the "Glassfish Javax JSON" libraries to be able to parse the String received through the AAS into an equivalent JSON Object we can use. Once we have the JSON Object, we will need to also have the dependency to the "IO Micrometer Core" library so we have access to the interfaces it provides for the different Meters, which will be what the *Meter Representations* will be using after parsing the JSON Object. Just like in the previous subsection, we will include the POM file used in the client-side's prototype as an example in the appendix.

Unlike the server-side, no additional annotations are required to be added for the client-side. The AAS is created the same way it had been created up till now. We only need to use the corresponding methods provided by the *Metrics AAS Constructor* class to add the metrics to the submodel and the implementation to the protocol server. The client of the AAS server that retrieves the metrics will use the corresponding classifier scope methods to parse the JSON Object into the corresponding *Meter Representation*. We will include in the appendix an example of usage for both the AAS Server constructor and the AAS client parsing the retrieved metrics.

8 Final remarks

In this section we conclude the project's document by adding some final words to answer the research questions we had introduced at the beginning of the document as well as allowing ourselves to take a step back and have a look at the project as a whole, trying to gather together the lessons that we learned during the development of this project.

8.1 Conclusion

This project has managed to create a component based on Spring Cloud Stream and Micrometer-API capable of monitoring the system resources of an OT resource. This component can actually be used, due to its implementation decisions, to create AAS in multiple IT resources to monitor the same OT resource, or multiple AAS in the same IT resource to monitor multiple OT resources. We decided, in spite of the constraints that some components had with certain data-types, to make our component as standardized as possible as well as allowing for as much flexibility as we could possibly add in order to stay in cannon with the rest of the IIP-Ecosphere project.

We have covered the project requirements we had initially established. We have indeed used Spring Cloud Stream and Micrometer-API to inject dependencies and manage and create probes by using a RESTful service. The created component can be added to the IIP-Ecosphere and, as a matter of fact, has actually been integrated within the project. The created RESTful service allows us to perform all the CRUD operations, and the requests sent to said service have successfully been masked by an AAS. The processing time has also proven to be within the project's guidelines and the project has been sufficiently documented, both in the code and in this document.

With the knowledge we have gathered from this project, we can also answer the research questions that guided us during the initial stages of this project.

Does Micrometer-API simplify architecture and/or implementation, and how can it be integrated with AAS on both IT infrastructure and OT resources?

Micrometer-API does simplify both architecture and implementation. The architecture is barely modified when using Micrometer-API as it acts as an addition to already existing layers and elements that the architecture already had, not really requiring to change much at all. It also simplifies implementation a lot. Many probes or meters are already automatically registered and administered under the hood by the *Meter Registry*. It also makes it easy to manage and use them as they can all be managed by using the *Meter Registry* instance, so the code required to add new probes, read already existing ones or modify probes can be done mostly in a single line for operation.

Regarding the second part of this question, due to the already implemented flexibility of the IIP-Ecosphere implementation of AAS, it was very easy to establish HTTP and REST as transport and application protocols to share the information extracted. All we needed is a component that started a REST server to allow us to access the metrics, and then have an AAS that uses a RESTful client to retrieve the values. The only requirement is that the AAS and the *Metrics Provider* are to not run in the same project scope, due to the incompatibilities between some server dependencies each of the two parts in the component has, but if in different project scopes, we can have an AAS running on either the IT infrastructure or the OT resource that would allow us to operate on the metrics in the exact same way.

Is AAS fast enough for IoT monitoring, taking latency into account?

To answer this question, we refer to the performance test we carried out where we recorded the time required to retrieve and parse the different meters. This question, although it may sound quite concrete, it is actually a bit ambiguous, as there is no clear definition as to what is considered to be "fast enough". As it tends to happen industrial environments, it may be fast enough for some cases, but not for others, so we need to elaborate the question a bit in order to provide an answer. We will, because of this, refer to IIP-Ecosphere's requirement R10, that establishes that the soft real-time processing for critical functions cannot surpass 100ms, as this is the only requirement we can use to establish a line between what we can consider "fast enough" and not. As the performance tests indicated, the time taken to retrieve and parse the meters never surpasses 10ms, in its worst case scenario, and with an average of 5.4ms. This means that the worst case scenario is below a tenth of the limit, and the average is actually roughly about a twentieth of this limit. With this in mind, we can consider that AAS is fast enough for IoT monitoring in term with the IIP-Ecosphere's current requirements.

How could custom probes be integrated?

This research question was one of the most interesting in our opinion. Monitoring the system's resources is always a go to in monitoring, but it is always extremely interesting to be able to monitor something other than system resources. Being able to add custom probes or meters is, more than a helpful possibility, a requirement at this point if we want a monitoring component to be considered good. For this precise reason we decided to add this as an actual goal to be achieved by the component, and we were able to easily do so with the help of Micrometer-API. Micrometer-API already includes operations and methods to construct all three types of meters it recognizes and add them to the *Meter Registry*. By encapsulating this functionality into the a single component represented by the *Metrics Provider*, we can easily add our own custom probes from the OT resource programmatically using the *Metrics Provider* class, or from both the OT resource and IT infrastructure using an AAS. All we need to do is specify the type of meter we need and give it a name, from there onwards, we just have to update the value.

To provide a more concrete answer to the question and to summarize the above paragraph, custom probes can be integrated by using the Micrometer-API functionality that the *Metrics Provider* component provides. This can either be done programmatically on the server-side using the *Metrics Provider* class, or on the client-side using the *Metrics AAS Constructor* class methods that were provided for that reason.

8.2 Outlook

One of the reasons why I decided to take this project was because it was a combination of elements that were familiar, such as the Java language and REST; and other elements and concepts that were completely new. This gave two elements that I consider vital for a correct motivation: a solid starting point from where to begin working, and something new and unknown that can provide some new knowledge. I also find it specially motivating if a software component or element created can actually be used in a real life scenario. Creating a useful software component is particularly satisfying. Every last bit of help that this component might represent for the IIP-Ecosphere project will be considered a win on my book.

Other than seeing the theoretical concept of IoT and monitoring at university lectures, I had never worked on anything related to either of them. This project, as a result, was a perfect opportunity to get started and learn a bit from both elements. Before this project, I had no knowledge what so ever of the existence of AAS. I had some knowledge about the concept of digital twins, but never in the sense of AAS.

I had never worked with Spring Cloud Stream and Micrometer-API either, so there was a lot to learn from these two elements. I was thoroughly impressed by the Spring Cloud Stream and the functionality this framework provides, so it will definitely be an interesting asset to consider in future work. These two frameworks work very well together from what I gathered in this project, so monitoring should be relatively easy with this two elements working in unison.

To finish off, we did learn some valuable lessons. The first one is that we should always test functionality as soon as possible. This was the philosophy used in this project, what helped us to detect issues early before they were too difficult to handle. We also learned that a bottom-up approach with an incremental style project works very well when delving into new territory. The bottom-up approach does give very good insight about how each element works so we can understand how to combine them correctly, even if there is the downside of having to step back from time to time to rethink the direction the project has to take. The incremental aspect allows us to set goals we know we can reach with our understanding. When we reach them, we can use the new knowledge we acquired to set a new goal, and so on.

A more particular lesson, this time regarding Micrometer-API, is that the Timers are usually the most inefficient meters, so gauges and counters should be used instead when possible. We also learned that gauges calculate the value upon request, which causes some performance issues if the operation is too complex. We solved that issue in the component by adding simple getters, but this is a valuable lesson for possible future work with Micrometer-API.

Another interesting detail regarding the Spring Cloud Stream scheduled operations, such as the one we use to record the system data, due to the "infinite loop" they produce, can be very resource consuming. Keeping a very low rate of execution could cause the efficiency of the entire system to be somewhat compromised. When testing this out, if we reduced the delay to 1ms, the time required to obtain the meters almost doubled or even tripled. That is why we kept 2 seconds by default. A triple digit number like 100ms shouldn't be an issue either, but we must really be careful if we start to go too far below that.

Appendix

A.1 Example of *application.yml* file

```
server:
    port: 8080
test:
    debug: false
metricsprovider:
    schedulerrate: 3000
    memorybaseunit: kilobytes
    diskbaseunit: megabytes
mqtt:
    port: 8883
    actionTimeout: 2000
    mqtt.qos: AT_MOST_ONCE
    host: localhost
    clientId: test
amqp:
    host: localhost
    port: 8883
    user: user
    password: pwd
logging:
    level:
        root: INFO
management:
    health:
        binders:
            enabled: true
    endpoints:
        web:
            exposure:
                include: "metrics"
spring:
    main:
        banner-mode: off
    cloud:
        function:
            definition: create;log
        stream:
            poller:
                fixedDelay: 8
                maxMessagesPerPoll: 1
            bindings:
                create-out-0:
                    destination: ingest
                log-in-0:
                    destination: ingest
            defaultBinder: amqpBinder

---

spring:
    config:
        activate:
            on-profile: source
    cloud:
        function:
            definition: create

---

spring:
    config:
        activate:
            on-profile: sink
    cloud:
        function:
            definition: log
```

The above example of an *application.yml* file has been extracted from the server-side prototype used for testing. The most relevant information is the one nested under the tag *metricsprovider* that sets the values for the configurable properties of the *Metrics Provider*. The only changes required to be added to the file are the ones nested under *server* and *metricsprovider*. The rest have nothing to do with the *Metrics Provider* component.

A.2 Example of a server-side Spring Cloud Stream Application

```
[...]
@SpringBootApplication
@ComponentScan("de.iip_ecosphere.platform.services.environment.metricsprovider.server
side")
@EnableScheduling
@Import(MetricsProvider.class)
public class Test {
    [...]
    @Autowired
    private MetricsProvider metrics;

    private boolean first = true;

    /**
     * Creates the data.<br>
     * Additionally, it will produce a number to be added to a custom gauge and time
     * the operation.
     *
     * @return the data generator
     */
    @Bean
    public Supplier<String> create() {
        return () -> {
            return metrics.recordWithTimer(SUPPLIER_TIMER_ID, () -> {
                double num = Math.random();
                metrics.addGaugeValue(SUPPLIER_GAUGE_ID, num);
                metrics.increaseCounter(SUPPLIER_COUNTER_ID);
                metrics.increaseCounterBy(SUPPLIER_COUNTER_ID, num);
                if(first) {
                    metrics.addGaugeValue(REST_GAUGE_ID, 0);
                    metrics.increaseCounterBy(REST_COUNTER_ID, 0);
                    metrics.recordWithTimer(REST_TIMER_ID, 0, TimeUnit.MILLISECONDS);
                    first = false;
                }
                return String.valueOf(num);
            });
        };
    }

    /**
     * Consumes the data received.
     *
     * @return the data consumer
     */
    @Bean
    public Consumer<String> log() {
        return data -> {
            metrics.recordWithTimer(CONSUMER_TIMER_ID, () -> {
                double num = Math.random();
                try {
                    Thread.sleep((long) num * 500);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                metrics.addGaugeValue(CONSUMER_GAUGE_ID, num);
                metrics.increaseCounter(CONSUMER_COUNTER_ID);
                metrics.increaseCounterBy(CONSUMER_COUNTER_ID, num);
                metrics.addGaugeValue(CONSUMER_RECV_ID, Double.valueOf(data));
            });
        };
    }

    /**
     * Main function.
     *
     * @param args command line arguments
     */
    public static void main(String[] args) {
        // start spring cloud app
        SpringApplication app = new SpringApplication(Test.class);
        app.run(args);
    }
}
```

Above is a fragment of the prototype used for the server-side. We have excluded the imports and some constant values used to name the custom metrics as they were irrelevant to showcase the use of the *Metrics Provider*.

A.3 Example of a client-side application constructing an AAS Server

```
public class AasServerV2 {
    [...]
    /**
     * Sets up and starts the AAS server.<br>
     * The AAS server and protocol server are set up using the Metrics Constructor
     * to help add the metrics, all of them mapped as properties. This process also
     * adds a few custom metrics. To be tested out.<br>
     * The server side must be running in order for this process to work, otherwise,
     * the AAS will crash when we attempt to use it.<br>
     * The AAS will automatically stop after running for 2 minutes.
     *
     * @param args command line arguments, never used
     * @throws Exception if the test fails
     */
    public static void main(String[] args) throws Exception {
        // Create the addresses and the endpoints
        ServerAddress aasServerAddress = new ServerAddress(Schema.HTTP, REGISTRY_PORT_NO);
        ServerAddress vabServerAddress = new ServerAddress(Schema.HTTP);
        Endpoint aasServerBase = new Endpoint(aasServerAddress, "");
        Endpoint registry = new Endpoint(aasServerAddress,
AasPartRegistry.DEFAULT_REGISTRY_ENDPOINT);
        System.out.println("Endpoints created");

        // Create the builders and put them in a bundle
        AasFactory factory = AasFactory.getInstance();
        AasBuilder aasBuilder = factory.createAasBuilder(AAS_NAME, AAS_URN);
        InvocablesCreator iCreator =
factory.createInvocablesCreator(AasFactory.DEFAULT_PROTOCOL,
                vabServerAddress.getHost(), vabServerAddress.getPort());
        SubmodelBuilder smBuilder = aasBuilder.createSubmodelBuilder(SM_NAME, SM_URN);
        ProtocolServerBuilder pBuilder = AasFactory.getInstance()
                .createProtocolServerBuilder(AasFactory.DEFAULT_PROTOCOL,
vabServerAddress.getPort());
        MetricsExtractorRestClient client = new MetricsExtractorRestClient(SERVICE_HOST,
SERVICE_PORT);

        MetricsAasConstructionBundle bundle = new MetricsAasConstructionBundle(smBuilder,
pBuilder, iCreator, client);
        System.out.println("Bundle created");

        // Create base properties
        smBuilder.createPropertyBuilder(PROP_NAME).setType(Type.STRING)
                .bind(iCreator.createGetter(PROP_NAME), InvocablesCreator.READ_ONLY).build();
        smBuilder.createPropertyBuilder(PROP_VERSION).setType(Type.STRING)
                .bind(iCreator.createGetter(PROP_VERSION),
InvocablesCreator.READ_ONLY).build();
        smBuilder.createPropertyBuilder(PROP_DESCRIPTION).setType(Type.STRING)
                .bind(iCreator.createGetter(PROP_DESCRIPTION),
InvocablesCreator.READ_ONLY).build();


        pBuilder.defineProperty(PROP_NAME, () -> "My service", null);
        pBuilder.defineProperty(PROP_VERSION, () -> "1.2.3", null);
        pBuilder.defineProperty(PROP_DESCRIPTION, () -> "Prototype 4 AAS", null);


        // We add the metrics and custom metrics to the submodel
        MetricsAasConstructor.addMetricsToBundle(bundle);
        MetricsAasConstructor.addCustomMetric(bundle, SUPPLIER_TIMER_ID, MeterType.TIMER);
        MetricsAasConstructor.addCustomMetric(bundle, SUPPLIER_GAUGE_ID, MeterType.GAUGE);
        MetricsAasConstructor.addCustomMetric(bundle, SUPPLIER_COUNTER_ID, MeterType.COUNTER);
        MetricsAasConstructor.addCustomMetric(bundle, CONSUMER_TIMER_ID, MeterType.TIMER);
        MetricsAasConstructor.addCustomMetric(bundle, CONSUMER_GAUGE_ID, MeterType.GAUGE);
        MetricsAasConstructor.addCustomMetric(bundle, CONSUMER_COUNTER_ID, MeterType.COUNTER);
        MetricsAasConstructor.addCustomMetric(bundle, CONSUMER_RECV_ID, MeterType.GAUGE);
        MetricsAasConstructor.addCustomMetric(bundle, REST_TIMER_ID, MeterType.TIMER);
        MetricsAasConstructor.addCustomMetric(bundle, REST_GAUGE_ID, MeterType.GAUGE);
        MetricsAasConstructor.addCustomMetric(bundle, REST_COUNTER_ID, MeterType.COUNTER);
        [...]
        System.out.println("Metrics added");


        // Now that all has been added, we build the submodel, the aas and deploy
        smBuilder.build();
        Aas aas = aasBuilder.build();
        Server aasServer = pBuilder.build();
        aasServer.start();
        System.out.println("AAS server started");


        Server httpServer =
factory.createDeploymentRecipe(aasServerBase).addInMemoryRegistry(registry.getEndpoint())
                .deploy(aas).createServer();
        httpServer.start();
        System.out.println("HTTP server started");


        System.out.println(
                "Registry can be found on address: " + aasServerAddress.toServerUri() +
registry.getEndpoint());
         [...]
    }


}
```

Imports, constants and some checks done to improve testing coverage have been removed as they were irrelevant to the construction of the AAS to monitor the *Metrics Provider*.

A.4 Example of a client-side application that uses an existing AAS to extract and operate with metrics

```
public class AasJerseyClientV2 {
[...]
    public static void main(String[] args) throws Exception {
        // Get endpoint reference
        Endpoint registry = new Endpoint(new ServerAddress(Schema.HTTP, 
AasServerV2.REGISTRY_PORT_NO),
                AasPartRegistry.DEFAULT_REGISTRY_ENDPOINT);

        // Retrieve AAS and Submodel
        AasFactory factory = AasFactory.getInstance();
        Aas aas = factory.obtainRegistry(registry).retrieveAas(AasServerV2.AAS_URN);
        Submodel sm = aas.getSubmodel(AasServerV2.SM_NAME);

        // TESTING THE GET FUNCTIONS
        System.out.println("Connection established successfully, retrieving data...");
        System.out.println();

        // Base Properties
        System.out.println("Base properties:");
        System.out.println(sm.getProperty(AasServerV2.PROP_NAME).getValue());
        System.out.println(sm.getProperty(AasServerV2.PROP_VERSION).getValue());
        System.out.println(sm.getProperty(AasServerV2.PROP_DESCRIPTION).getValue());
        System.out.println();

        // Lists
        JsonArray gaugesList = retrieveArray(sm.getProperty(GAUGE_LIST).getValue());
        JsonArray counterList = retrieveArray(sm.getProperty(COUNTER_LIST).getValue());
        JsonArray timerList = retrieveArray(sm.getProperty(TIMER_LIST).getValue());
        JsonArray taggedList = retrieveArray(sm.getProperty(TAGGED_METER_LIST).getValue());
        JsonArray simpleList = retrieveArray(sm.getProperty(SIMPLE_METER_LIST).getValue());

        System.out.println("Meter lists:");
        System.out.println("\tGauges list:");
        for (int i = 0; i < gaugesList.size(); i++) {
            System.out.println("\t\t" + gaugesList.getString(i));
        }
        [...]
        System.out.println();

        // Prepare auxiliary variables
        ArrayList<JsonObject> objects;
        ArrayList<JsonArray> arrays;
        JsonObject meter;
        String[] tags;

        // Counters
        ArrayList<Counter> counters = new ArrayList<Counter>();
        objects = new ArrayList<JsonObject>();
        arrays = new ArrayList<JsonArray>();


    objects.add(retrieveObject(sm.getProperty(AasServerV2.SUPPLIER_COUNTER_ID).getValue()));

    objects.add(retrieveObject(sm.getProperty(AasServerV2.CONSUMER_COUNTER_ID).getValue()));
        objects.add(retrieveObject(sm.getProperty(JVM_GC_MEMORY_ALLOCATED).getValue()));
        objects.add(retrieveObject(sm.getProperty(JVM_GC_MEMORY_PROMOTED).getValue()));
        objects.add(retrieveObject(sm.getProperty(JVM_CLASSES_UNLOADED).getValue()));
        arrays.add(retrieveArray(sm.getProperty(LOGBACK_EVENTS).getValue()));

        for (JsonObject jo : objects) {
            counters.add(CounterRepresentation.parseCounter(jo));
        }

        for (JsonArray ja : arrays) {
            for (int i = 0; i < ja.size(); i++) {
                tags = retrieveTags(ja.getJsonObject(i).getJsonArray(TAGS_ATTR));
                meter = ja.getJsonObject(i).getJsonObject(METER_ATTR);
                counters.add(CounterRepresentation.parseCounter(meter, tags));
            }
        }

        System.out.println("Counters:");
        for (Counter c : counters) {
            System.out.println(c.getId() + " >>> " + c.count());
        }
        System.out.println();

        // Gauges
        ArrayList<Gauge> gauges = new ArrayList<Gauge>();
        objects = new ArrayList<JsonObject>();
        arrays = new ArrayList<JsonArray>();

        objects.add(retrieveObject(sm.getProperty(AasServerV2.SUPPLIER_GAUGE_ID).getValue()));
        objects.add(retrieveObject(sm.getProperty(AasServerV2.CONSUMER_GAUGE_ID).getValue()));
        objects.add(retrieveObject(sm.getProperty(AasServerV2.CONSUMER_RECV_ID).getValue()));
        objects.add(retrieveObject(sm.getProperty(JVM_GC_LIVE_DATA_SIZE).getValue()));
        objects.add(retrieveObject(sm.getProperty(JVM_GC_MAX_DATA_SIZE).getValue()));
        objects.add(retrieveObject(sm.getProperty(JVM_THREADS_DAEMON).getValue()));
        objects.add(retrieveObject(sm.getProperty(JVM_THREADS_LIVE).getValue()));
        objects.add(retrieveObject(sm.getProperty(JVM_THREADS_PEAK).getValue()));
```

```
[...]
arrays.add(retrieveArray(sm.getProperty(JVM_MEMORY_MAX).getValue()));
arrays.add(retrieveArray(sm.getProperty(JVM_MEMORY_USED).getValue()));
arrays.add(retrieveArray(sm.getProperty(JVM_BUFFER_MEMORY_USED).getValue()));
arrays.add(retrieveArray(sm.getProperty(JVM_BUFFER_TOTAL_CAPACITY).getValue()));
arrays.add(retrieveArray(sm.getProperty(JVM_BUFFER_COUNT).getValue()));

for (JsonObject jo : objects) {
    gauges.add(GaugeRepresentation.parseGauge(jo));
}

for (JsonArray ja : arrays) {
    for (int i = 0; i < ja.size(); i++) {
        tags = retrieveTags(ja.getJsonObject(i).getJsonArray(TAGS_ATTR));
        meter = ja.getJsonObject(i).getJsonObject(METER_ATTR);
        gauges.add(GaugeRepresentation.parseGauge(meter, tags));
    }
}

System.out.println("Gauges:");
for (Gauge g : gauges) {
    System.out.println(g.getId() + " >>> " + g.value());
}
System.out.println();

// Timers
ArrayList<Timer> timers = new ArrayList<Timer>();
objects = new ArrayList<JsonObject>();
arrays = new ArrayList<JsonArray>();

objects.add(retrieveObject(sm.getProperty(AasServerV2.SUPPLIER_TIMER_ID).getValue()));
objects.add(retrieveObject(sm.getProperty(AasServerV2.CONSUMER_TIMER_ID).getValue()));
arrays.add(retrieveArray(sm.getProperty(JVM_GC_PAUSE).getValue()));

for (JsonObject jo : objects) {
    timers.add(TimerRepresentation.parseTimer(jo));
}

for (JsonArray ja : arrays) {
    for (int i = 0; i < ja.size(); i++) {
        tags = retrieveTags(ja.getJsonObject(i).getJsonArray(TAGS_ATTR));
        meter = ja.getJsonObject(i).getJsonObject(METER_ATTR);
        timers.add(TimerRepresentation.parseTimer(meter, tags));
    }
}
```

```
        System.out.println("Timers:");
        for (Timer t : timers) {
            System.out.println(t.getId() + " >>> CNT > " + t.count() + " || MAX > " +
t.max(t.baseTimeUnit())
                    + " || TOT > " + t.totalTime(t.baseTimeUnit()));
        }
        System.out.println();

        // TESTING THE REST FUNCTIONALITY

        // Retrieving the sample objects
        JsonObject gaugeObj =
retrieveObject(sm.getProperty(AasServerV2.REST_GAUGE_ID).getValue());
        JsonObject counterObj =
retrieveObject(sm.getProperty(AasServerV2.REST_COUNTER_ID).getValue());
        JsonObject timerObj =
retrieveObject(sm.getProperty(AasServerV2.REST_TIMER_ID).getValue());
        Gauge gauge = GaugeRepresentation.parseGauge(gaugeObj);
        Counter counter = CounterRepresentation.parseCounter(counterObj);
        Timer timer = TimerRepresentation.parseTimer(timerObj);

        // Printing the sample objects initial values
        System.out.println("Initial state of REST sample objects:");
        printRestSampleObjects(gauge, counter, timer);

        // Changing the values
        ((GaugeRepresentation) gauge).setValue(12.32);
        counter.increment(12.34);
        timer.record(1, TimeUnit.SECONDS);
        timer.record(3, TimeUnit.SECONDS);
        timer.record(2, TimeUnit.SECONDS);

        // Retrieving the updaters
        String gUpdater = ((GaugeRepresentation) gauge).getUpdater().toString();
        String cUpdater = ((CounterRepresentation) counter).getUpdater().toString();
        String tUpdater = ((TimerRepresentation) timer).getUpdater().toString();
        [...]
        // Requesting the update
        sm.getOperation(UPDATE + AasServerV2.REST_GAUGE_ID).invoke(gUpdater);
        sm.getOperation(UPDATE + AasServerV2.REST_COUNTER_ID).invoke(cUpdater);
        sm.getOperation(UPDATE + AasServerV2.REST_TIMER_ID).invoke(tUpdater);
        [...]

        // Requesting the deletion
        sm.getOperation(DELETE + AasServerV2.REST_GAUGE_ID).invoke();
        sm.getOperation(DELETE + AasServerV2.REST_COUNTER_ID).invoke();
        sm.getOperation(DELETE + AasServerV2.REST_TIMER_ID).invoke();
        [...]
        // Testing the configuration options
        [...]

        // Preparing the update info
        JsonObjectBuilder memoryJob = Json.createObjectBuilder().add("unit", "bytes");
        JsonObjectBuilder diskJob = Json.createObjectBuilder().add("unit", "kilobytes");

        // Requesting theupdate
        sm.getOperation(SET_MEMORY_BASE_UNIT).invoke(memoryJob.build().toString());
        sm.getOperation(SET_DISK_BASE_UNIT).invoke(diskJob.build().toString());
        [...]
    }
```

This class is very long because it corresponds to the client-side testing prototype, who was responsible of testing most of the functionality that the unit tests could not cover. To avoid adding excessive information, lots of the test cases that were irrelevant for what is intended to be shown here, as well as lines of code that repeated already showed operations have been removed to keep the file as clear as possible.

A.5 Examples of JSON objects representing meters and Meter updaters

A.5.1 JSON Object Representing a Timer

```
{
    "name": "custom.timer",
    "description": "A sample custom timer",
    "baseUnit": "seconds",
    "measurements": [
        {
            "statistic": "COUNT",
            "value": 3
        },
        {
            "statistic": "TOTAL_TIME",
            "value": 6.0
        },
        {
            "statistic": "MAX",
            "value": 3.0
        }
    ],
    "availableTags": [
    ]
}
```

A.5.2 JSON Object Representing a Gauge

```
{
    "name": "custom.gauge",
    "description": "A sample custom gauge",
    "baseUnit": "bytes",
    "measurements": [
        {
            "statistic": "VALUE",
            "value": 0.6245328097098749
        }
    ],
    "availableTags": [
    ]
}
```

A.5.3 JSON Object Representing a counter

```
{
    "name": "custom.counter",
    "description": "A sample custom counter",
    "baseUnit": "threads",
    "measurements": [
        {
            "statistic": "COUNT",
            "value": 139.64683506423114
        }
    ],
    "availableTags": [
    ]
}
```

A.5.4 JSON Object Representing a Timer Updater

```
{
    "name": "custom.timer",
    "recordings": [
        1000000000,
        3000000000,
        2000000000
    ]
}
```

A.5.5 JSON Object Representing a Counter Updater

```
{
    "name": "custom.counter",
    "increment": 5.1
}
```

A.5.6 JSON Object Representing a Gauge Updater

```
{
    "name": "custom.gauge",
    "value": 3.2
}
```

A.5.7 JSON Object used to update the system's disk or memory base unit

```
{
    "unit":"kilobytes"
}
```

A.5.8 JSON Object obtained when retrieving the *jvmbuffermemory* property of the AAS

```
[
    {
        "tags": [
            "id:direct"
        ],
        "meter": {
            "name": "jvm.buffer.memory.used",
            "description": "An estimate of the memory that the Java virtual machine is using
for this buffer pool",
            "baseUnit": "bytes",
            "measurements": [
                {
                    "statistic": "VALUE",
                    "value": 16338976
                }
            ],
            "availableTags": [
            ]
        }
    },
    {
        "tags": [
            "id:mapped"
        ],
        "meter": {
            "name": "jvm.buffer.memory.used",
            "description": "An estimate of the memory that the Java virtual machine is using
for this buffer pool",
            "baseUnit": "bytes",
            "measurements": [
                {
                    "statistic": "VALUE",
                    "value": 0.0
                }
            ],
            "availableTags": [
            ]
        }
    }
]
```

Glossary

Resource: umbrella term used to refer to both services and devices that compose or are part of the system.

Industry 4.0: term used to refer to the ongoing fourth industrial revolution. The main characteristic of this industrial revolution is the automation of industrial traditional industrial processes and practices[WP21].

The project/This project: term used to refer to the project that this document refers to.

Information technology: the science and activities that involve the use of computers and other electronic devices to store, send and manage information[CUP21].

Ubiquitous information system: a system that falls under the category of information system that is "seemingly everywhere". The idea behind ubiquitous information systems is the existence of electronic devices and systems around us to aid us in our day-to-day life while at the same time going unnoticed.

Edge device: a device that is located at the end of a network and that provides an entry point for information into said network[WP21].

Abstraction: a generic representation of a particular subject that can represent the subject without being based on a real situation[CUP21].

The cloud: a computer network that can be accessed via internet that allows the storage of data and programs as well as allowing the execution of certain applications.[WP21]

Cluster: group of computers connected together through a network that work together towards achieving the same goal[WP21].

System resources: term used to refer to the "tangible" resources that a computer system has. Some examples of this resources are the physical memory, the disk capacity or the CPU usage.

Prototype: an initial example of a product that can be shown and can later on be worked on to develop a more complete form of the product[CUP21].

Library: collection of non-volatile software resources that can be used by computer programs for software development[WP21].

Digital Twin: virtual representation that serves as a real-time digital counterpart of a physical object or process or, in this project's case, resource[WP21].

Stakeholder: a person or group of people that is involved with an organization, society, project, etc. and, therefore, has responsibilities and interests in it[CUP21].

Off-the-shelf: product that is available immediately and does not require to be specially made to fit a particular propose[CUP21].

Framework: a supporting structure around which something can be built[CUP21]. In the world of software engineering, this refers to a type of software that provides generic functionality that can be selectively changed by writing additional code[WP21].

Overhead: regular and necessary costs involved in operating a business or, in our case, a process[CUP21].

Java Script Object Notation (JSON): open standard file and data interchange format that uses human-readable text to store and transmit data objects consisting of attribute-value pairs and arrays[WP21].

Expensive Markup Language (XML): system of annotating a document in a way that is clearly distinguishable from the content that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable[WP21][MRC06].

Host: device or resource that contains a running process or service.

Port: a communication endpoint which consists of a logical construct that identifies a specific process or a type of network service, most commonly used in network protocols[WP21].

Redundancy: situation in which something is unnecessary because it is more than is needed[CUP21].

Element: term used in this document to refer to a small software component, usually part of a bigger component, and used to mark a difference between the smaller and the bigger component.

Java Class: in Java, a class is an individual component of code containing the properties and operations that a logical object created as an instance of that class contains. A class defines the attributes and behavior its instances will have[O21].

Java Object: particular instance of a Java Class. An object has the functionality defined in a class, but has an independent memory position as well as a particular state that will make it different to other objects from the same class. Said state is determined by the value of its attributes.

Attribute: a property corresponding to an object. Properties have a significant name to identify them as well as a value.

Package: in Java, a package is used as a namespace, a way to organize different classes in a way that is easy to determine the role played by a class as well as in what area it operates. Packages in Java also have an additional mechanism that modifies the visibility of certain elements within a class, such as its properties and operations in a way that some may only be visible for other classes within the package and others might be visible independent to the package[O21].

Enumeration: a data type consisting of a set of named values that behave as constant values in a language[WP21][O21].

Repository: in simple terms, it is described as a place where things are stored[CUP21]. In the world of software, a repository is a storage space for software projects used to create backups to an ongoing project or to store a finished software product that can be downloaded from there[WP21]. Many repositories also include a version control system to track progress and changes.

Server-side: this component is supposed to be run by two different and independent resources, one being monitored and one carrying out the monitoring. The server-side refers to the resource we are intending to monitor using the *Metrics provider* component.

Client-side: this component is supposed to be run by two different and independent resources, one being monitored and one carrying out the monitoring. The client-side refers to the resource carrying out the monitoring, that uses an AAS to represent the resource being monitored and, as a result, uses the corresponding part of the *Metrics Provider* component to do so.

Invocables creator: interface provided by the IIP-Ecosphere component used to programmatically add operations into an AAS during the program's runtime.

Serialization: process of turning a software element such as an object into a String of bytes, often in order to be sent through a network or to be stored in a database. This is often a requirement for database entities and objects intended to be sent via networks such as Java Beans or JSON Objects.

Java Beans: a Java software component that encapsulates multiple classes into a single entity that aims to create reusable software components for Java. For this reason, Java Beans have a zero-argument constructor, are serializable and have getters and setters for each of the different attributes it has[WP21].

Standard output: the standard output for an application is a directory to witch any output elements except errors will be sent. This can be set when running the application, like providing a file name, for example. If left untouched, the standard output will be the system's console terminal.

Super-type: in programming, there can be a hierarchical relationship established between different classes, where one class "inherits" some or all properties and operations from another class. This other class is what we call a super-class or super-type, if we decide to consider classes as ADTs (Abstract Data-Types). If we were to represent organic life as classes, an "animal" would be a super-type, as it refers to multiple different sub-types such as "humans" or "cats".

Sub-type: this is the opposite of the super-type. Following the hierarchy we described, the sub-class or sub-type would be the one inheriting the characteristics and properties from the super-type, and adding extra properties of its own, making it different to all other subtypes. Following the same analogy as before, "humans" would be a sub-type of "animals". This is because all humans are animals, however, a "human" is different from a "cat" which would be another sub-type from the "animal" super-type.

Port: at a software level, a port is a logical element that identifies a specific process or a type of network service.[WP21]

Try-catch clause: when executing code, a series of errors may occur. If an error occurs, the process' execution is halted, as it cannot continue. In order to define what to do in this situations, we can use a try-catch clause. The try statement allows us to define a block of code to be tested for errors in runtime. The catch statement allows us to define a block of code to be executed in case of a specific error occurring or, using Java's jargon, a specific exception is thrown. Try and catch statements have to work in pairs, thus defining the concept of try-catch clause.[RD21]

JAR file or jar file: JAR stands for Java Archive. A JAR file represents a format based on the popular ZIP file format used to aggregate multiple files unto a single one. A JAR file is commonly used to allocate all the different components of a Java Application in a single file that can be retrieved in a single HTTP transaction.[O04]

Method: in Java, a Method is an umbrella term used to refer to operations that an object might be able to execute. A method can be either a function, which returns some sort of result when it has finished its execution; or a procedure, which alters the state of the system in some way, but doesn't return a concrete result. Procedures in Java are marked as *void*, and functions will have a return type such as *integer, String*, etc.

Method scope: the method scope refers to the visibility of a particular method to certain software elements. In the sense used in this document, a method can have two different types of scope: a classifier scope or an instance scope. A classifier scope method is a stateless method visible to external classes. An instance scope method can only be executed by a specific instance belonging to that class and will either return a result or modify a value related to the state of that particular instance.

Bibliography

[IIPE19] IIP-Ecosphere. (2019). *IIP-Ecosphere*. 18.06.2021. https://www.iip-ecosphere.eu/

[PI418] Plattform Industrie 4.0. (2018). *Details of the Asset Administration Shell*. 18.06.2021. https://www.plattform-i40.de/PI40/Redaktion/EN/Downloads/Publikation/vws-in-detail-presentation.pdf?__blob=publicationFile&v=12

[WP21] Wikipedia. (2021). *Wikipedia: the Free Enciclopedia*. 18.06.2021. https://en.wikipedia.org/

[CUP21] Cambridge University Press. (2021). *Cambridge Dictionary*. 18.06.2021. https://dictionary.cambridge.org/

[EF21] Eclipse Foundation. (2021). *BaSyx*. 18.06.2021 https://www.eclipse.org/basyx

[IH21] IIP-Ecosphere, H. Eichelberger. (2021). *IIP-Ecosphere Platform Handbook. 0.20-SNAPSHOT.*

[VMM21] VM Malware. (2021). *Spring Cloud Stream*. 18.06.2021 https://spring.io/projects/spring-cloud-stream

[PS21] Pivotal Software. (2021). *Micrometer Application Monitoring. Vendor-neutral application metrics Facade*. 18.06.2021 https://micrometer.io/

[BFF96] Berners-Lee, T., Fielding, R., & Frystyk, H. (1996). *RFC1945: Hypertext Transfer Protocol--HTTP/1.0.*

[ASF02] Apache Software Foundation. (2002). *Apache Maven project*. 18.06.2021 https://maven.apache.org

[O21] Oracle. (2021). *Java Platform, Standard Edition 8 API Specification*. 18.06.2021. https://docs.oracle.com/javase/8/docs/api/

[MH20] M. Hoffmann. (2020). *Monitoring Spring Boot Application With Micrometer, Prometheus And Grafana Using Custom metrics*. 18.06.2021. https://www.mokkapps.de/blog/monitoring-spring-boot-application-with-micrometer-prometheus-and-grafana-using-custom-metrics/

[GL21] Grafana Labs. (2021). *Grafana*. 18.06.2021. https://grafana.com/

[SE21] Stack Echange Inc. (2021) *Stack overflow*. 18.06.2021. https://stackoverflow.com/

[O20] Oracle. (2020). *Interface OperatingSystemMXBean*. 18.06.2021. https://docs.oracle.com/javase/7/docs/api/java/lang/management/OperatingSystemMXBean.html

[EF18] Eclipse Foundation. (2018). *Eclipse Jersey*. 18.06.2021. https://eclipse-ee4j.github.io/jersey/

[ASF21] Apache Software Foundation. (2021). *Apache Tomcat*. 18.06.2021. https://tomcat.apache.org/

[MO20] MQTT Organization. (2020). *MQTT: The standard for IoT Messaging.* 18.06.2021. https://mqtt.org/

[O14] OASIS. (2014). *AMQP: Advanced Message Queuing Protocol.* 18.06.2021. https://www.amqp.org/

[RD21] Refsnes Data. (2021). *W3Schools.* 18.06.2021. https://www.w3schools.com/

[O04] Oracle. (2004). *JAR File Specification.* 18.06.2021. https://docs.oracle.com/javase/1.5.0/docs/guide/jar/jar.html

[G21]	Gradle Inc. (2021). *Gradle Build Tool.* 18.06.2021. https://gradle.org/
[WCY+18]	T. Wicksell, T. Cellucci, H. Yuan, A. Bross, N. Yap, D. Liu. (2018). *Netflix OOS and Spring Boot - Coming Full Circle*. 18.06.2021. https://netflixtechblog.com/netflix-oss-and-spring-boot-coming-full-circle-4855947713a0
[BD18]	J. Bursik, P. Dintakurthi. (2018). *Spring-Boot-Service-to-service Communication*. 18.06.2021. https://tech.target.com/2018/12/18/spring-feign.html
[MRC06]	McDonald, D., Rucker, T., & Coley, K. (2006). *XML (Extensive Markup Language).*