BEng Thesis

Arrowhead - Eventhandler System

José Sousa

CISTER-TR-161204

2016/11/11

Arrowhead - Eventhandler System

José Sousa

*CISTER Research Centre

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail: 1110852@isep.ipp.pt

http://www.cister.isep.ipp.pt

Abstract

The following document is the final report regarding PESTI 13 Internship Project of the Informatics Degree in Computer Science of ISEP.

The Eventhandler project is englobed in the European Project Arrowhead. This framework allows the development of collaborative applications between several devices also known as Internet of Things or IoT. Its base foundation lies on the SOA architecture and currently offers services such as: Service Discovery, Authentication, Orchestration, Authorization and others. Each of which will be described in this document.

Using the Eventhandler as an Arrowhead platform, Event Producer applications are able to register any kind of event such as: a temperature provided by sensor, warnings, errors, connection failures, etc. All this information must be stored in a database, local file or the Historian service provided by the Arrowhead Framework.

Event Consumers can subscribe to the Eventhandler using a filter. Therefore, being able to receive in real time incoming events or access them through the permanent storage referred above.

Besides implementing the Eventhandler locally, the main objective of this project is to also integrated this system within the Arrowhead cloud.

DEPARTAMENTO DE ENGENHARIA
INFORMÁTICA
Instituto Superior de Engenharia do Porto

Arrowhead - Eventhandler System

CISTER
Research Center in
Real-Time & Embedded
Computing Systems

2015 / 2016

111052 José Pedro Neto Ferreira de Almeida e Sousa

isep
Instituto Superior de
Engenharia do Porto

Arrowhead - Eventhandler System

CISTER
Research Center in
Real-Time & Embedded
Computing Systems

Degree in Informatics Engineer

October 2016

111052 José Pedro Neto Ferreira de Almeida e Sousa

Isep supervisor: Luis Miguel Moreira Lino Ferreira

CISTER supervisor: Michele Albano

DEPARTAMENTO DE ENGENHARIA
INFORMÁTICA
Instituto Superior de Engenharia do Porto

Acknowledgments

Despite all difficulties and distress encountered during the seven months working in CISTER I would like to sincerely express my gratitude to all that always supported me and never lost their faith in me.

I would like to give a special thanks to Luis Lino Ferreira, José Bruno and Michele Albano whom always gave me all the necessary support and guidelines in order to complete this project.

Also a very warm gratitude goes towards my family and girlfriend who were always there for me when I needed.

Finally, I would like to thank DEI-ISEP to provide the possibility to enroll in a project of international magnitude and allowed me to start my career in the labor market.

Summary

The following document is the final report regarding PESTI – Internship Project of the Informatics Degree in Computer Science of ISEP.

The *Eventhandler* project is englobed in the European Project Arrowhead. This framework allows the development of collaborative applications between several devices also known as Internet of Things or IoT. Its base foundation lies on the SOA architecture and currently offers services such as: Service Discovery, Authentication, Orchestration, Authorization and others. Each of which will be described in this document.

Using the *Eventhandler* as an Arrowhead platform, Event Producer applications are able to register any kind of event such as: a temperature provided by sensor, warnings, errors, connection failures, etc. All this information must be stored in a database, local file or the Historian service provided by the Arrowhead Framework.

Event Consumers can subscribe to the *Eventhandler* using a filter. Therefore, being able to receive in real time incoming events or access them through the permanent storage referred above.

Besides implementing the *Eventhandler* locally, the main objective of this project is to also integrated this system within the Arrowhead cloud.

KEYWORDS (SUBJECT): PESTI, ISEP, Arrowhead, Eventhandler, Internet of Things

KEYWORDS (TECHNOLOGY): SOA (Service-Oriented Architecture)

Table of Contents

Figures Index

Tables Index

Notation and Glossary

AAA	Authentication, Authorization and Accounting
ACS	Arrowhead Core Services
Apache	Multiplatform HTTP server
CP	Communication Profile
DB	Database
Framework	Development tool
GIT	Distributed Version Control
GUI	Graphical user interface
HTTP	HyperText Transfer Protocol
IDD	Interface Design Description
IDE	Integrated Development Environment
Jetty	HTTP server and Servlet container
Jersey	RESTful Web Services framework
JSON	JavaScript Object Notation
Maven	Project Management Tool
MySQL	SQL database system
REST	Representational State Transfer
SD	System Description
SQL	Structured Query Language
SOA	Service Oriented Architecture
SoSD	System of Systems Description
SoSDD	System of Systems Design Description
SP	Sematic Profile
SVN	Arrowhead Common Design Repository
URI	Uniform Resource Identifier
XML	eXtensible Markup Language
WADL	Web Application Description Language

1 Introduction

This chapter is dedicated to the introduction of this project by describing in what context it is inserted. A brief presentation of the Research Centre is made mainly because CISTER is the hosting institution for this project and it is also directly connected to ISEP. Being a collaborative project, everyone that was directly involved with my work will be mentioned. Lastly the structure of this document is described.

1.1 Context

The development of this project was carried out under umbrella of the discipline PESTI – Project/Internship of the School of Engineering (ISEP) of the Polytechnic Institute of Porto (IPP).

The main purpose of this course is to give students the opportunity to be enrolled in a project inside a company in order to develop and enhance their skills and knowledge and provide some of the experience requested by the ever demanding labor market.

This internship was held in CISTER and its main purpose is to develop an event handling application to be later integrated with an already existent framework (Arrowhead). Some of the main areas related to this project are: IoT (Internet of Things), Distributed Systems and Service Oriented Architectures (SOA).

1.2 Project Presentation

1.2.1 Arrowhead Framework

The Arrowhead Framework objective is to support IoT-based automation applications. The creation of this automation is based on the idea of local automation clouds. A local Arrowhead Framework cloud can be compared to a global cloud while providing improvements and guarantees regarding:

- Real time data handling
- Data and system security
- Automation system engineering

- Scalability of automation systems

The Arrowhead Framework is built on three the fundamental principles of SOA, lookup, loosely coupled and late binding. A service consuming system has little or even no knowledge of other systems providing the services it is interested in consuming. Systems can be deployed in networks without initial bindings to other systems, where Service bindings (establishing a service instance provision-consumption binding) can be established, broken up or changed in runtime.

The purpose is to enable the application systems in an easy and flexible way to be able to collaborate successfully.

The Arrowhead framework concept is based on three functional areas, called core functionalities or core systems.

The Arrowhead Framework offers the core services functionalities through the definition of three groups, Information Assurance services (IA), Information Infrastructure services (II) and System Management services (SM). It defines three mandatory systems, one belonging to each one of the three groups: Service Discovery (SD), Authorization and Authentication (AA) and Orchestration (O), to provide the services mentioned above. The SD system is used to allow service consumers to find the address of registered service producers. The AA system is used to authenticate and provide authorization for connections between services. The O system is used to determine the service producers that match specific criteria, e.g. choosing between services producers serving in the same geographical area where the service consumers are located. It is also responsible for the negotiation of QoS and keeping track of the system configuration. An example of its usage pertaining to home automation scenarios involves determining which services are capable of providing temperature readings in a house and to dynamically connect systems that need such kind of services with the most adequate providers of the service (e.g. according to the sensitivity of the readings).

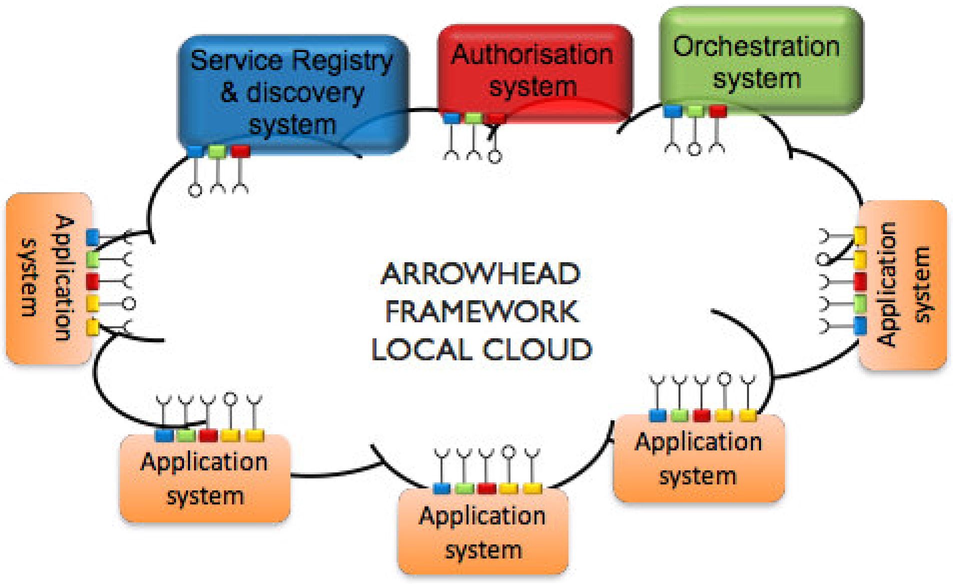


Figure 1 - Arrohead Framework Local Cloud

The Arrowhead Framework also provides a documentation structure for all developers. These documents structure is described in Section 1.2.2.

1.2.2 Arrowhead Documentation

Since this project was proposed to be developed in the scope of the Arrowhead Framework in demands that several documents must be created specifically.

In order to understand these documents there are some definitions that must be explained alongside the purpose of each.

1.2.2.1 Definitions

In this section all the core arrowhead terms used through the development of this project and used to provide the necessary documentation are detailed. These definitions are provided in [8] although not all of them are here defined since they are not being use in the scope of this project:

- A **System** is what is providing and/or consuming services. A System can be the Service Provider of one or more services and at the same time the Service Consumer of one or more services. It normally includes software executing on hardware. It may also be referred to as Component or Device. A system can be user interface display, used to control the air-conditioning within a house, but is can also be a small temperature sensor that complies with the Arrowhead Framework.

- A **Service**, in the context of the Arrowhead Framework, is what is used to exchange information from the providing System to the consuming system. It is based on a number of service orientation principles derived from high level objectives and properties of the Arrowhead framework. Furthermore, a service can be realized by an arbitrary number of service producers and service consumers.
- A **Core Service** is a service offered by the Arrowhead Common Framework core systems. Core services are varied. They address among others security, registry, orchestration and quality issues. The services can be divided into 3 different areas: Information Infrastructure (II), Information Assurance (IA) and System Management (SM).
- The **Information Infrastructure** is the domain of core services and systems mainly in charge of providing support for service registry and service discovery.
- The **Information Assurance** provides support for secure information exchange. The IA provides authorization, authentication, certificate distribution, security logging and service intrusion functionality.
- The **System Management** provides support for late binding and solving system of systems composition. The SM provides logging, monitoring and status functionality. It also addresses orchestration, software distribution, network QoS and performance, configuration and policy.

1.2.2.2 Documents

The Arrowhead documentation follows a very specific structure and provides developers with guidelines that aids in the planning phase and allows a clear understanding of what is being done and how it’s done [paper IECON2014].

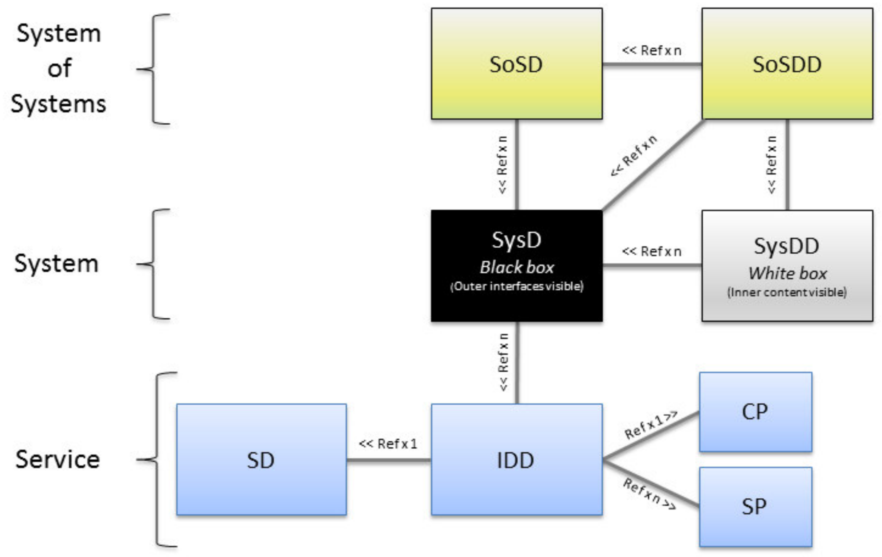


Figure 2 - Arrowhead Documentation Structure

During this project I've been working, alongside my supervisors, on four document types. The **System of Systems Description (SysD)**, **Service Description (SD)**, **Semantic Profile (SP)** and the **Interface Design Description (IDD)**.

The **System of Systems Description** Is a high level view, describing how system of systems main functionalities have been technologically implemented, i.e. which technologies have been used and how it is physically implemented. In the case of the *Eventhandler*, a brief overview is provided alongside the configuration properties files description. The core use cases are described in a table format and a UML Sequence diagram with the step by step procedures of this system. Finally, there is a reference to the application services, these application services are simply what services are being consumed by the *Eventhandler* and what services are being produced.

Following the System of Systems Description, the **Service Description** provides an abstract description of the purpose and behaviour of a specific service, including what information it is aiming to distribute. It is referred to by one or more IDD (each stating a way of implementing the service with a specific technology). In this document there is also a brief overview of the service purpose and what entities are involved in the process. A UML Component diagram provides a clear picture of what interfaces are being exposed and what functions does it offers. Finally, a sequence diagram allows the readers to understand how the service works and how should it be used.

After describing all the *Eventhandler* services the **Semantic Profile** offers a description of the data format and what is the type of the encoding, in this case XML and JSON.

Lastly the **Interface Design Description** offers a detailed description of how a service is implemented/realized. The core section for the IDD documents is the *interfaces* section which provides a table with all the functions, the path to access it, the method used (POST, PUT, GET, DELETE, etc.) and the input and output.

1.2.3 Eventhandler System

The project proposed to be developed is an event handling application based on the SOA (Service Oriented Architecture) and the producer/consumer paradigm. The *Eventhandler* should provide 3 core services. The *Registry* service which is responsible for the registry of all the Event Producer and Event Consumers that are part of the system. The *Publish Events* service will handle all incoming events received from Event Producers and forward them according to filtering rules to the *Notify* service of all interested Subscribers (Event Consumers). The *Notify* service is not provided by the *Eventhandler* but should be provided by all Subscribers, so implementing this service was also a big part of this project. Lastly the *GetHistoricalData* service must be able to provide applications to retrieve data from a permanent data storage source, such as a database or a file. Providing a filter, applications must be able to get information relative to events published in the past. This information may be: the subscribers that received the event, the payload or the timestamp of when it was received.

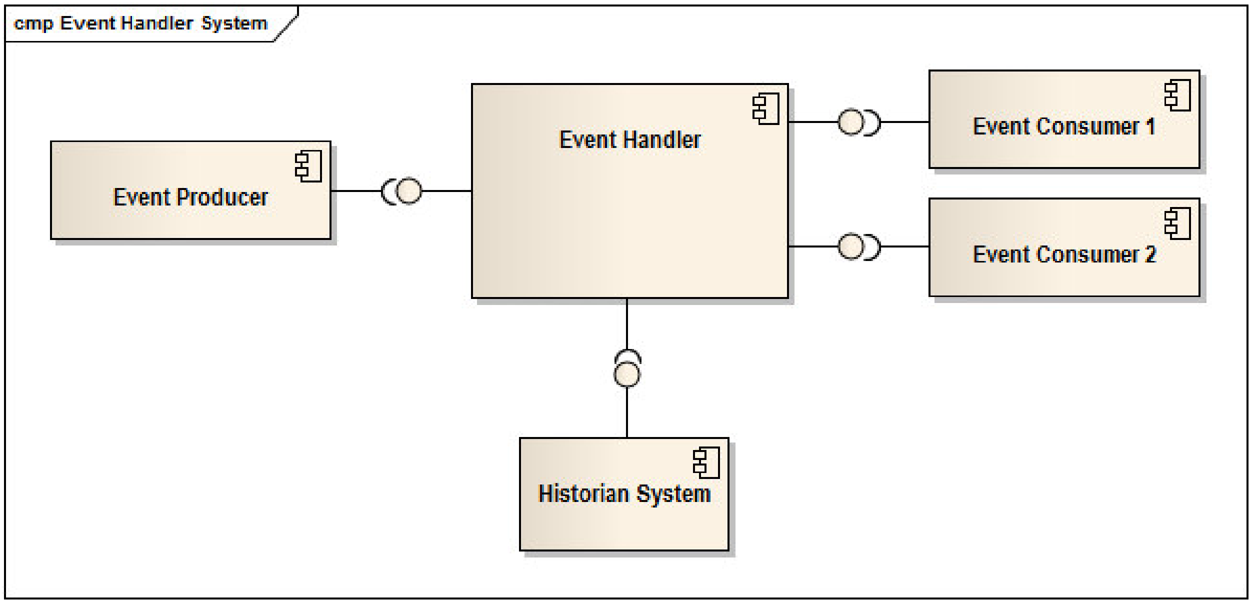


Figure 3 - Eventhandler Overview

1.3 <u>CISTER Research Centre</u>

CISTER (Research Centre in Real-Time and Embedded Computing Systems) is a top-ranked research unit based at the School of Engineering (ISEP) of the Polytechnic Institute of Porto (IPP). PhD programs are available in the Research Centre and various publications and articles are constantly being published. It provides a fully professional environment and not disregarding the interaction between co-workers, with a coffee-break we are allowed to spend some time eating breakfast and interacting with people from various countries and cultures which I found fantastic.

Being internationally recognized CISTER is involved in several European projects of major relevance:

MANTIS - Cyber Physical System Based Proactive Collaborative Maintenance.

EnerGAware - Energy Game for Awareness of energy efficiency in social housing communities.

EMC^2-Embedded multi-core systems for mixed criticality applications in dynamic and changeable real-time environments.

1.4 Contributions

Despite being recently completed and still susceptible to change, this project will be a part of the Arrowhead Framework core systems, this means that all applications in an Arrowhead cloud can connect and use the functionalities developed in the scope of this project.

One good example of the *Eventhandler* being used by third applications is in the *QoS Manager* and *QoS Monitor*, both of these projects were developed at CISTER by my colleagues in ISEP Paulo Barbosa and Renato Ayres as a final project of PESTI.

The *QoS Manager* and *QoS Monitor* applications will use the *Eventhandler* to send events regarding the QoS operations between the several nodes. These messages may contain errors, faults and logs.

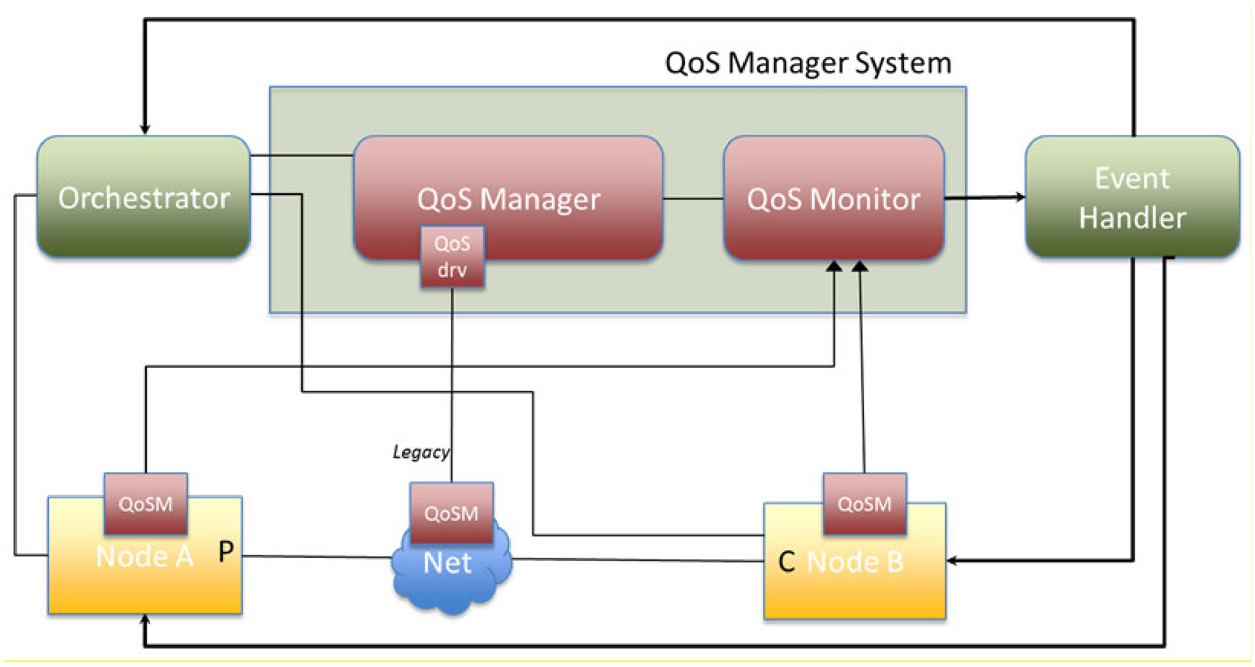


Figure 4 - QoS Manager/Monitor

1.5 Document Structure

This subsection describes the structure and organization of this document. There will be a brief description of each chapter along with its main objectives.

Chapter 1, Introduction, gives a small introduction to this work. This is achieved by presenting the project itself and describing the context in which it is inserted. There is also a small description of the company and what were the major contributions of this project.

Chapter 2, Scope, introduces the problem and what features should the applications provide in order to solve these problems. It is made a reference to the business area in which this project is inserted and also a reference to the state of the art.

Chapter 3, Working Methodologies, after the first two chapters the project is fully presented and we can move over to a more technical explanation. In this third chapter the working methodologies used by me and my supervisors are described and explained. How the several stages of the project were planned is the main focus of this chapter. Finally, we reference all the technologies used in order to be able to develop the *Eventhandler*.

Chapter 4, Technical Description, focuses greatly on the analysis, modelation and development of the solution for the problem, it is in this section that most of the UML diagrams are found along with the code developed and tests. Actors and stakeholders are presented but most importantly the functional and non-functional requirements are mapped. Making use of a use case diagram to illustrate all the functional requirements and tables to explain these requirements with more detail. The Arrowhead documentation is presented some of which abbreviated with the objective of pointing only to core topics regarding this document. Lastly, the code developed to provide a solution is presented along with the functional, unit and stress tests used make sure that all the requirements are met.

Chapter 5, Conclusions, this final chapter is used to provide the results of the work developed. Starting with a summary of this document and also referencing what goals were achieved or not. The main difficulties encountered along all the phases of development along with future improvements to be made mostly connected to technical details. To conclude, a final appreciation is conducted describing the benefits that being involved in a grand scale project like this brought me.

2 Scope

In this section a summary the project summary is provided, as well as the development process used, business areas involved and all the essential technologies during the development of the *Eventhandler*.

2.1 Problem

The *Eventhandler* is part of the Arrowhead Framework, which aims to apply Service Oriented Architecture to the embedded systems' world. The *Eventhandler* is a component that supports the handling of events, and in that sense it enriches service-oriented applications with the capabilities of interacting via the publish/subscribe paradigm. In fact, the *Eventhandler* core system is in charge of the notification of events that occur in a given Arrowhead compliant installation, manages producers and consumers of events, allows filtering of messages, and manages historical data regarding events. This latter capability is performed either on local files, on a database, or through another component of the Arrowhead Framework - the Historian system. Two examples of the application of the *Eventhandler* are described: the management of application faults, and the support to quality of service of orchestrated services.

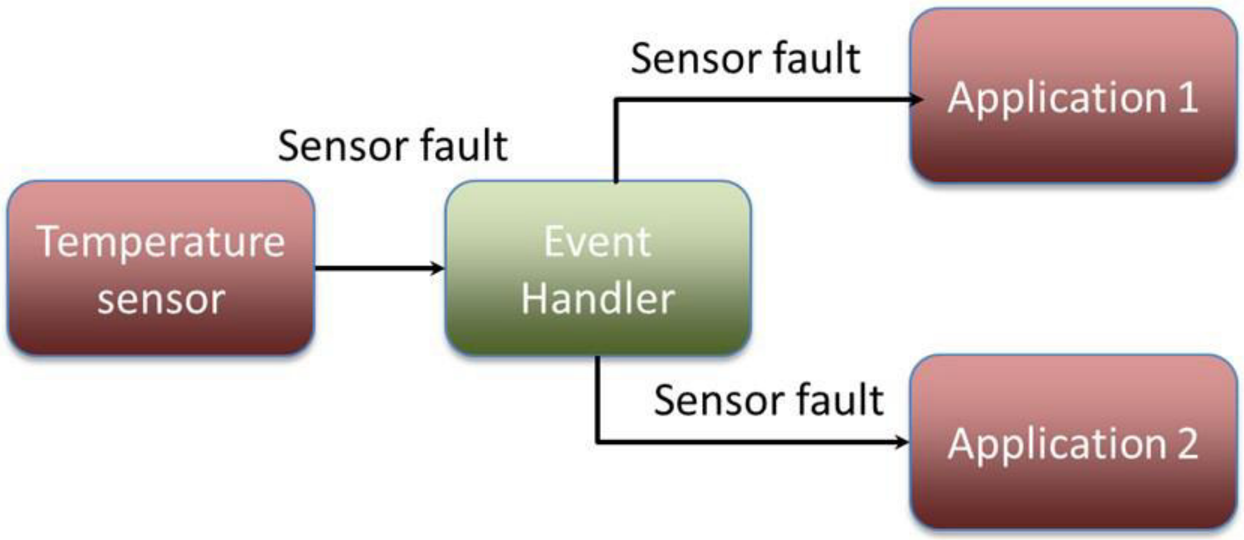


Figure 5 - Eventhandler Applications Fault

2.2 Business Area

The main topic regarding the business area of this project is the IoT (Internet of Things). This section will focus greatly on this topic and it will provide some examples gathered through research that I've conducted. I would also like to highlight some of the investments that are being done in this area alongside some of the prospects for the evolution of this field of research.

In Figure 5 we have an image that illustrates the complete "Ecosystem" of the IoT field.

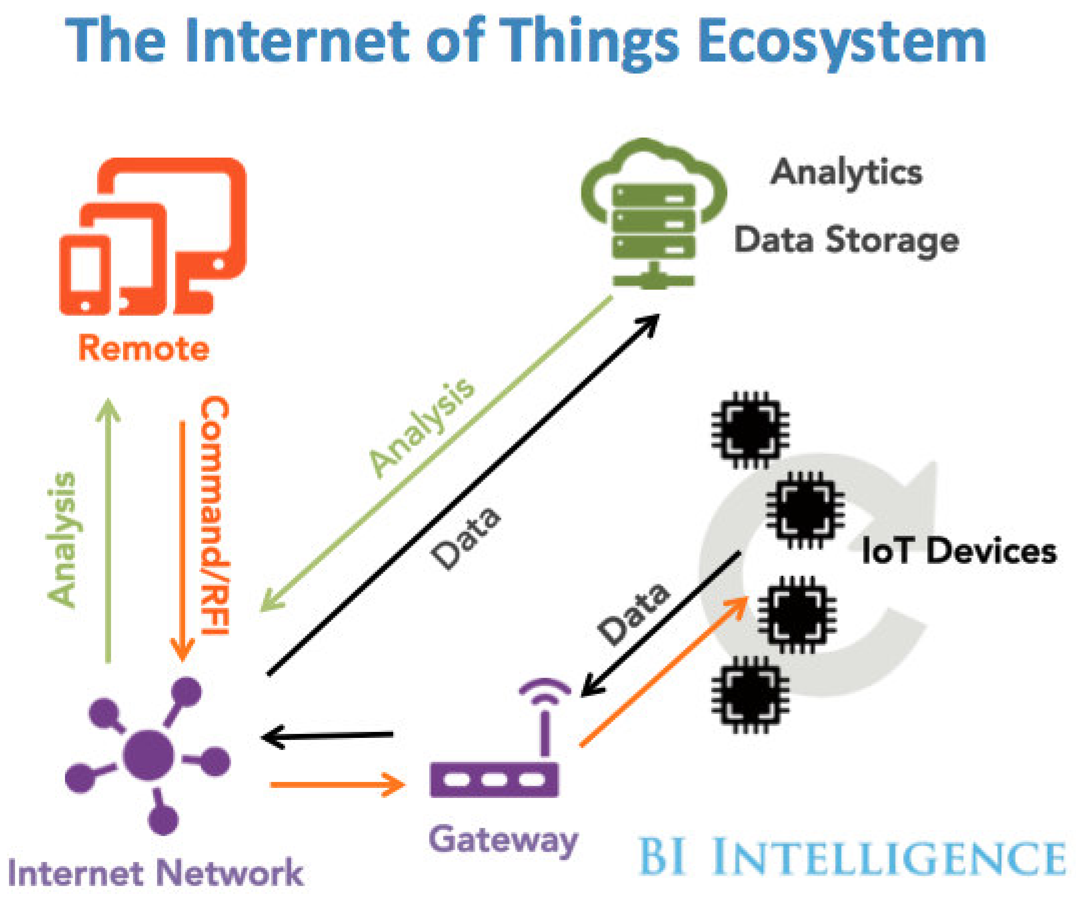


Figure 6 - Internet of Things Ecosystem

The following research was conducted by the firm IDC and shows the amount of money invested in Internet of Things hardware, software, services and connectivity.

As shown in figure 5, this investment will reach 232 billion dollars in the year 2016, it will keep growing at a rate of approximately 16% and will reach more than 350 billion dollars in 2019 [June 2016, IDC report].

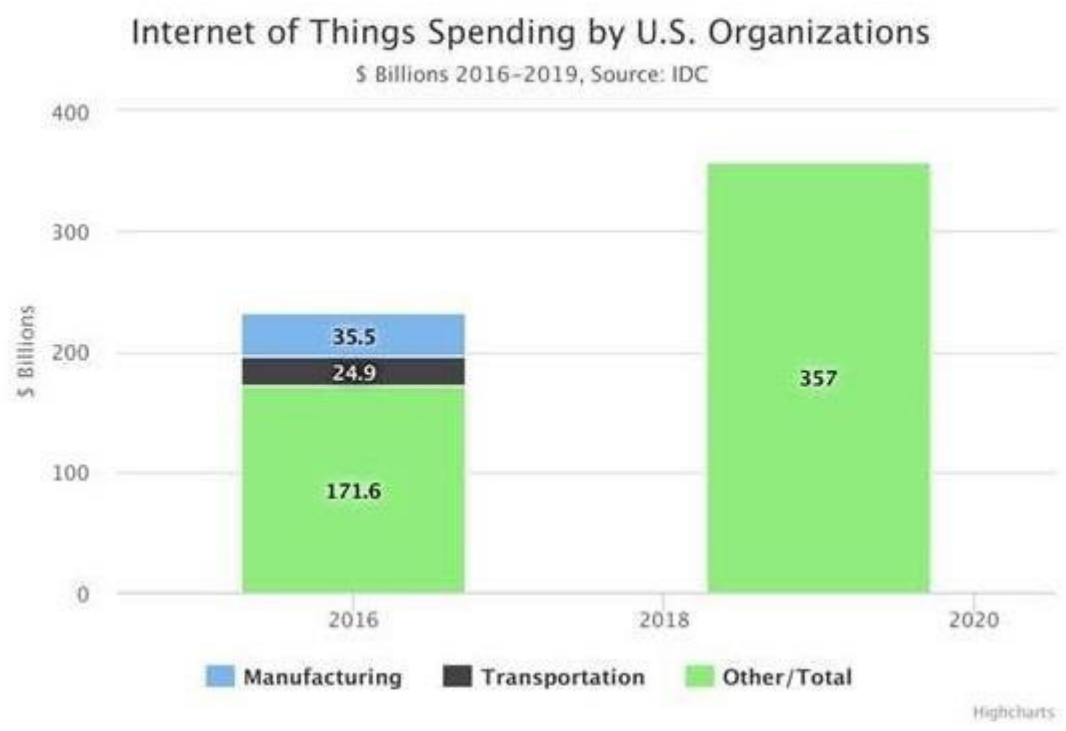


Figure 7 - Internet of Things Spending by U.S Organizations

Finally, here are a few sections of an article which were drawn from a series of Goldman Sachs research reports on the Internet of Things that has included contributions from more than 20 analysts across multiple sectors.

This information was taken from the article referenced in [9].

“The Internet of Things is emerging as the third wave in the development of the internet. While the fixed internet that grew up in the 1990s connected 1 billion users via PCs, and the mobile internet of the 2000s connected 2 billion users via smartphones, the IoT is expected to connect 28 billion “things” to the internet by 2020, ranging from wearable devices such as smartwatches to automobiles, appliances, and industrial equipment. The repercussions span industries and regions”.

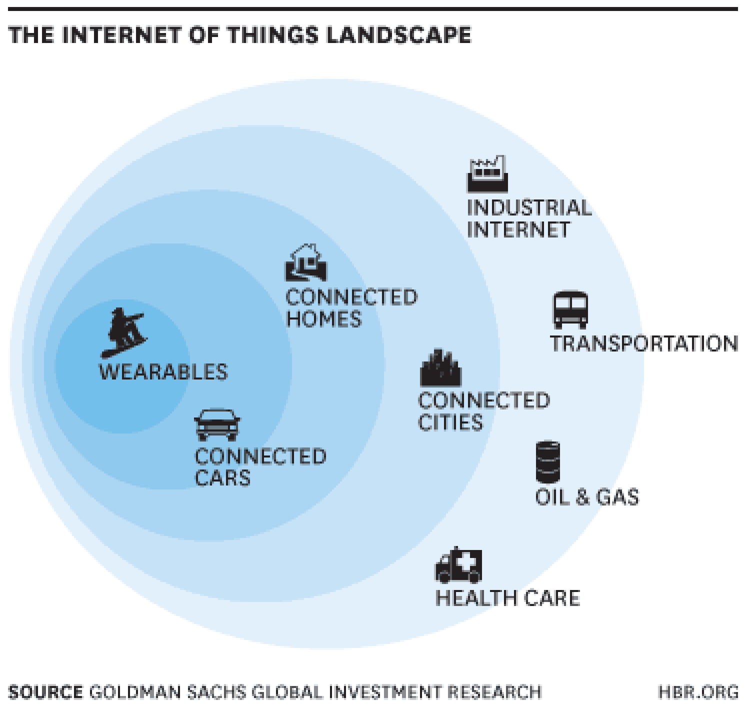


Figure 8 - Internet of Things Landscape

“In connected cities, the U.S. has emerged as a leading adopter of smart meter technology for power utilities, approaching 50% penetration of 150 million total endpoints. The initial foray into connected cities was catalyzed by over 3 billion dollars in stimulus funding and support for smart grid technology as part of the 2009 American Recovery and Restoration Act. Government initiatives are likely to drive growth internationally as well. In Europe there is a target for 80% of households to have smart meters by 2020”.

“Smart meters and the grid network architecture lay the foundation for further connectivity throughout cities, including smart street lighting, parking meters, traffic lights, electric vehicle charging, and others. According to The Climate Group, a non-profit organization dedicated to reducing carbon use, combining LED lamps in streetlights with smart controls can reduce CO2 emissions by 50%-70%”.

“Within the vast Industrials sector, the IoT represents a structural change akin to the industrial revolution. Equipment is becoming more digitized and more connected, establishing networks between machines, humans, and the internet and creating new ecosystems. While we are still in the nascent stages of adoption, we believe the Industrial IoT opportunity could amount to 2 trillion dollars by 2020. Included within this

Industrial category are numerous sectors, from transportation to health care to oil and gas, each of which will be affected”.

“As with any gold rush, the early winners from the IoT are likely to be the suppliers selling the “shovels” to make the connections possible and to process the vast amounts of data. But in the long run, the ultimate impact of this third wave of the Internet depends on the adopters in these proving grounds finding gold in connecting billions of devices into an intelligent network.”

We can conclude from this article that in fact the IoT is a booming market and a field of expertize that will still evolve in the near future.

3 Work Environment

For the first month it was necessary to understand all the concepts regarding the Arrowhead Framework and the technologies they are using to implement their applications. After reading most of the documentation provided it was necessary to start practicing with some of the technologies that I never worked with during the course in ISEP, these technologies are described in the subsection bellow.

After this first month the planning of the project had to be made. It was proposed by my supervisors that we should have daily meetings at the start of each day. These meeting had two main purposes. Firstly, to verify the progress of the development, secondly to brain storm new ideas regarding all the different procedures that the application must be able to accomplish and how could they be achieved.

The Arrowhead documentation had to be completed in parallel to the development, and a full section will be dedicated to this topic.

Finally, after all the projects requirements were completed we proceeded to the testing phase where the application had to pass several stress and functional tests.

3.1 Working Methodologies

During the development of the *Eventhandler* there was a need to present several demos of the application in order to get feedback from external developers who were also involved in this project. So we adapted to an agile development method.

The word “agile” is the new software equivalent to “lean” operations, processes, and startups. Agile development is a new method in software development that depends on layering development and iteration instead of pushing one ‘complete’ product to the market. Developers will learn from market and user’s feedback how to optimize software, remove or add new features [19].

Using this working methodology we we’re able to adapt to the “client” requirements and develop the application according to their feedback on what they thought it was good and what they thought could be improved.

Figure 2 (Agile Development approach) illustrates the steps of this development methodology).

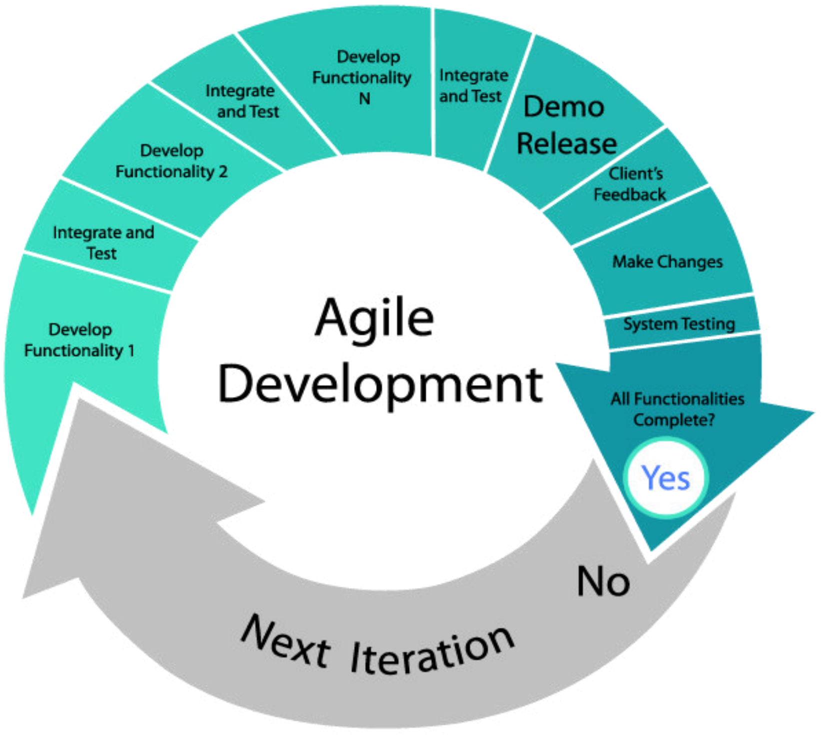


Figure 9 - Agile Development approach

3.2 <u>Planning</u>

The project's first working step was to draw a planning map in which it described the first phases of the project development. During the first month or so it was proposed that I should focus on learning all the technologies (mainly the ones that are not learned during my studies in ISEP) with which the application will be developed. Parallel to this task I also needed to read most of the Arrowhead documentation, most of which I had access through the Arrowhead SVN in the cloud.

After this first month me and my supervisors held daily meetings. In these meetings we brainstormed about the core features of the application and how these features could be implemented, we discussed how could the application be integrated with the Arrowhead Framework and later what was the current status of the project and how to improve current implementations.

It is proposed to the student to develop a Gantt diagram at the start of the project.

A Gantt chart is a horizontal bar chart developed as a production control tool in 1917 by Henry L. Gantt, an American engineer and social scientist. Frequently used in project management, a Gantt chart provides a graphical illustration of a schedule that helps to plan, coordinate, and track specific tasks in a project (Definition from Whatis.com).

The respective Gantt diagram can be found in Section 7.1.

3.3 Technologies

In this section all core technologies used to develop this project are mentioned and explained with more detail.

3.3.1 Java

Java is a class-based and object-oriented computer programming language that provides a lot of features and characteristics that are crucial to this project. Assuming the number of devices that could be using the *Eventhandler* application in the future, portability is a core requirement provided by Java since it can run in almost all the popular platforms. Having to deal with multiple requests from multiple users constantly, speed and security are crucial and are also provided by Java. With an incredible number of standard API and simple grammar Java was the primary choice for the development of this project.

3.3.2 Maven

Apache Maven is a software project management and comprehension tool that provides several powerful functions. A POM (Project Object Model) file written in XML describes the software project being built, its dependencies on other external modules and components, the build order, directories, and required plug-ins. This was one of the most useful tools during the development phase mostly because of the testing, building and dependency management provided.

3.3.3 REST

Representational State Transfer is the software architectural style of the World Wide Web. It relies on a stateless, client-server communications protocol. The Eventhandler application uses HTTP requests to post data (create or update), read data (queries), and delete data. REST uses HTTP for all four CRUD operations.

3.3.4 Jersey

Jersey RESTful Web Services framework is an open source, production quality, framework for developing RESTful Web Services in Java. Jersey provides its own API that extends the JAX-RS toolkit with additional features and utilities to further simplify RESTful service and client development. Using Maven, Jersey can be imported to the application as a dependency.

3.3.5 Jetty

Originally developed in the Sydney suburb of Balmain by software engineer Greg Wilkins, Jetty is a free and open source project that provides a Java HTTP (Web) server and Java Servlet container. While Web Servers are usually associated with serving documents to people, Jetty is now often used for machine to machine communications, usually within larger software frameworks. This framework is used mainly in the *Eventhandler* and Event Consumer applications in order to receive the HTTP requests.

3.3.6 LOG4J

Apache log4j is a Java-based logging utility. It was originally written by Ceki Gülcü and is now a project of the Apache Software Foundation.

Storage is one of the most important requisites of this project and saving information in a local file is a key function. I found this technology very helpful because it is very well documented, uses a single file for all its configuration and requires only one line of code to write a message to a file.

3.3.7 MySQL

MySQL is “the world’s most popular open source database’ (MySQL, 2015), with also an open source version. It uses SQL language as interface. It is easy to use and haves a good performance and stability.

In this project, a MySQL database was created in an Ubuntu Server operating system to enable permanent storage of all information deemed relevant for the application.

3.3.8 Git

Git is a distributed revision control system, ideal for data integrity and data/version tracking designed to handle projects with speed and efficiency. The operations are performed locally giving it an advantage in terms of speed because it doesn’t communicate constantly to a server. Since it was originally built to work on the Linux Kernel, it has the ability to handle small or large projects. It is ideal when multiple developers need to synchronize their work and where data needs to be controlled and kept safe.

4 Technical Description

This section contains an overview of the technical description to allow a better comprehension of the problem followed by the requirements analysis in Sections 4.1.1, 4.1.2, 4.1.3 and 4.1.4. In the requirements analysis the stakeholders and actors involving this project are represented and described, followed by all the functional and nonfunctional requirements. The domain model is presented is to allow a better understanding of all the core entities involved in the communications process, Section 4.1.5. Lastly, all the Arrowhead documentation created in the scope of this project is presented in detail, Sections 4.1.6 to 4.1.12.

4.1 Analysis and Modeling

The analysis and modeling of the problem was essential for a better understanding of the project and its possible solutions. The requirements analysis was divided into two categories, functional and non-functional requirements. Functional requirements largely correspond to features that have been requested for the application, while non-functional requirements are often called "quality attributes" of a system. Other terms for non-functional requirements are "qualities", "quality goals", "quality of service requirements", "constraints" and "non-behavioral requirements". Some examples of non-functional requirements are: execution qualities, such as security and usability and evolution qualities, such as testability, maintainability, extensibility and scalability [16].

The requirements analysis is a software engineering process that includes the search, analysis, documentation and requirements/restrictions verification for a software product. This process should allow a deeper comprehension of the problem. Being as core step in the software development process, the success of this project depends on a clear understanding of the problem. All functional and nonfunctional requirements will be described with all the necessary detail. These requirements where settled during daily meetings with both supervisors and where changed along the development of this project.

4.1.1 **Stakeholders**

A *stakeholder* in the architecture of a system is an individual, team, organization, or classes thereof, having an interest in the realization of the system (Software Systems Architecture, 2nd Edition). After knowing these facts, it is of most importance that all stakeholders are clearly defined.

The following stakeholders were identified:

- System Administrators: Will run and administer the system once it is deployed.
- Developers: Each producer/subscriber application will need to be implemented according to the business requisites from where it has been deployed.
- Organizations: Represent all entities that will ultimately make use of all the providing services.

4.1.2 **System Actors**

An actor is a role represented by an entity, in this case, an application, that will interact with the system and will make use of some, or all its functionalities/services being so, able to make decisions.

There are clearly three actors in this system:

- *Eventhandler* System: It is the core application, responsible for the registry of the other actors, handling and forwarding incoming events and the storage of all interesting data.
- Event Producer: It is an application that will create and send events to the *Eventhandler* System.
- Event Consumer: It is an application that will consume events according to a specific filter.

4.1.3 **Functional Requirements**

In this subsection all functional requirements proposed are depicted and described in detail.

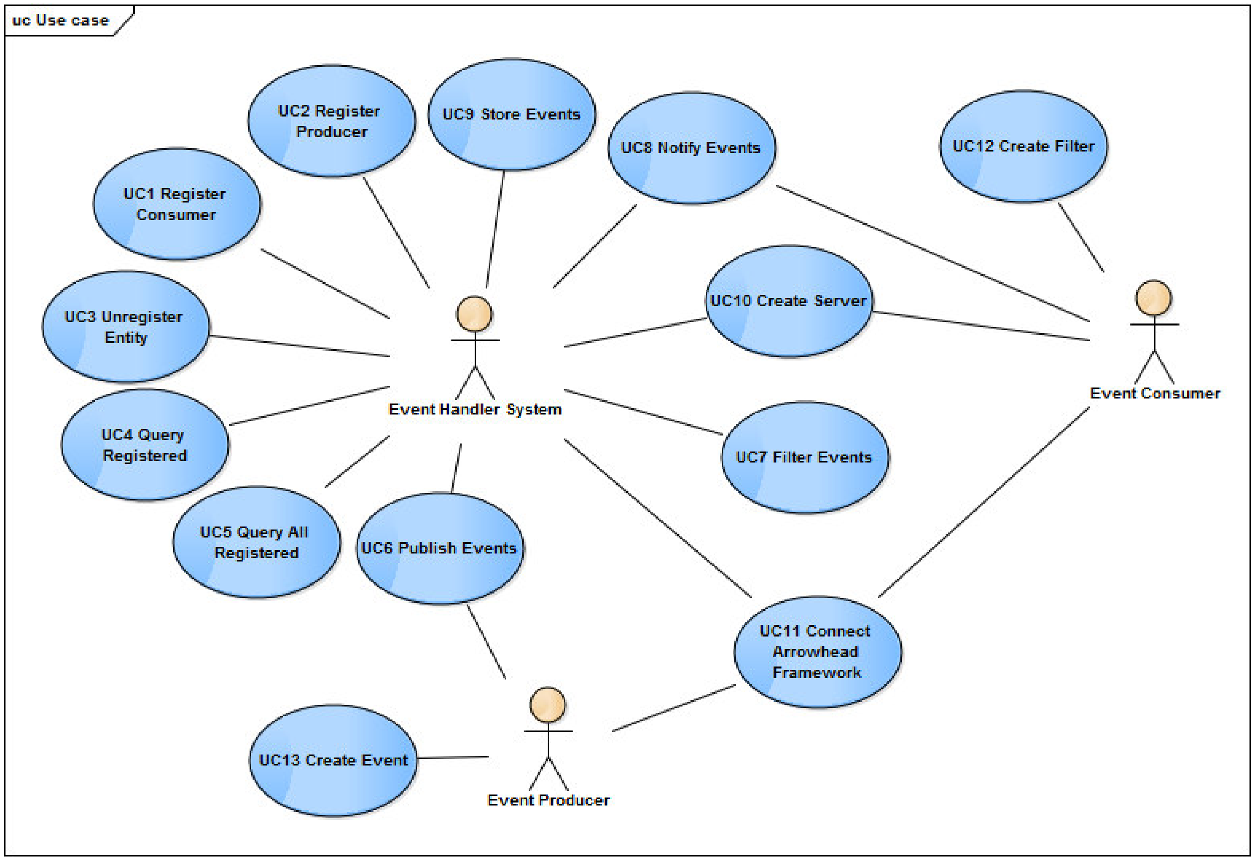


Figure 10 - Use Case Diagram

A use case is a methodology used in system analysis to identify, clarify, and organize system requirements. The use case is made up of a set of possible sequences of interactions between systems and users in a particular environment and related to a particular goal. It consists of a group of elements (for example, classes and interfaces) that can be used together in a way that will have an effect larger than the sum of the separate elements combined. The use case should contain all system activities that have significance to the users. A use case can be thought of as a collection of possible scenarios related to a particular goal, indeed, the use case and goal are sometimes considered to be synonymous. (searchsoftwarequality.techtarget.com).

Figure 33 represents all the use cases for this particular problem. The following subsection explains these use cases using the format provided in the Software Development lectures.

Use case 1 – Register Consumer

ACTOR	**EVENTHANDLER SYSTEM**
OBJECTIVE	Register an Event Consumer in the system.
PRE CONDITIONS	None
POST CONDITIONS	None
FLOW	1. The Event Consumer uses the *registerConsumer()* function of the *Registry* service of the *Eventhandler* providing an unique identifier(uid) 2. The *Eventhandler* sends a HTTP response status
ALTERNATIVE PATHS	None

Table 1 - Use Case 1

In order to interact with the *Eventhandler*, all entities must firstly be registered. Using the *Registry* service of the *Eventhandler* system, an Event Consumer can use the *registerConsumer()* function as long as it provides an unique identifier that is not already registered. If so an OK HTTP message will be received.

Use case 2 – Register Producer

ACTOR	**EVENTHANDLER SYSTEM**
OBJECTIVE	Register an Event Producer in the system.
PRE CONDITIONS	None
POST CONDITIONS	None
FLOW	1. The Event Producer uses the *registerProducer()* function of the *Registry* service providing an unique identifier(uid)

	2. The *Eventhandler* sends a HTTP response status
ALTERNATIVE PATHS	None

Table 2 - Use Case 2

The principle used in this use case is exactly the same used in the previous (Use Case 1 – Table 11). The only difference being the data sent alongside the unique identifier which is described with more detail in Section 4.1.7.4.

Use case 3 – Unregister Entity

ACTOR	**EVENTHANDLER SYSTEM**
OBJECTIVE	Unregister an Event Consumer or Producer from the system
PRE CONDITIONS	The entity requesting this option must be registered in the system
POST CONDITIONS	None
FLOW	1. An entity request the *Eventhandler* to be unregistered using the *unregisterEntity()* function of the *Registry* service providing an unique identifier(uid) 2. The *Eventhandler* sends a HTTP response status
ALTERNATIVE PATHS	None

Table 3 - Use Case 3

This use case describes the removal of a consumer or a producer from the system. In the case of the Event Consumer, the system will cease to receive events. In the case of

Event Producer, the *Eventhandler* will be informed that the Event Producer does not exist anymore, and can take actions such as notifying Event Consumers subscribing to the Event Producer that this service is no longer being provided.

Use case 4 – Query Registered

ACTOR	**EVENTHANDLER SYSTEM**
OBJECTIVE	Obtain a list of Event Consumers and Producers that match a specific query
PRE CONDITIONS	None
POST CONDITIONS	None
FLOW	1. An entity sends a query to the *Eventhandler* using the *query ()* function of the *Registry* service 2. The *Eventhandler* produces a list of matching Consumers/Producers and sends it to the application performing the query
ALTERNATIVE PATHS	None

Table 4 - Use Case 4

As exposed in Table 13 the core purpose of this solution is to provide applications the capability of getting a list containing all entities that match a list of parameters. These parameters are embedded in a filter which is described in Section 4.1.7.3.3.

Use case 5 – Query All Registered

ACTOR	**EVENTHANDLER SYSTEM**

OBJECTIVE	Obtain the list of all registered Event Consumers and Producers
PRE CONDITIONS	None
POST CONDITIONS	None
FLOW	1. An application uses the *queryAll ()* function of the *Registry* service 2. The *Eventhandler* creates a list containing all registered Consumers/Producers and sends it to the application performing the query
ALTERNATIVE PATHS	None

Table 5 - Use Case 5

This is a simpler version of the previous use case (Use case 4 – Table 13) where instead of sending a query to the *Eventhandler* an application can use the *queryAllRegistred()* function and receive a list of all the entities currently registered with the *Eventhandler*.

Use case 6 – Publish Events

ACTOR	**EVENTHANDLER SYSTEM**
OBJECTIVE	Event Producers will use this service to send Events to the *Eventhandler*
PRE CONDITIONS	None
POST CONDITIONS	None
FLOW	1. An Event Producer uses the *publishEvent()* function of the *Publish Events* service to send an event

	2. The *Eventhandler* receives this event and creates a list of consumers to be notified according to filtering rules 3. The *Eventhandler* replies to the Producer with an acknowledgement confirming that the event was published
ALTERNATIVE PATHS	None

Table 6 - Use Case 6

When an interesting event is triggered in an Event Producer the function `publishEvent()` is used to send all necessary data to the *Eventhandler*. Events received are filtered and forwarded to Event Consumers that had subscribed to this type of event. At the end of the workflow, an acknowledgment message is returned from the *Eventhandler* to the *Producer*, this process is clearly described in Section 4.1.8.

Use case 7 – Filter Events

ACTOR	**EVENTHANDLER SYSTEM**
OBJECTIVE	Create a list of Subscribers to be notified with the incoming event.
PRE CONDITIONS	An event had to be received through the *Publish Events* service
POST CONDITIONS	None
FLOW	1. An event is received though the *Publish Events* service 2. The *Eventhandler* applies filtering rules and creates a list of Consumers to be notified

ALTERNATIVE PATHS	None

Table 7 - Use Case 7

This a task performed in the middle of the communication process and completely hidden to all other entities but it is crucial for the process involving the next use case (Use case 8 – Table 17).

After receiving an event the *Eventhandler* uses filters and creates a list of interested *Consumers*.

Use case 8 – Notify Events

ACTOR	**EVENTHANDLER SYSTEM**
OBJECTIVE	Notify Events to interested Consumers
PRE CONDITIONS	An event must be received
POST CONDITIONS	None
FLOW	1. An event is received though the *Publish Events* service 2. The *Eventhandler* notifies the event to interested Consumers using the function *NotifyEvent()*
ALTERNATIVE PATHS	None

Table 8 - Use Case 8

When an event is received the process detailed in the previous use case (Use 7 – Table16) is triggered and all the interested *Consumers* are notified using the `NotifyEvent()` function which is invoked by the *Eventhandler.* This function is provided by *Consumers* in order to receive data. This process is explained with more detail in the Arrowhead documentation is section 4.1.10.

Use case 9 – Store Events

ACTOR	EVENTHANDLER SYSTEM
OBJECTIVE	Permanently store information regarding events
PRE CONDITIONS	An event must be received
POST CONDITIONS	None
FLOW	3. An event is received though the *Publish Events* service 4. The *Eventhandler* stores information in a database, local file or both
ALTERNATIVE PATHS	None

Table 9 - Use Case 9

The permanent storage of information is one of the core objectives of this project. Either using a local file, a database (internal or external) or the Arrowhead's Historian service, data must be stored permanently so that it can be later accessed by the *GetHistoricals* service.

Use case 10 – Create Server

ACTOR	EVENTHANDLER SYSTEM, EVENT CONSUMER
OBJECTIVE	Create a local HTTP server in order to received data
PRE CONDITIONS	None
POST CONDITIONS	None
FLOW	1. All initial configuration of Jetty must be defined 2. Start HTTP server
ALTERNATIVE PATHS	None

Table 10 - Use Case 10

The communication system used in the scope of this project is mainly REST which is described in the Arrowhead Communication Profile [24]. REST is an abstraction of the World Wide Web therefore takes use of the HTTP protocol. Data is transferred across the network and its receivers must be able to interpret HTTP therefore the *Eventhandler* and the Event Consumers must create a local server to receive all information.

Use case 11 – Connection to the Arrowhead Framework

ACTOR	**EVENTHANDLER SYSTEM, EVENT CONSUMER, EVENT PRODUCER**
OBJECTIVE	Connect to a local or remote Arrowhead Cloud
PRE CONDITIONS	Arrowhead configuration file with correct settings
POST CONDITIONS	None
FLOW	1. Edit Arrowhead configuration file with the appropriate settings 2. Use the *connectACS()* function to connect to the Arrowhead cloud
ALTERNATIVE PATHS	None

Table 11 - Use Case 11

As explained in Section 2, the *Eventhandler* will be used in the Arrowhead framework context so, all entities must be able to connect to the Arrowhead cloud either if it is located in a local or a remote network.

Use case 12 – Create filter

ACTOR	**EVENT CONSUMER**
OBJECTIVE	Create a subscription filter
PRE CONDITIONS	None
POST CONDITIONS	None

FLOW	1. Edit the subscriber configuration file to set all filter properties accordingly 2. Use the *setFilter()* function so create the filter
ALTERNATIVE PATHS	None

Table 12 - Use Case 12

The filtering attributes are one of the main functions of the *Eventhandler* because they are used at run time and allow the creation of a list of interested subscribers every time a new event arrives.

Use case 13 – Create event

ACTOR	**EVENT PRODUCER**
OBJECTIVE	Generate an event
PRE CONDITIONS	Depending on the producer's context something will trigger an event
POST CONDITIONS	None
FLOW	1. An event creation is triggered 2. Using the function *createEvent()* a producer application creates an event
ALTERNATIVE PATHS	None

Table 13 - Use Case 13

Depending on the context the Event Producer is inserted in, an event generation process will be triggered and the application must have the capability to create an event so it can be forwarded to the *Eventhandler*.

4.1.4 Non Functional Requirements

Non-functional requirements cover all the remaining requirements which are not covered by the functional requirements. They specify criteria that judge the operation of a system, rather than specific behaviors.

A non-functional requirement for a hard hat might be “must not break under pressure of less than 10,000 PSI” (Ulf Eriksson, reqtest.com).

4.1.4.1 Portability

As depicted in the Business Area chapter this application will be in an IoT schema, so portability is a core requirement for this project so it must be able to support most of the platforms currently on the market.

4.1.4.2 Usability

The reason for the creation of the *Eventhandler* is the possibility to have an automated event handling system which requires minimal configuration and at the same time is must be robust and trustworthy.

4.1.4.3 Performance

Since the application must be able to establish a connection to a database in almost all features available, it is imperative that the development of those features is as optimized as possible. The number of accesses to the database and the data transferred in each connection needs to be always taken in account. Having to deal with the possibility of hundreds of requests daily the *Eventhandler* will have to clean all unnecessary information in the database and in the local files as well as in memory.

4.1.4.4 Security

One of the *Eventhandler ‘s* usages will be in the QoS (Quality of Service) system which is currently being developed by the ISEP students referenced in Section 1.4. It is crucial that all data exchanged between applications is secured and in some way encrypted. The arrowhead framework already provides with some security mechanisms that are to be explained in the following section. This nonfunctional requirement is currently underdeveloped and should face improvement in the near future.

4.1.4.5 Availability

With its usage in an industrial environment or as middle man receiving important data of a building energy expenditures (just an example) the *Eventhandler* application must be available twenty-four hours a day, 365 days a year.

4.1.5 Domain Model

At its worst business logic can be very complex. Rules and logic describe many different cases and slants of behavior, and it's this complexity that objects were designed to work with. A Domain Model creates a web of interconnected objects, where each object represents some meaningful individual, whether as large as a corporation or as small as a single line on an order form (Martin Fowler in *Patterns of Enterprise Application Architecture*).

4.1.6 System Description

The concepts of the *System Description*, *Service* Description, *Semantic* Profile and *Interface Design Description* are described in Section 1.2.2.2.

The *Eventhandler* core system provides functionality for the notification of events that occur in a given Arrowhead compliant installation.

A high level view of the *Eventhandler* is shown in Figure 1. The *Eventhandler* receives the events from Event Producers and forwards them to subscribing Event Consumers. The following list details the main actors for such a system and their roles:

- **Event Producer**: is the component that creates an event and sends it to the *Eventhandler*.
- **Eventhandler**: is the component that logs events to persistent storage, registers producers and consumers of event, applies filtering rules to event distribution.
- **Event Consumer**: is the component that consumes the events, forwarded by the *Eventhandler*.
- **Historian**[1]: is the component for storing historical regarding events.

- Depending on the context, an event can represent an exceptional occurrence on a particular system

(e.g.: a value of a variable that reaches a critical level), or a simple change of state. Each event is classified according to a number of fields that represent the event's meta-data. An example of meta-data is the severity level of the event:

- **Debugging**: information collected for debugging purposes
- **Info**: tracing program execution: input/output data, changes in tagged variables, etc. Not to be used in production environment
- **Notification**: state changes, execution of functions, etc
- **Warning**: just a "something might go wrong" notification
- **Error**: malfunction in the system, application failure
- **Critical**: total malfunction of the system, the system need human intervention to continue its operation

The *Eventhandler* has the capability of applying filtering rules to incoming events, based on the meta-data of the event (e.g.: severity level of the events), the system that produced the event, etc. In this way it is possible to restrict forwarding to the Event Consumers of events which are of their interest only.

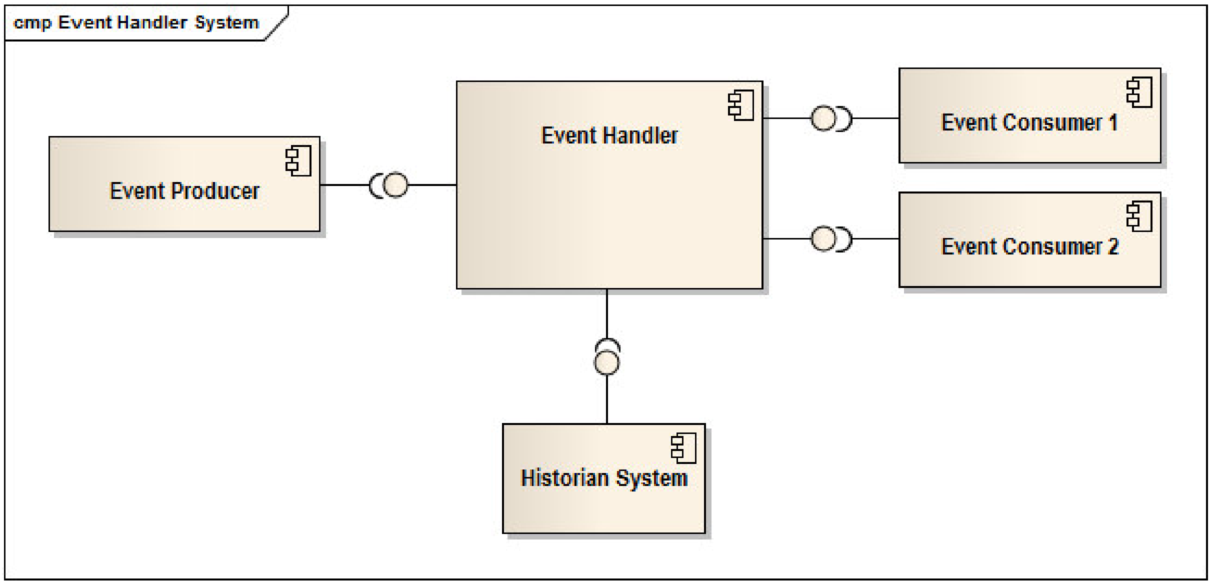


Figure 11 - High Level View of the Eventhandler System

4.1.6.1 Mandatory property files

In order to store events information permanently, the *Eventhandler* can use two strategies: it can store information in a local file, and/or in a database. Currently the *Eventhandler* must be connected to a database in order to properly function so, the database base storage is always assured. The information stored in the log file depends on the definitions given in the *log4j.properties* file and can be cancelled by simply commenting/removing this file.

4.1.6.2 Database Properties

```
# Database
dburl=database_url
username=db_user
password=db_password
driver=java_driver_for_the_database
```

- dburl is the ip address or hostname from where database can be accessed.
- username and password are the credentials to login in the database.
- driver is the java driver that will describe what type of database is being accessed, for example if it is a MySQL or a Microsoft SQL Server.

4.1.6.3 Log4j Properties

```
Local file properties file (log4j.properties)

# Root logger option
log4j.rootLogger=DEBUG, stdout, eventsFile

# Redirect log messages to console
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L -
%m%n

# Rirect log messages to a log file
log4j.appender.eventsFile=org.apache.log4j.RollingFileAppender
log4j.appender.eventsFile.File=log4j-eh.log
log4j.appender.eventsFile.MaxFileSize=5MB
log4j.appender.eventsFile.MaxBackupIndex=10
```

```
log4j.appender.eventsFile.layout=org.apache.log4j.PatternLayout
log4j.appender.eventsFile.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p
%c{1}:%L %m%n
```

To store events information in a local file the log4j properties defines all necessary settings. These settings can be altered using the documentation and manuals referenced in [5].

4.1.6.4 Arrowhead connection properties

```
## Arrowhead core services properties

#arrowhead.server=bnearit
arrowhead.server=hungary

# ARROWHEAD HUNGARY MODULE PROPERTIES

core.server=arrowhead2.tmit.bme.hu
core.tsig=RM/jKKEPYB83peT0DQnYGg==

# ARROWHEAD BNEARIT MODULE PROPERTIES
# Core Services Discovery
#core.server=10.200.0.10
#core.domain=test.bnearit.arrowhead.eu
#core.hostname=localhost
#core.tsig=2qB73d2AFrjlC3tELnBl+g==

# Truststore/keystore
#truststore.file=./eventhandler.jks
#truststore.password=abc1234

# Authorisation
# Backup URL if not found in SR
#authorisation.url=https://10.200.0.10:8181/authorisation-control

# Orchestration
# Backup URL if not found in SR
## Orchestration Store poll interval
#orchestration.monitor.interval=10

# Define supported consumption service types
#service.consume.support=_registry-ws-http._tcp|_registry-ws-https._tcp
```

```
#service.consume.polling.interval=10
#service.consume.support=_publish-ws-http._tcp|_publish-ws-https._tcp
#ervice.consume.polling.interval=10
#service.consume.support=_historicals-ws-http._tcp|_historicals-ws-https._tcp
#service.consume.polling.interval=10
```

There are currently two approaches for connecting to the Arrowhead Framework, using the BnearIT framework or the Hungary framework. The application should connect to either one of these at a given time. Using the Hungary method the application needs only to know the core server `hostname` or `ip address` and the `tsig` in order to use the DNS-SD and all of the Arrowhead services described in Section 1.2.1. Using the REST interfaces provided by the Hungary service registry the *Eventhandler* is able to register and publish its core services.

The BnearIT approach is a bit more complex, similarly to the previous example the *Eventhandler* must be have access to the core server `ip address` and a tsig file although the hostname of the *Eventhandler* is also configured in this file. The main diferences between the two frameworks reside on the security measures implemented. Firstly, in order to use the BnearIT framework a VPN connection must be estabished to their private network, this procedure is detailed in [6]. Secondly, to use the service discovery we must have access to the certificate files corresponding to the ip address that we get, in this properties file there must the a pointer to the corresponding *jks* file (JKS stands for Java KeyStore), this is simply a repository of certificates (signed public keys) and (private) keys. In the case when the application cannot find the authorization URL through the service discovery a backup URL is provided.

4.1.6.5 **Application services**

Figure 11 depicts a representation of the services provided and consumed by the *Eventhandler*. The *Eventhandler* produces 3 services, namely: *Registry* service, the *Publish Events* service, and the *GetHistoricalData* service. The *Registry* service is used to register *Producers* and *Consumers*. The *Eventhandler* produces the *Publish Event* service to allow entities to provide data regarding events. The access to historical data is done through the *GetHistoricalData* interface of the *GetHistoricalData* service, which accesses the data stored on the Historian through its DataOutput service, logs stored on the filesystem or data stored on a database.

The *Eventhandler* also consumes 3 core services, namely: the Authorization Control service, the Service Discovery service and the Orchestration Management service. It is also a consumer of the Notify Events service, and of the DataInput and DataOutput interfaces of the Historian service. All the aforementioned services and interfaces are further explained in the following two sub-sections, except for the DataInput and DataOutput interfaces, which are described in the Historian SD document [4].

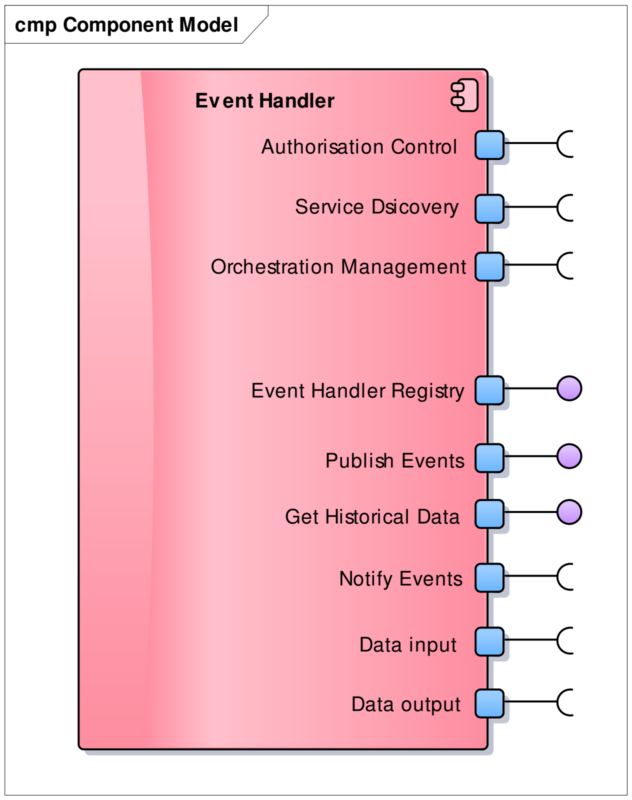


Figure 12 - Eventhandler produced/consumed services

4.1.6.6 Consumed Services

SERVICE	IDD DOCUMENT REFERENCE
SERVICE DISCOVERY	Arrowhead IDD Service Discovery DNS-SD [Klisics, M. (2013). Arrowhead IDD Service Discovery DNS-SD v1.0.0.]
AUTHORIZATION CONTROL	Arrowhead IDD Authorization Control REST_WS [Klisics, M. (2013). Arrowhead IDD Authorisation Control REST_WS v1.0.0]

ORCHESTRATION MANAGEMENT	Arrowhead IDD Orchestration Management REST_WS [Klisics, M. (2013). Arrowhead IDD Orchestration Management REST_WS v1.0.0]
NOTIFY	To be filled
DATAINPUT	Arrowhead SD Historian [Eliasson, J. (2015). Arrowhead SD Historian v0.1]
DATAOUTPUT	Arrowhead SD Historian [Eliasson, J. (2015). Arrowhead SD Historian v0.1]

Table 14 - Pointers to IDD documents

The description of the Service Discovery, Authorization Control and Orchestration Management core services, and of the DataInput and DataOutput services can be found in their respected references.

The Notify service is responsible for the delivery of events to the Event Consumers. The *Eventhandler* accesses the Notify Event service on each subscriber interested in an event, for each event it needs to deliver.

4.1.7 Registry Service Description

This section describes the *Eventhandler* Registry service, including its abstract interfaces and its abstract information model. The purpose of the *Eventhandler* Registry service is storing and keeping track of all the consumers and producers that are registered to the *Eventhandler*. If a consumer wants to receive events, or a producer wants to publish events, first they need to register themselves into the *Eventhandler*. In particular:

- When a system decides to produce events, it registers itself and advertises the kind of events it produces
- When a system decides it wants to receive events, it registers itself as a consumer, also specifying the filtering rules regarding incoming events (kind of events, severity level, etc)

The *Eventhandler* Registry service is also responsible for removing a consumer or a producer from the system, and for modifying the filtering capabilities of the *Eventhandler*, i.e.:

- A consumer can decide to stop receiving all the events, or to change the filtering rules;
- A producer can decide not to publish any more events.

Finally, this service is used to retrieve information regarding registered actors, in particular

- During debugging activities, a system can query this service to retrieve information regarding all the consumers and producers registered into the *Eventhandler*;
- A system can use the service to retrieve information on consumers/producers responding to some criteria.

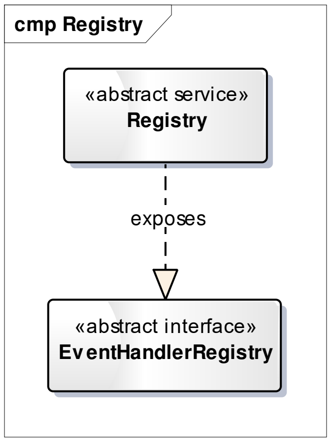


Figure 13 - Eventhandler Registry Overview

4.1.7.1 EventHandlerRegistry

The Event Producers and Event Consumers register themselves against the *Eventhandler* Registry service using the *registerProducer()*and *registerConsumer()* functions, specifying which kinds of events are intended to be sent or received respectively. The function *unRegisterEntity()* is used to remove from the system an Event Consumer or Event Producer. Finally, the functions *query ()* and *queryAll ()* are used to perform queries on the list of Producer and Consumers, and to retrieve the full record regarding Producers or Consumers.

Note: In the UML Sequence Diagrams the **blue** lifelines represent internal *Eventhandler* classes. The **white** lifelines represent external structures.

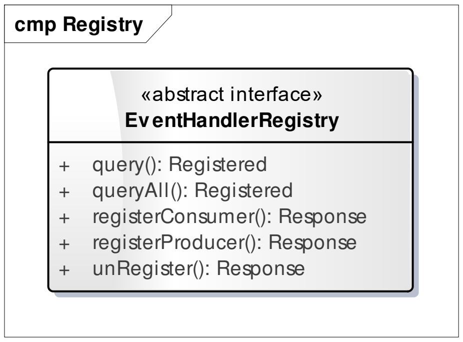


Figure 14 - Functions implemented by EventHandlerRegistry interface

4.1.7.2 <u>Functions</u>

4.1.7.2.1 <u>registerConsumer</u>

The Event Consumer registers itself against the *Eventhandler* Registry using the *registerConsumer()* function. It receives as parameters a unique identifier(uid) and the consumer data both of which are described in section 3.2. This function allows the event Consumer to define a filter that will decide on which events will be received by the Consumer. A unique identifier must be provided to the *Eventhandler*, and a Response status will be returned by this function.

4.1.7.2.2 <u>registerProducer</u>

The Event Producer registers itself against the *Eventhandler* Registry using the *registerProducer()* function. . It receives as parameters a unique identifier(uid) and the producer data both of which are described in section 3.2. This function allows the Event Producer to define the kind of events that will be produced. A unique identifier must be provided to the *Eventhandler*, and a Response status will be returned by this function.

4.1.7.2.3 <u>unRegister</u>

This function is invoked to remove a consumer or a producer from the system. It requires only the uid of the entity to be unregistered. In the case of the Event Consumer, the system will not receive any events anymore. In the case of Event Producer, the *Eventhandler* will be informed that the Event Producer does not exist anymore, and can take more actions such as notifying Event Consumers related to the Event Producer that logged out.

4.1.7.2.4 query

The function `query()` is used to perform queries on the list of Event Producer and Event Consumers. The function returns the registered systems that satisfy the provided query.

4.1.7.2.5 queryAll

The function *queryAll()* is used to retrieve the full record regarding all entities registered in the *Eventhandler*.

4.1.7.3 Sequence Diagrams

4.1.7.3.1 registerEntity

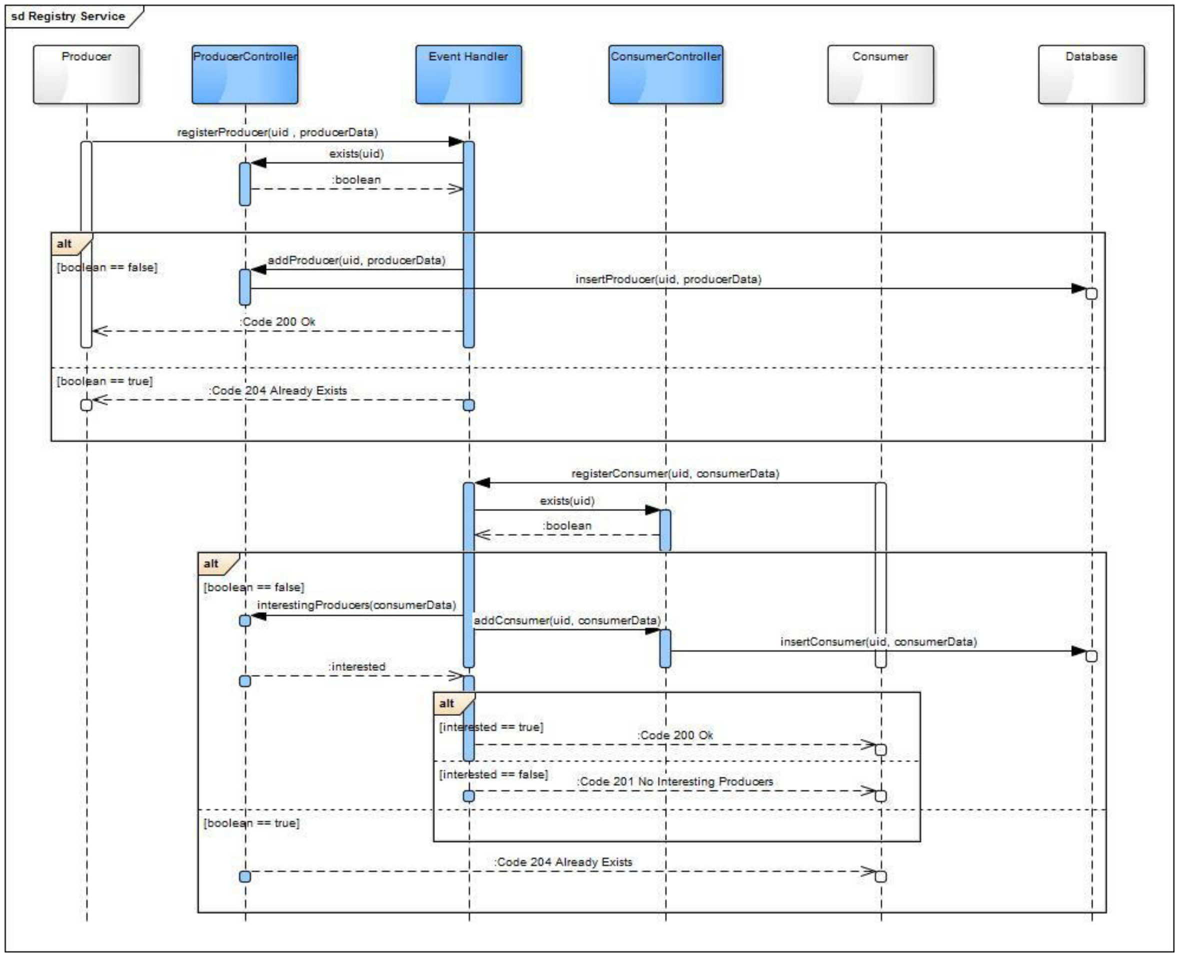


Figure 15 - Register Entity Sequence Diagram

An Event Consumer or Event Producer can register within the *Eventhandler* using the functions *registerConsumer()* and *registerProducer()* respectively, both require an UID and the proper data and both are represented in figure 8 as consumerData or producerData. These latter are wrappers for the consumer and the producer abstract data. One implementation of these data structures will be described in the Semantic profile of this service.

4.1.7.3.2 unRegisterEntity

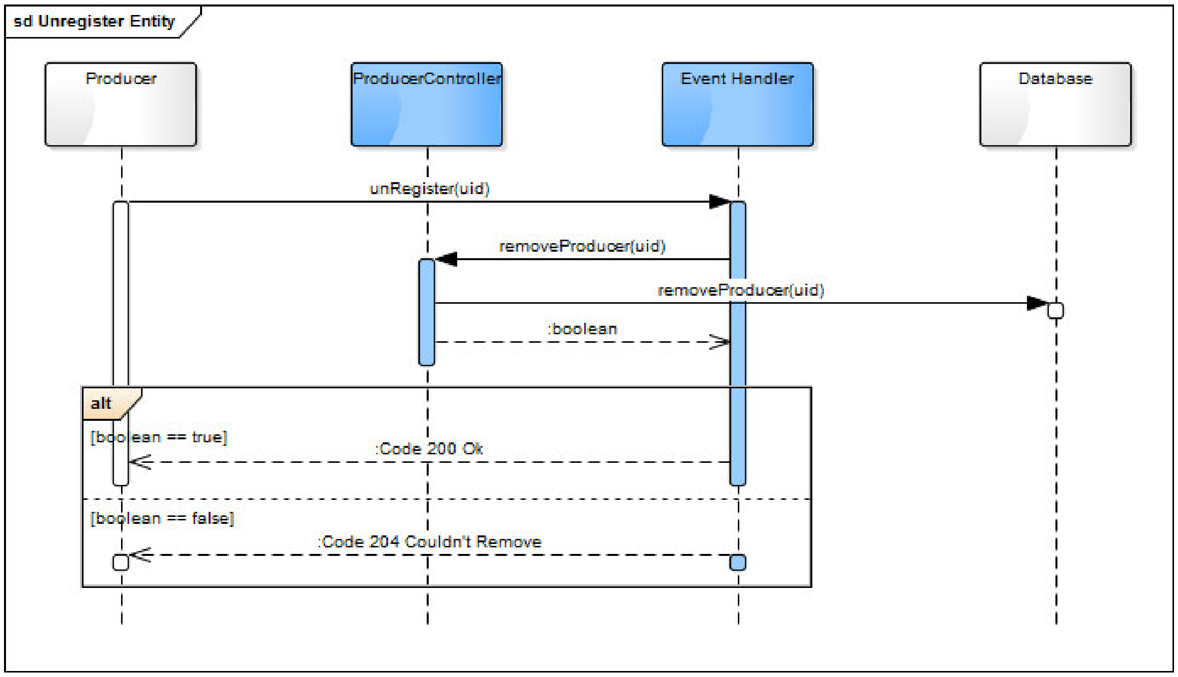


Figure 16 - unRegisterEntity Sequence Diagram

An Event Consumer or Event Producer can unregister and no longer receive or produce events. Using the *unRegisterEntity()* function, only the uid is required. A response code will be emitted after the request is complete according if the operation was successful or not. The data involved with this function are described the Registry Semantic Profile.

4.1.7.3.3 queryAll

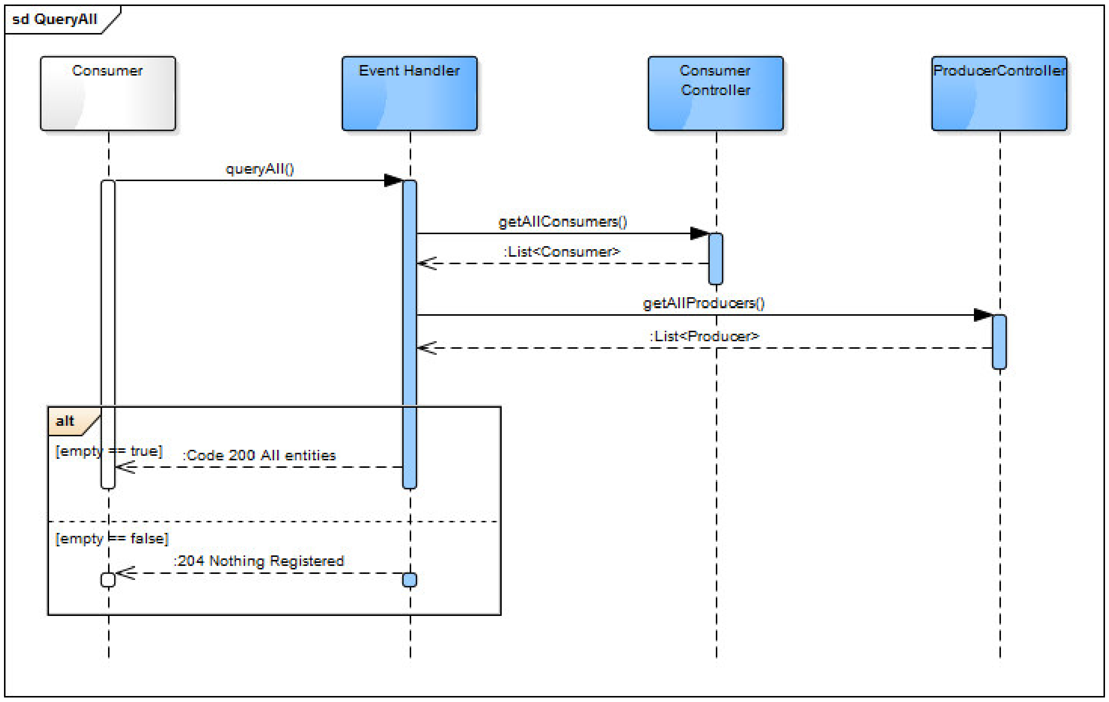


Figure 17 - Query All Registered Sequence Diagram

Using the *queryAll()* function, all data concerning registered producers and consumers is returned.

4.1.7.3.4 query

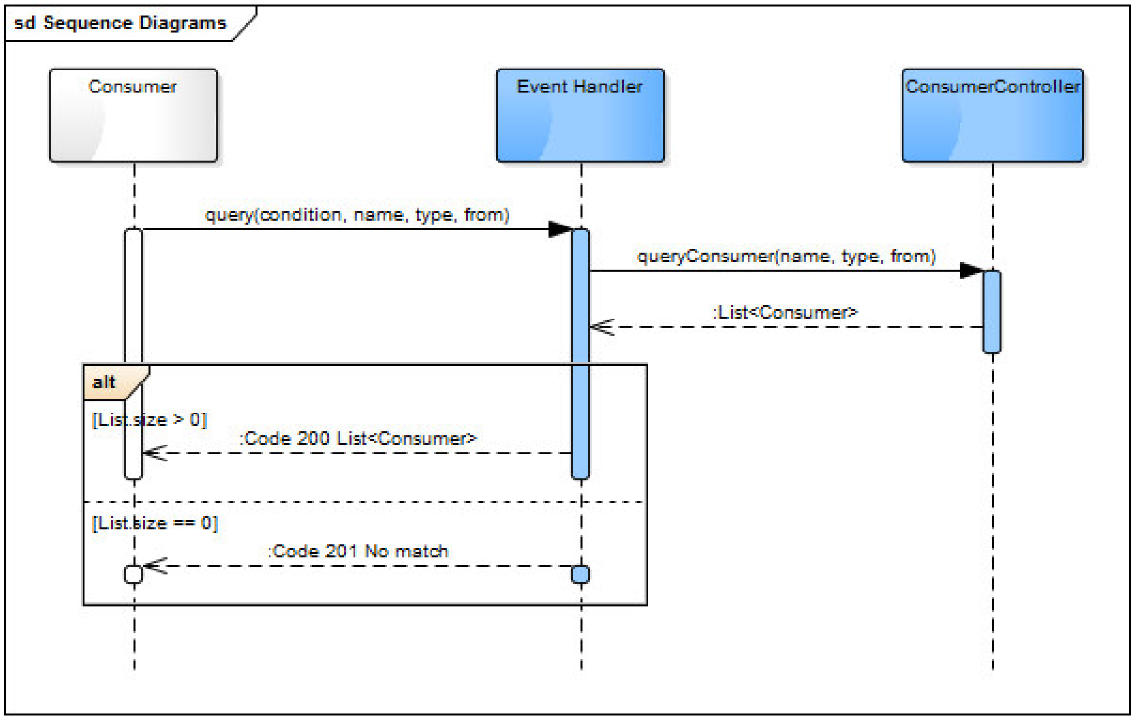


Figure 18 - Query Registered Sequence Diagram

A query containing logic variables and regular expressions query can be made against the *Eventhandler* Registry, e.g. to retrieve all producer with its name starting with Porto, which might correspond to all producers physically located on the city of Porto. Note that the capabilities of the query are defined by each implementation on the corresponding Semantic Profile for the implementation of this service. In some implementations, using the *query()* function an entity can retrieve more detailed information than with the *queryAll()* function.

4.1.7.4 Service Information Data

FIELD	DESCRIPTION
CONSUMERSLIST	It contains a reference to each registered Event Consumer
PRODUCERSLIST	It contains a reference to each registered Event Producer
FILTER	It is a filter, to be applied to incoming events before delivery to an Event Consumer

Table 15 - Registry Data Type Description

4.1.7.4.1 <u>**consumersList**</u>

`comsumerList` is a list of `consumer` data type.

`consumer` is an abstract data type describing an Event Consumer in the network, including the name of the entity, the kind of events that are being subscribed, and the filtering rules for the incoming events.

- *Name* is the name of the entity.
- *Uid* is an uniqued identifier that must be diferent for all entities.
- *Type* is the event service type that was subscribed.
- *NotifyUnregister* indicates that the Consumer wants to be notified each time a subscribed Producer is unregistered.
- *Filter* is the abstract data type describing filtering rules for incoming events.

4.1.7.4.2 <u>**producersList**</u>

`producerList` is a list of `producer` data type.

`producer` is an abstract data type describing an Event Producer in the network, including the name of the system and the kind of events that are going to be produced.

- *Name* is the unique name of the system.
- *Type* is the event service type that will be produced.

4.1.7.4.3 <u>**Filter**</u>

`filter` is the abstract data type describing filtering rules for incoming events. Concretization of this data type are present in the Semantic Profiles. Examples of filters are the following:

- *Severity* to be compared to the Severity meta-datum of the event.
- *A Start time;End time* and a *uid* of a specific Event Producer, to receive events generated in the specified period, and a set of Producers. Example: all events of February 29th, with the producer id starting with “Porto”.
- A *type* which represents the event type that the consumer is subscribing for.
- The *from* value is simply the uid of a Producer that the consumer wants to subscribe to. If null this element is simply ignored.

4.1.7.4.4 Query format

The *query* data is composed by several elements all of which can be *null* with the exception of the condition. When an element is *null* the query simply ignores this parameter.

- *Condition* is a boolean which specifies if we are searching for a *Consumer* or a *Producer*.
- *Name* is a String which will be used to search for an entity that will match this specific name.
- *Type* is the same as described in 4.1.7.4.3.
- *From* is also the same as described 4.1.7.4.3.

4.1.8 Publish Events Service Description

The purpose of the Publish Events service is to allow applications to send events to the *Eventhandler*.

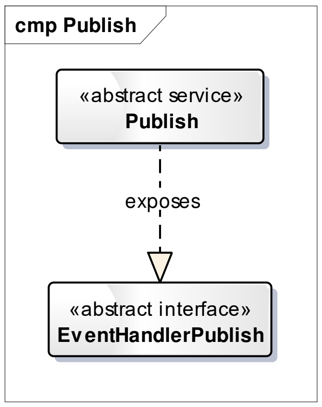


Figure 19 - Eventhandler Publish Events Overview

4.1.8.1 EventHandlerPublish

The only function of the EventHandlerPublishEventProvider interface is *publishEvent()*, which is invoked by Event Producers to send events to the *Eventhandler*.

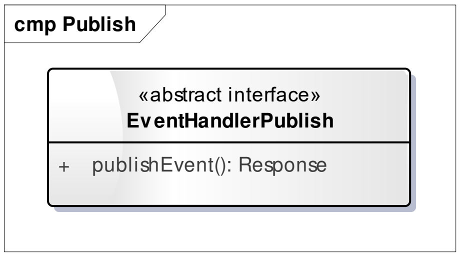


Figure 20 - Functions implemented by EventHandlerPublish interface

4.1.8.2 Functions

4.1.8.2.1 publishEvent

The function is invoked by Event Producers to send events to the *Eventhandler*. An event is sent to this function as a parameter.

4.1.8.3 Sequence Diagrams

4.1.8.3.1 Publish Event

The diagram in Figure 3 describes the sequence of events triggered by the publishEvent function, including the interaction with the subscribers and the storage systems, which can be a database (as depicted in the figure).

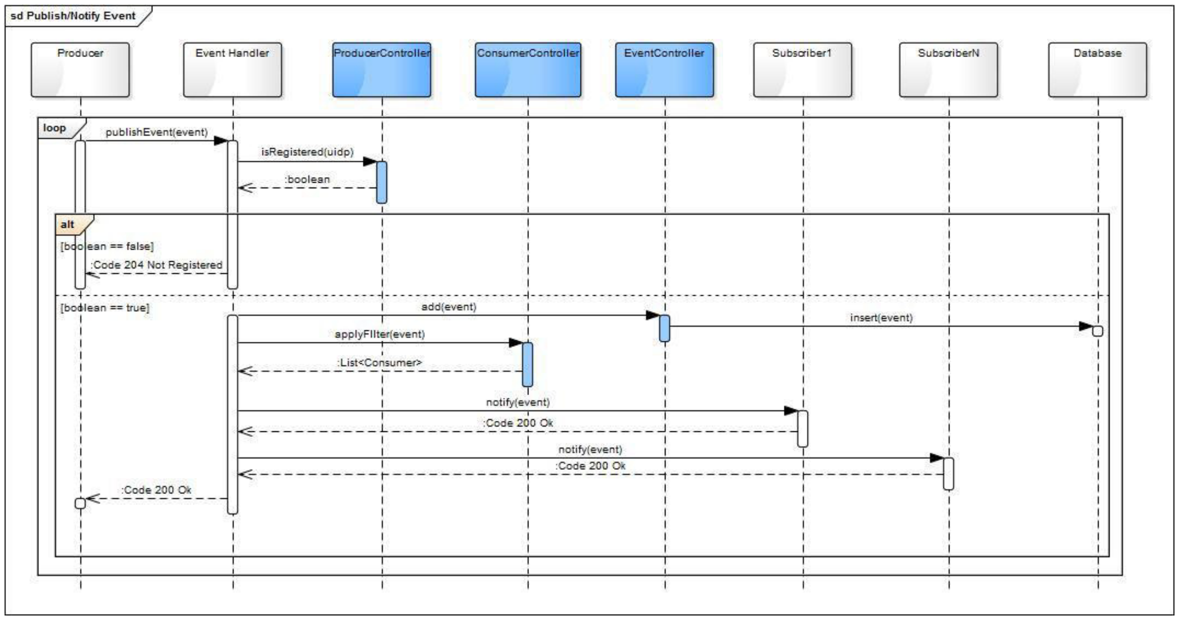


Figure 21 - Publish Event Sequence Diagram

When an interesting event is triggered in an Event Producer the function `publishEvent()` is used to send all necessary data to the *Eventhandler*. When an event is received, the *Eventhandler* must firstly heck if the sending producer is registered, if it not registered than it has no permission to use this function and a response code 204 is returned. If the producer permitted to send information, than a list of interested subscribers must be created in order to send the information accordingly, this list is obtained using the *applyFilter()* function which return a list of interested Consumers in receiving the event. Finally, all the subscribers are notified and the procedure is logged in a database, a file or both and a response with the Code 200 is sent to the Producer.

4.1.8.4 Service Information Data

FIELD	DESCRIPTION
EVENT	It contains all data associated with the event received

Table 16 - Publish Events Data Type Description

4.1.8.4.1 Event

`Event` is an abstract data type that is described as follows:

- *from*: Indicates who created the event, e.g. the producer id
- *type*: The code of the event e.g., a number that defines the type of the event
- *description*: Provides metadata that describes the event, and it is used by the filtering rules on the *Eventhandler*. An example of metadata regards the severity level of the event
- *payload*: Payload of the event

4.1.9 GetHistoricalData Service Description

This section describes the *Eventhandler* GetHistoricalData service. This service applies filtering rules to permanent stored events (e.g. in a Database, log file or on the Historian) and returns data containing information regarding events.

4.1.9.1 Overview

The purpose of the Get Historical Data service is to allow to retrieve a set of events stored by the *Eventhandler* in a Database, log file or on the Historian.

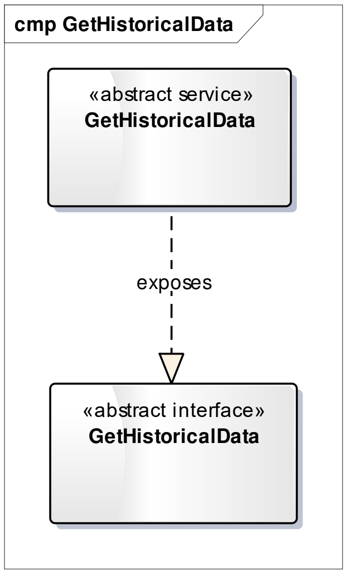


Figure 22 - Get Historical Data Overview

4.1.9.2 Abstract Interfaces

The *Eventhandler* Get Historical Data service exposes the GetHistoricalData interface, which is used to query historical data regarding events. Filtering rules are applied in order to get information regarding a specific event or list of events.

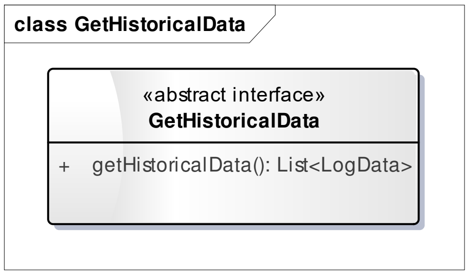


Figure 23 - Function implemented by GetHistoricalData interface

The only function of the GetHistoricalData interface is *getHistoricalData().*

4.1.9.3 Functions

4.1.9.3.1 GetHistoricalData

This function is used to query the historical data on past events. The parameter to be passed to the function has got the semantics of a filter.

This method is responsible for retrieving all events within the date range provided and according to other filter elements, like for example, the severity level and the event type. If the start date and end date parameters are null, all events logged will be retrieved. The returned data structure contains the list of events that matched the filter, and the list of subscribers that had received each event when it was produced.

4.1.9.4 Sequence Diagrams

4.1.9.4.1 getHistoricalData

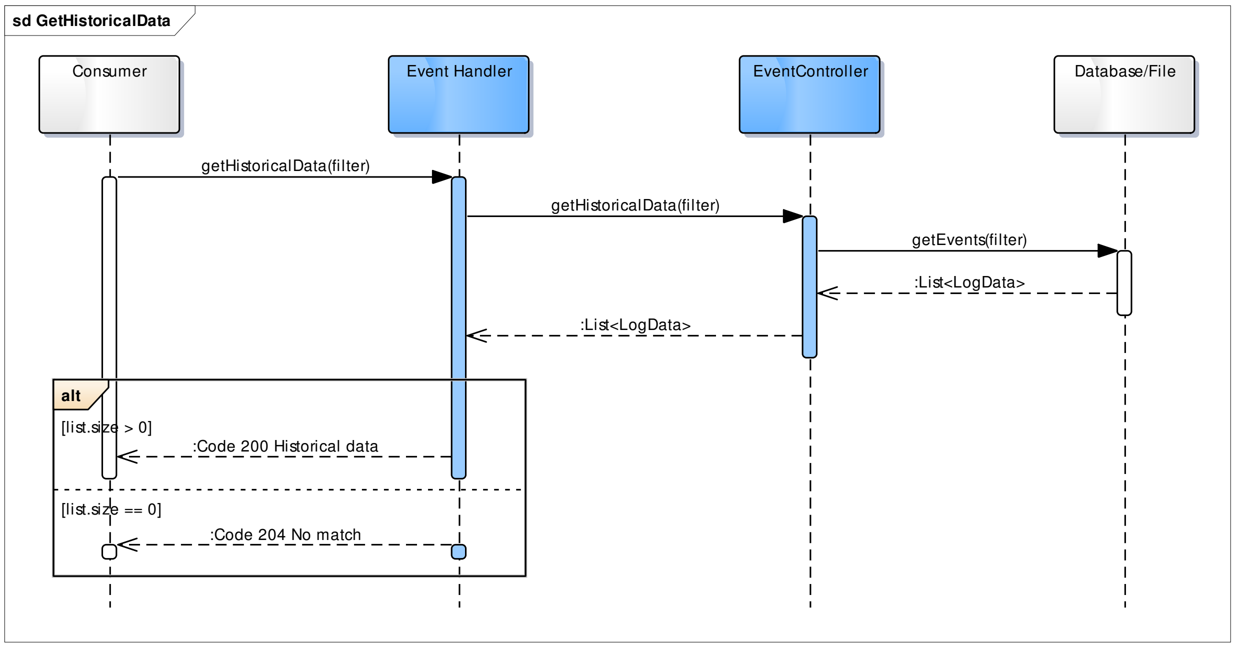


Figure 24 - GetHistorical Data Sequence Diagram

4.1.9.5 Abstract Information Model

FIELD	DESCRIPTION
FILTER	Contains rules to decide which events to retrieve
LOGDATA	An object containing an event and the list of Consumers notified.

Table 17 - GetHistoricalData Data type description

4.1.9.5.1 LogData

LogData contains an `event` data type and a list of `consumers` that received that `event`. All information regarding this data type is exposed in the publish service description, check Section 4.1.8.

4.1.9.5.2 Filter

All information regarding this data type is exposed in the registry service description, check Section 4.1.7.

4.1.10 Notify Service Description

This section describes the Notify Events service, which must be exposed by any system interested into subscribing against the *Eventhandler*.

4.1.10.1 Overview

The purpose of the Notify Events service is the delivery of events to Event Consumers. The *Eventhandler* accesses the Notify Events service on each subscriber for each event it needs to deliver.

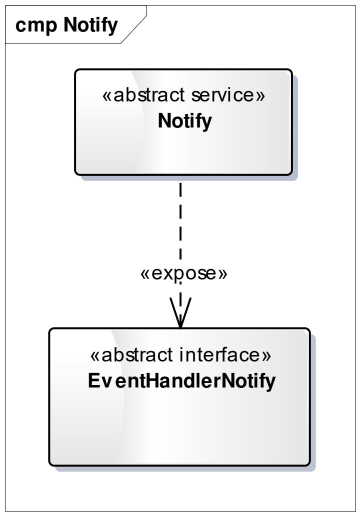


Figure 25 - Notify Events Overvie

4.1.10.2 Abstract Interfaces

The *Eventhandler* Notify Event service exposes the EventHandlerNotify interface, which must be implemented by systems that register themselves as Event Consumers against the *Eventhandler*.

4.1.10.3 EventHandlerNotify

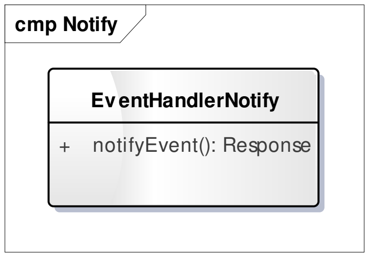


Figure 26 - Function implemented by EventHandlerNotify interface

The only function of the EventHandlerNotify interface is *NotifyEvent()*, which is invoked by the *Eventhandler* to send events to Event Consumers.

4.1.10.4 Functions

4.1.10.4.1 NotifyEvent

Any Event Consumer must provide the *NotifyEvents()* function. After an event is published, using the Publish service, interested Event Consumers are notified using this function.

4.1.10.5 Sequence Diagram

This diagram is already exposed in the Publish Service Description, Section 4.1.8.

4.1.10.6 Service Information Data

FIELD	DESCRIPTION
EVENT	Contains all data associated with the event received
FILTER	Contains the filtering rules applied by the *Eventhandler* in order to get a list of Event Consumers to notify

Table 18 - Notify Data Type Description

Both the *Event* and *Filter* data types are exposed in Sections 4.1.5 and 4.1.4 respectively.

4.1.11 Sematic Profile

As mentioned in Section 1.2.2 the Sematic Profile will offer a description of the data format and what is the type of the encoding, in this case XML and JSON. In order to understand this concept, the only information required is the order in which the data is presented. Firstly, the name of the class is presented, followed by the UML diagram. Finally, an example of a XML and JSON encoding are presented.

4.1.11.1 Consumer

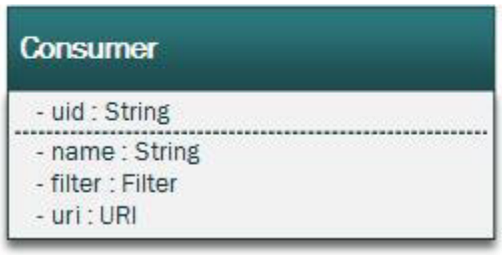


Figure 27 - Consumer Class

XML

```
<consumer>
       <uid>porto-sub-1</uid>
       <name>cister-subscriber1</name>
       <uri> http://192.168.50.125:8081/subcriberUIDeXample</uri>
       <filter>
              <description severity="2"/>
              <startDateTime>2015-08-01T12:00:30.125Z</startDateTime>
              <endDateTime>2015-09-01T12:00:30.125Z</endDateTime>
              <type>temperature</type>
              <from>porto-sensor-10</from>
       </filter>
</consumer>
```

JSON

```
{"consumer":[
       {"uid":"subcriberUIDeXample",
       "name":"cister-subscriber1",
       "uri":"http://192.168.50.125:8081/subcriberUIDeXample",
       "filter":{
              "uid":"filter_uid",
              "description":{"severity":4},
              "startDateTime":null,
              "endDateTime":null,
              "type":"temperature",
              "from":"sensor_134"
```

```
        }
}
```

4.1.11.2 <u>Producer</u>

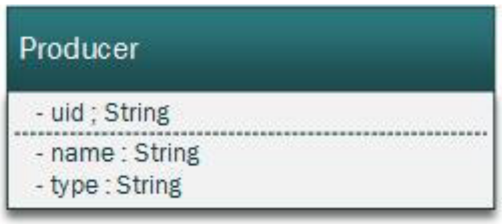


Figure 28 - Producer Class

XML

```
   <producer>
      <uid>porto-sensor-10</uid>
      <name>Sensor 10</name>
      <type>temperature</type>
   </producer>
```

JSON

```
"producer":[{
       "uid":"sensor_134",
"name":"cister-sensor1",
"type":"temperature"}]
}
```

4.1.11.3 <u>Registered</u>

This class is created only to simply the presentation of all the entities registered is the system. It simply contains a list of *Consumers* and *Producers*.

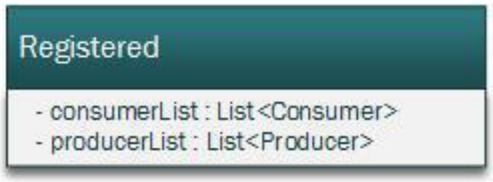

Registered
- consumerList : List<Consumer>
- producerList : List<Producer>

Figure 29 - Registered Class

XML

```
<Registered>
   <consumer>
```

```
      <uid>porto-sub-1</uid>
      <filter>
         <description severity="2"/>
         <startDateTime>2015-08          01T12:00:30.125Z</startDateTime>
         <endDateTime>2015-09-01T12:00:30.125Z
         </endDateTime>
         <type>temperature</type>
         <from>porto-sensor-10</from>
      </filter>
   </consumer>
   <producer>
      <uid>porto-sensor-10</uid>
      <name>Sensor 10</name>
      <type>temperature</type>
   </producer>
</Registered>
```

JSON

```
{
       "consumer": [{
              "uid": "subcriberUIDeXample",
              "name": "cistersubscriber1",
              "uri": "http://192.168.50.125:8081/subcriberUIDeXample",
              "filter": {
                     "uid": "filter_uid",
                     "description": {
                            "severity": 4
                     },
                     "startDateTime": null,
                     "endDateTime": null,
                     "type": "temperature",
                     "from": "sensor_134"
              }
       }],
       "producer": [{
              "uid": "sensor_134",
              "name": "cister-sensor1",
              "type": "temperature"
       }]
}
```

4.1.11.4 Filter

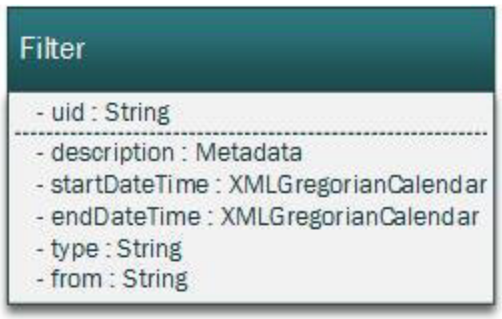

Filter
- uid : String
- description : Metadata
- startDateTime : XMLGregorianCalendar
- endDateTime : XMLGregorianCalendar
- type : String
- from : String

Figure 30 - Filter class

XML

```
<Filter>
	<startDateTime>2016-07-26 23:34:18</startDateTime>
	<endDateTime>2016-07-27 00:00:00</endDateTime>
	<from>sensor_124</from>
	<type>temperature</type>
	<description>
		<severity>1</severity>
	</description>
</Filter>
```

JSON

```
{
	"Filter": {
		"startDateTime": "2016-07-26 23:34:18",
		"endDateTime": "2016-07-27 00:00:00",
		"from": "sensor_124",
		"type": "temperature",
		"description": {
			"severity": "1"
		}
	}
}
```

4.1.11.5 Event

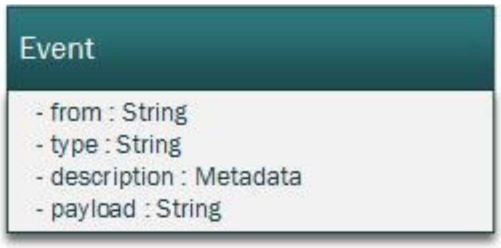


Figure 31 - Event Class

XML

```
<Event>
       <from>porto-sensor-10</from>
       <type>temperature</type>
       <description>
              <severity>5</severity>
       </description>
       <payload>10</payload>
</Event>
```

JSON

```
{
       "Event": {
              "from": "porto-sensor-10",
              "type": "temperature",
              "description": {
                     "severity": "5"
              },
              "payload": "10"
       }
}
```

4.1.11.6 Metadata

Currently the metadata is only suporting a severity attribute but it can be added any other argument depending on the application usage.

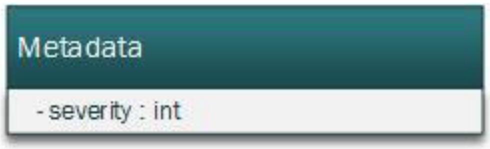


Figure 32 - Metadata Class

XML

```
<Metadata>
	<severity>5</severity>
</Metadata>
```

JSON

```
{
	"Metadata": {
		"severity": "5"
	}
}
```

4.1.11.7 LogData

The *LogData* object contains all information related to an event received in the past. This data type is composed by an *Event* data type which was described previously, and the list of consumers that we're successfully notified.

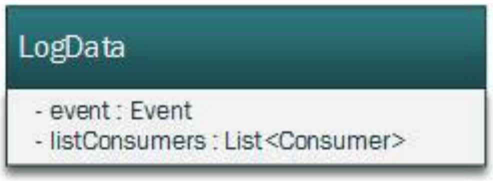


Figure 33 - LogData Class

XML

```
<LogData>
	<Event>
		<from>porto-sensor-10</from>
		<type>temperature</type>
		<description>
			<severity>5</severity>
		</description>
		<payload>10</payload>
	</Event>
	<consumerList>
		<consumer>
			<uid>porto-sub-1</uid>
			<filter>
				<description severity="2"/>
				<startDateTime>2015-0801T12:00:30.125Z</startDateTime>
				<endDateTime>2015-09-01T12:00:30.125Z</endDateTime>
				<type>temperature</type>
				<from>porto-sensor-10</from>
			</filter>
		</consumer>
	</consumerList>
</LogData>
```

JSON

```
{
	"LogData": {
		"Event": {
			"from": "sensor_1234",
			"type": "pressure",
			"description": {
				"severity": "4"
			},
			"payload": "20"
		},
		"consumerList": {
			"Consumer": {
				"uid": "subcriberUIDeXample",
				"name": "cister-subscriber1",
				"uri": "http://192.168.50.125:8081/subcriberUIDeXample",
				"filter": {
					"uid": "filter_uid",
					"description": {
						"severity": 4
					},
					"startDateTime": null,
					"endDateTime": null,
					"type": "temperature",
					"from": "sensor_134"
				}
			}
		}
```

```
        }
}
```

4.1.12 Interface Design Descriptions

4.1.12.1 Registry Rest Interface

FUNCTION	SERVICE	METHOD	INPUT	OUTPUT
REGISTERCONSUMER	/subscriber/UID	POST	Consumer	Response
REGISTERPRODUCER	/producer/UID	POST	Producer	Response
QUERY	-	GET	Filter	Registered
QUERYALL	/UID	GET	-	Registered
UNREGISTERENTITY	/UID	DELETE	-	Response

Table 19 - Registry Rest Interfaces

4.1.12.2 Publish Events Rest Interface

FUNCTION	SERVICE	METHOD	INPUT	OUTPUT
PUBLISHEVENTS	/publish	POST	Event	Response

Table 20 - Publish Events Rest Interface

4.1.12.3 GetHistoricalData Rest Interface

FUNCTION	SERVICE	METHOD	INPUT	OUTPUT
GETHISTORICALS	/historicals	POST	Filter	List<LogData>

Table 21 - GetHistoricalData Rest Interface

4.1.12.4 Notify Rest Interface

FUNCTION	SERVICE	METHOD	INPUT	OUTPUT
NOTIFYEVENTS	/notify	POST	Event	Response

Table 22 - Notify Rest Interface

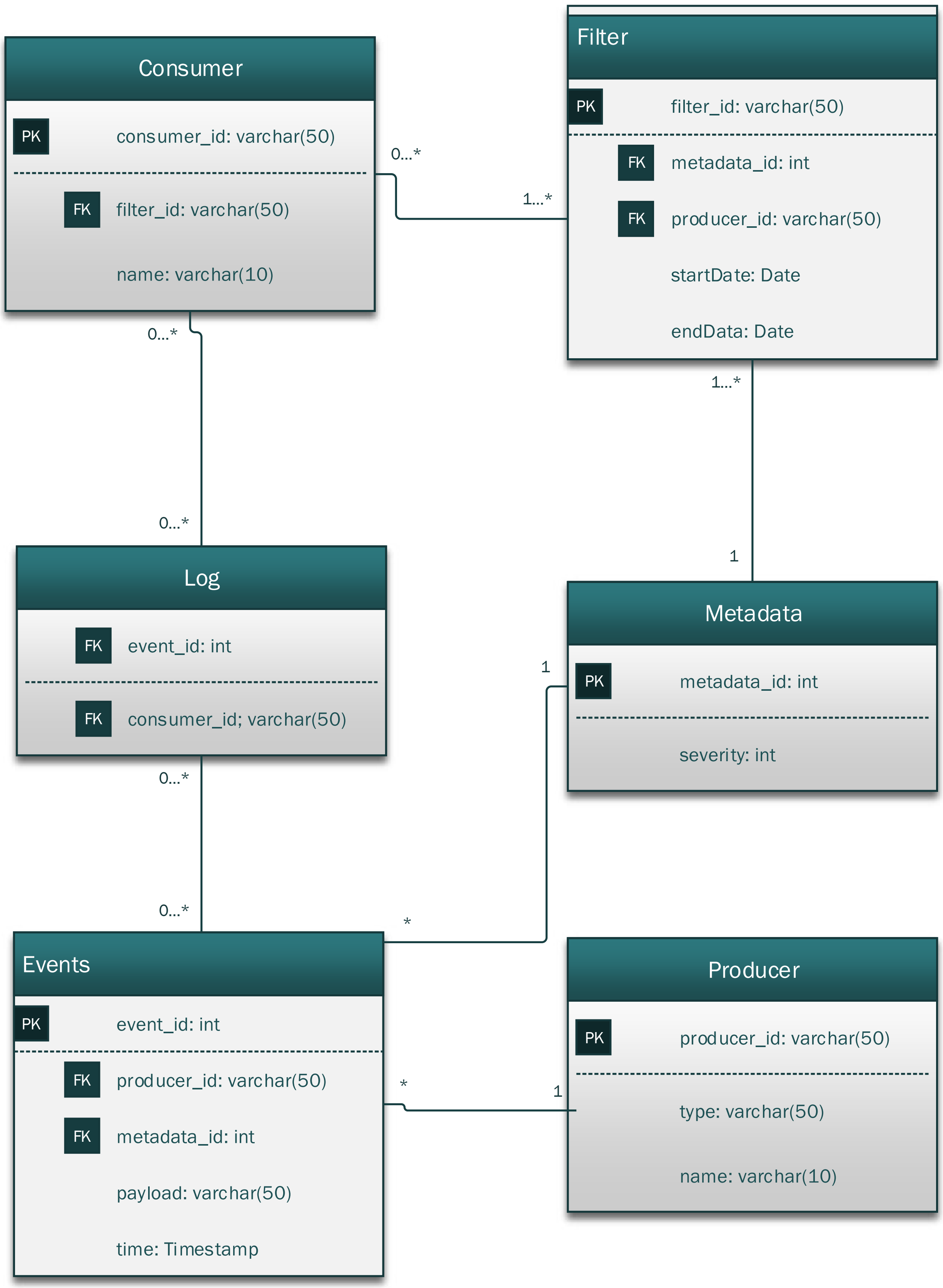

Consumer
PK consumer_id: varchar(50)
FK filter_id: varchar(50)
name: varchar(10)
Filter
PK filter_id: varchar(50)
FK metadata_id: int
FK producer_id: varchar(50)
startDate: Date
endData: Date
Log
FK event_id: int
FK consumer_id; varchar(50)
Metadata
PK metadata_id: int
severity: int
Events
PK event_id: int
FK producer_id: varchar(50)
FK metadata_id: int
payload: varchar(50)
time: Timestamp
Producer
PK producer_id: varchar(50)
type: varchar(50)
name: varchar(10)
0...*
1...*
0...*
0...*
1...*
1
1
0...*
0...*
*
*
1

Figure 34 - Domain Model

Referring Figure 3 the domain model can be described as follows:

A *Consumer* is composed by an *uid(unique identifier),* a *name* and contains a foreign key to a *filter_*id. This filter is composed by a *startDate*, and an *endDate* both of *Date* data type and contains to foreign keys, one referring to a *Producer* and the other referring to a *Metadata* which is currently simply composed by a *severity* level only and will be enhanced or altered according to the implementation requirements.

The *Producer* composition is similar to a *Consumer,* containing a *uid*, *name* and a *type*. A *Producer* creates *Events* data which are composed by an auto incremented *event_id*, a payload and the timestamp. In order to get the information regarding the production and severity of this *Event* data we use a foreign key to the *producer_id* and to the *metadata_id*.

Notes:

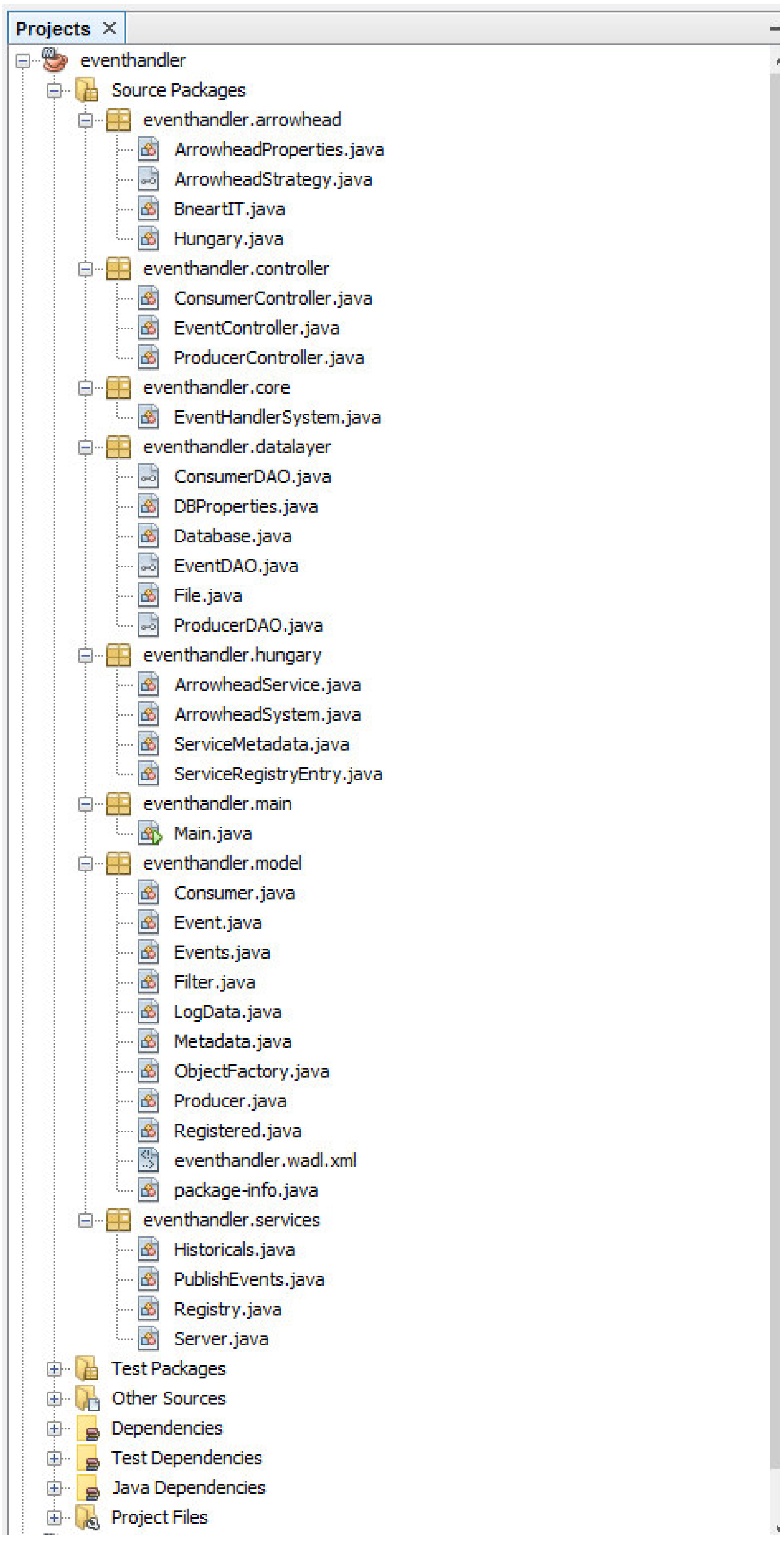
In order to properly retrieve all the information regarding the income of an *event,* a table name *Log* was created. Using for example *join* operations the application can easily write a log message.

4.2 Development of The Solution

Now that the project analysis is complete we proceed to the development phase. In this section the system architecture is presented, which includes details about the projects tree in Netbeans and how the main features were implemented and how are they related. To conclude, after all the system's functionalities are explained a final section is presented providing functional, stress tests, among others.

4.2.1 System Architecture

To begin explaining how the system is structured and organized a visual representation of the project file tree is displayed in Figure 35. Since Maven is used as the building and dependency management system, the packages used are the standard for all Maven projects. The *Source Packages* contains all the code developed while the *Test Packages* contains the unit tests for the core services. Finally, the *Project Files* contains only the *pom.xml* file where the Maven features are implemented.



```
Projects ×
eventhandler
  Source Packages
    eventhandler.arrowhead
      ArrowheadProperties.java
      ArrowheadStrategy.java
      BneartIT.java
      Hungary.java
    eventhandler.controller
      ConsumerController.java
      EventController.java
      ProducerController.java
    eventhandler.core
      EventHandlerSystem.java
    eventhandler.datalayer
      ConsumerDAO.java
      DBProperties.java
      Database.java
      EventDAO.java
      File.java
      ProducerDAO.java
    eventhandler.hungary
      ArrowheadService.java
      ArrowheadSystem.java
      ServiceMetadata.java
      ServiceRegistryEntry.java
    eventhandler.main
      Main.java
    eventhandler.model
      Consumer.java
      Event.java
      Events.java
      Filter.java
      LogData.java
      Metadata.java
      ObjectFactory.java
      Producer.java
      Registered.java
      eventhandler.wadl.xml
      package-info.java
    eventhandler.services
      Historicals.java
      PublishEvents.java
      Registry.java
      Server.java
  Test Packages
  Other Sources
  Dependencies
  Test Dependencies
  Java Dependencies
  Project Files
```

Figure 35 - Netbeans Project Tree

The *eventhandler.arrowhead* package contains all the classes related to the usage of the *Strategy Pattern*. The *ArrowheadStrategy* interface provides a single function called *register()* which allows the application to dynamically adapt to either the BnearIT approach or the Hungary approach.

Using the *Controller Pattern,* it is possible to provide with a much more robust application. In this case we simply need three controllers, one for each of the handled model classes, *Producers, Consumers* and *Event.*

In this stage and since the Hungary Arrowhead approach is still being configured and changed, instead of using a *jar* file like with the BnearIT approach we need to have the core classes in our application. These classes are inserted in the *eventhandler.hungary* package.

As the name states the *eventhandler.main* package simply contains the project's main class.

All the model classes are inserted in the *eventhandler.model* package. This provides with a cleaner view and whenever a new model class needs to be created we can simply add it here.

Concerning the data layer classes, I opted to use the *Data Access Object Pattern.* The package containing this logic is the *eventhandler.datalayer* package.

Finally, all the *REST* services and the *Jetty* configuration classes are in the *eventhandler.services* package.

4.2.2 Features

In this subsection the code developed to create the three *Eventhandler* core services is described, along with the *Notify* service of the subscribers. The database/file functions are also explained since they are crucial to maintain a good performance of the application.

4.2.2.1 **Registry Service**

The Registry service includes four core features: register an Event Consumer/Producer, unregister entity, query and query all. In this chapter the code developed to provide the application with such features is represented and explained.

In order to have a better picture of all the features, the *Registry* part of the *application.wadl* file is described in Figure 36.

```
<resource path="registry">
            <resource path="/{uid}">
                <param xmlns:xs="http://www.w3.org/2001/XMLSchema" name="uid"
style="template" type="xs:string"/>
                <method id="unRegister" name="DELETE">
                    <response/>
                </method>

          </resource>

             <resource path="producer/{uid}">
                <param xmlns:xs="http://www.w3.org/2001/XMLSchema" name="uid"
style="template" type="xs:string"/>
                <method id="registerProducer" name="POST">
                    <request>
                        <representation
xmlns:ns2="http://www.arrowhead.org/eventhandler/registered" element="ns2:producer"
mediaType="application/json"/>
                    </request>
                    <response/>

             </resource>

             <resource path="queryAll">

                <method id="queryAll" name="GET">
                    <response>
                        <representation mediaType="application/json"/>
                    </response>
                </method>

             </resource>

             <resource path="query">
                <method id="query" name="GET">
                    <request>
                        <param xmlns:xs="http://www.w3.org/2001/XMLSchema"
name="condition" style="query" type="xs:boolean" default="false"/>
                        <param xmlns:xs="http://www.w3.org/2001/XMLSchema" name="name"
style="query" type="xs:string" default=""/>
                        <param xmlns:xs="http://www.w3.org/2001/XMLSchema" name="type"
style="query" type="xs:string" default=""/>
                        <param xmlns:xs="http://www.w3.org/2001/XMLSchema" name="from"
style="query" type="xs:string" default=""/>
                    </request>
                    <response>
                        <representation mediaType="application/json"/>
                    </response>
                </method>

             </resource>

             <resource path="subscriber/{uid}">
                <param xmlns:xs="http://www.w3.org/2001/XMLSchema" name="uid"
style="template" type="xs:string"/>
                <method id="registerConsumer" name="POST">
                    <request>
                        <representation
xmlns:ns2="http://www.arrowhead.org/eventhandler/registered" element="ns2:consumer"
mediaType="application/json"/>
                    </request>
                    <response/>
                </method>

             </resource>
```

Figure 36 - Registry WADL File

4.2.2.2 Register Consumer

Event Consumers are able to use the Registry service to register themselves with the *Eventhandler*, using the function *registerConsumer* which is implemented as described in Figure 37.

```
/**
     *
     * REST interface used by consumers to register in the Eventhandler
system.
     * <p>
     * In case the UID is not yet registed and there is a Producer which
matches
     * the Consumer filter response 200 is returned In case the UID is not
yet
     * registed and there isnt a Producer which matches the Consumer filter
     * response 201 is returned In case the UID is already registed response
204
     * is returned</p>
     *
     * @param uid Consumer uid.
     * @param c Consumer object.
     * @return A Response code 200 if registered successfully. Code 201 if
     * registered successfully and there is no event production for this
     * subscriber. Code 204 if the Consumer uid already exists.
     */
    @POST
    @Path("subscriber/{uid}")
    @Consumes(MediaType.APPLICATION_JSON)
    public Response registerConsumer(@PathParam("uid") String uid, Consumer
c) {

        if (ehs.consumer_controller.getConsumer(uid) == null) {

            ehs.consumer_controller.addConsumer(c, ehs.getDatabase());
            if (ehs.producer_controller.interestingProducers(c)) {
                return Response.status(200).entity("Created subscriber with
UID: " + uid).build();
            } else {
                return Response.status(201).entity("Created subscriber with
UID: " + uid + "\nNo producers exist"
                        + " for the event type " +
c.getFilter().getType()).build();
            }
        } else {
            return Response.status(204).entity("UID: " + uid + "already
exists!!").build();
        }
    }
```

Figure 37 - Register Entity Code

After receiving the *uid* and *ConsumerType* through the *registerConsumer* function the *Eventhandler* must do some verifications before adding the Event Consumer to its database. Firstly, it verifies if an Event Consumer already exists for the provided *uid* using the *getConsumer* function, if it does not exist it can create a new one using the *addConsumer* function. Using the *interestingConsumers* function of the *ProducerController* class the *Eventhandler* can return a different response, if there aren't any interesting producers. The different returned codes are explained in Figure 37 in the Javadoc section. Depending of the returned code, Event Consumer can deal accordingly.

4.2.2.3 Register Producer

Event Producers are also able to use the Registry service to register themselves with the *Eventhandler*, using the function *registerProducer,* Figure 38.

```
/**
     * REST interface used by producers to register in the Eventhandler
system.
     *
     * In case the UID is not yet registered a response 200 is returned. In
case
     * the UID is already registered a response 204 is returned
     *
     * @param uid Producer uid.
     * @param p Producer object.
     * @return A Response code 201 if registered successfully. Code 204 if
the
     * Producer uid already exists.
     */
    @POST
    @Path("producer/{uid}")
    @Consumes(MediaType.APPLICATION_JSON)
    public Response registerProducer(@PathParam("uid") String uid, Producer
p) {

        if (ehs.producer_controller.getProducer(uid) == null) {
            ehs.producer_controller.addProducer(p, ehs.getDatabase());
            return Response.status(200).entity("created " + uid).build();
        } else {
            return Response.status(204).entity("UID: " + uid + "already
exists!!").build();
        }
    }
```

Figure 38 - Register Producer Code

This *registerProducer* method is very similar to the *registerConsumer* method described previously. The main difference being not having to check for any other property than the *uid*.

4.2.2.4 Unregister Entity

All entities must be able to unregister from the *Eventhandler* at any time. This requirement is provided through the *unregisterEntity* function of the *Registry* service, Figure 39.

```
/**
     * REST interface which allows all entities to be unregistered.
     *
     * @param uid Producer/Consumer uid.
     * @return A Response code 200 if removed a Producer/Consumer. Code 204
if
     * no entity has the specified uid.
     */
    @DELETE
    @Path("/{uid}")
    @Consumes(MediaType.APPLICATION_JSON)
    public Response unRegister(@PathParam("uid") String uid) {

        if (ehs.producer_controller.removeProducer(uid, ehs.getDatabase())) {
            return Response.status(200).entity("removed Producer " +
uid).build();
        } else if (ehs.consumer_controller.removeConsumer(uid,
ehs.getDatabase())) {
            return Response.status(200).entity("removed Consumer " +
uid).build();
        } else {
            return Response.status(204).entity("could not remove " +
uid).build();
        }
    }
```

Figure 39 - Unregister Entity Code

Using the entities *uid* the *removeConsumer/ removeProducer* function is called. Through the Javadoc the returned codes can be understood.

4.2.2.5 <u>Query Registered</u>

Using the *queryRegistered* function applications are able to ask the *Eventhandler* for specific entities (Event Consumer/Producers) depending on the criteria specified in the code in Figure 40.

```
    /**
     * REST interface used to query the Eventhandler subscribers/producers
that
     * match the specifiedd parameters. If condition = true -&gt; Event
Producer
     * | condition = false -&gt; Event.
     *
     * @param condition Query condition. True means a producer query. False
     * means a consumer query.
     * @param q_name Query name.
     * @param q_type Query type.
     * @param q_from Query producer id.
     * @return A Response code 200 if the query matches any object. Code 201
if
     * no entity matches the specified criteria.
     * @throws JAXBException A root Exception for all JAXBExceptions.
     */
    @GET
    @Path("query")
    @Produces(MediaType.APPLICATION_JSON)
    public Response query(@DefaultValue("false") @QueryParam("condition")
boolean condition,
            @DefaultValue("") @QueryParam("name") String q_name,
@DefaultValue("") @QueryParam("type") String q_type,
            @DefaultValue("") @QueryParam("from") String q_from) throws
JAXBException {

        ArrayList<Producer> producer_list = new ArrayList<>();
        ArrayList<Consumer> consumer_list = new ArrayList<>();
        Registered r = ehs.m_registered;

        if (condition) {
            producer_list = ehs.producer_controller.queryProducer(q_name,
q_type);
            r.setProducer(producer_list);
        } else {
            consumer_list = ehs.consumer_controller.queryConsumer(q_name,
q_type, q_from);
            r.setConsumer(consumer_list);
        }

        if (r.getConsumer().size() > 0 || r.getProducer().size() > 0) {
            return
Response.status(200).type(MediaType.APPLICATION_JSON).entity(r).build();
        } else {
            return Response.status(201).type(MediaType.TEXT_PLAIN).entity("No
entity matches the specified criteria\n")
                    .build();
        }
    }
```

Figure 40 - Query Registered Code

4.2.2.6 Query All Registered

Similar to the *queryRegistered* feature the objective of this function is to query the *Eventhandler* for registered entities, the difference being, with the *queryAllRegistered* function all registered entities are returned.

```
/**
       * REST interface used to get all entities registered in the
Eventhandler.
       *
       * @return A Response code 200 if the query wass successful. Or returns
code
       *         201 if no entity matches the specified criteria.
       * @throws JAXBException
       *             A root Exception for all JAXBExceptions.
       */
      @GET
      @Path("queryAll")
      @Produces(MediaType.APPLICATION_JSON)
      public Response queryAll() throws JAXBException {
            if (ehs.consumer_controller.getAllConsumers().size() > 0
                        || ehs.producer_controller.getAllProducers().size() >
0) {
                  Registered r = new Registered();
                  r.setConsumer(new
ArrayList<>(ehs.consumer_controller.getAllConsumers()));
                  r.setProducer(new
ArrayList<>(ehs.producer_controller.getAllProducers()));
                  return
Response.status(200).type(MediaType.APPLICATION_JSON).entity(r).build();
            } else {
                  return
Response.status(204).type(MediaType.TEXT_PLAIN).entity("There are currently
no entities registered!")
                              .build();
            }
      }
```

Figure 41 - Query All Registered Code

4.2.2.7 Publish Events Service

The Publish Events service has only one core feature, the *publishEvents* feature.

This service's *WADL* is described in Figure 42.

```
<resource path="publish">
            <resource path="/{uid}">
                <method id="publishEvents" name="POST">
                    <request>
                        <representation
xmlns:ns2="http://www.arrowhead.org/eventhandler/registered"
element="ns2:event" mediaType="application/json"/>
                    </request>
                    <response/>
                </method>

</resource>
```

Figure 42 - Publish Evens WADL

4.2.2.8 Publish Events

Since the great purpose of the *Eventhandler* is to serve as a man in the middle between Event Consumer and Event Producers it must be able to receive, treat and forward events. This functionality is provided by this feature and its implementation is described in Figure 43.

```
/**
     * REST interface used by producers to publish events.
     *
     * @param event Event object.
     * @return A Response with code 200 if the event is published
successfully.
     * Code 204 if the producer is not registered.
     */
    @POST
    @Path("/{uid}")
    @Consumes(MediaType.APPLICATION_JSON)
    public Response publishEvents(Event event) {

        EventHandlerSystem ehs = EventHandlerSystem.getInstance();

        if (ehs.producer_controller.existsProducer(event.getFrom())) {
            ehs.notifyEvent(event);
            return Response.status(200).entity("Events Posted!").build();
        } else {
            return Response.status(204).entity("This event producer is not
registered!").build();
        }

    }
```

Figure 43 - PublishEvents Code

As displayed is the figure, the *publishEvents* function receives an *Event* data type and an *uid* as a path variable. Firstly, it verifies if the *Producer* using this service is allowed to send events. To do so the *Eventhandler* simply verifies if the *Producer is* registered using the *existsProducer* method. If the *Producer* has permission to send events than the *Eventhandler* notifies all the intercedes *Consumers* and logs the process on a file.

4.2.2.9 Notify Events Service

The Notify Events service should not be confused with the previously detailed functions used by the Publish Events service, these functions where used by the *Eventhandler* to access the Notify Service running in each *Consumer*. This section is dedicated to explaining how this service is currently implemented.

```
      <resource path="notify">
            <method id="notifyEvents" name="POST">
                <request>
                    <representation
xmlns:ns2="http://www.arrowhead.org/eventhandler/registered"
element="ns2:event" mediaType="application/json"/>
                </request>
                <response/>
            </method>

      </resource>
```

Figure 44 - Notify Events WADL

4.2.2.10 Notify Events Function

Using the *notifyEvents* function Event Consumers are able to receive events forwarded by the *Eventhandler*.

```
/**
     *
     * @param event Event object.
     * @return A Response with the code 200 if the Event is received
     * successfully
     */
    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    public Response notifyEvents(Event event) {
        System.out.println("Event Payload: " + event.getPayload());
        return Response.status(200).entity("events posted ").build();
    }
```

Figure 45 - Notify Events Function Code

As shown in Figure 45 the *notifyEvents* function code is quite simple as it should be. After an event is received its data can be treated according to the developer needs. In this case we are only printing to the screen the event payload simply to confirm that the information is being received correctly.

4.2.2.11 GetHistoricalData Service

The GetHistorical data service is one of the three core services provided by the *Eventhandler* and provides information regarding events received in the past. This information is than sent to requesting entities in the form of a *LogData* object which is described in Section 4.1.9.5.1. In Figure 46 this service's *WADL* is described.

```
<resource path="historicals">
            <method id="getHistoricalData" name="POST">
                <request>
                    <representation
xmlns:ns2="http://www.arrowhead.org/eventhandler/registered"
element="ns2:filter" mediaType="application/json"/>
                </request>
                <response>
                    <representation mediaType="application/json"/>
                </response>
            </method>

</resource>
```

Figure 46 - GetHistoricalData Service WADL

4.2.2.12 GetHistoricalData Function

```
/**
     * REST interface used to get a list of events that match the specific
     * filter
     *
     * @param filter Filter object.
     * @return A response with code 200 if successful. Code 201 if no
criteria
     * matches the filter.
     * @throws JAXBException A root Exception for all JAXBExceptions.
     */
    @POST
    @Produces(MediaType.APPLICATION_JSON)
    @Consumes(MediaType.APPLICATION_JSON)
    public Response getHistoricalData(Filter filter) throws JAXBException {

        ArrayList<LogData> historical_events = new
ArrayList<>(ehs.event_controller.getHistoricalData(filter,
ehs.getDatabase()));

        if (historical_events.size() > 0) {
            return
Response.status(200).type(MediaType.APPLICATION_JSON).entity(historical_event
s).build();

        } else {
            return Response.status(204).type(MediaType.TEXT_PLAIN).entity("No
data matched the filter!").build();
        }

    }
```

Figure 47 - GetHistoricalData function code

4.2.2.13 Auxiliary Functions

```
/**
     * It checks if there is already a producer wth the specified uid.
     *
     * @param uid Producer uid
     * @return True if producer exists or false if it doesn't.
     */
    public boolean existsProducer(String uid) {
        for (Producer p : this.producer_list) {
            if (p.getUid().equalsIgnoreCase(uid)) {
                return true;
            }
        }
        return false;
    }
```

Figure 48 – Exists Producer Code

This method if used to verify if a producer is registered in the *Eventhandler*. It is located in the *ProducerController* class and simply runs through the list of registered *Producers* and if it has a match it returns *true* if don't it returns *false*.

```
/**
     * Creates a list of interested subscribers and uses the notifyAll func-
tion.
     *
     * @param event Event object.
     */
    public void notifyEvent(Event event) {

        List<Consumer> subs;
        subs = consumer_controller.applyFilter(event);
        this.notifyAll(event, subs);
    }

    /**
     * Applies a filter to currently registered consumers.
     *
     * @param event Event object.
     * @return The list of the consumers that pass the filter.
     */
    public ArrayList<Consumer> applyFilter(Event event) {

        ArrayList<Consumer> toNotify = new ArrayList<Consumer>();

        Iterator<Consumer> itSub = consumer_list.iterator();

        while (itSub.hasNext()) {

            Consumer c = itSub.next();

            // Using the type and from as filter, also can be used the
timestamp.. && c.getFilter().getFrom().compareTo(e.getFrom()) == 0
            if (c.getFilter() == null) {
                toNotify.add(c);
            } else if ((c.getFilter().getType().equals("") || c.getFil-
ter().getType().compareTo(event.getType()) == 0)
                    && (c.getFilter().getFrom().equals("") || c.getFil-
ter().getFrom().compareTo(event.getFrom()) == 0)) {
                toNotify.add(c);
            }

        }
        return toNotify;
    }
```

After receiving the event the *Eventhandler* must be able to notify interested *Consumers*. To do so it uses the filter of each *Consumer* to check if they are interested in receiving the incoming event. This is achieved through the *applyFilter* method. Currently we are only looking to match the event *type* and the *Producer uid(from)*. After having a list of interested *Consumers* the *Eventhandler* is able to access their REST *Notify* interface to send the event.

```
/**
     * Uses the Notify Rest interface on each interested subscriber and sends
     * the event.
     *
     * @param event Event object.
     * @param subscribers A list of subscribers to be notified.
     */
    public void notifyAll(Event event, List<Consumer> subscribers) {
        WebTarget target;
        Client client = ClientBuilder.newClient();
        for (Consumer consumer : subscribers) {
            System.out.println("NOTIFYING -> " + consumer.getURI());
            target = client.target(consumer.getURI());
            target.path("notify").request(MediaType.APPLICATION_JSON)
                    .post(Entity.entity(event, MediaType.APPLICATION_JSON));
        }
        log(event, subscribers);
    }
```

Figure 49 - NotifyAll Function Code

Now that we have the list of interested *Consumers* we can use their *Notify* REST interface to send the event. After all the *Consumers* are notified all the procedure is logged in a log file. In this project I opted to use the *Apache Log4j 2* which is a logging API.

```
final static Logger logger = Logger.getLogger(EventHandlerSystem.class);
```

```
/**
     * Function used to log in the log file the incoming event and notified
     * Consumers
     *
     * @param event An Event object
     * @param subs A List of Consumer type objects
     */
    public void log(Event event, List<Consumer> subs) {
        /* Database */
        event_controller.addEvent(event, subs, this.getDatabase());

        /* File */
        LogData log = new LogData();
        log.setConsumers(subs);
        log.setEvent(event);
        logger.debug(log.writeObject());
    }
```

Figure 50 - Log Event Code

Logging the incoming event is one of this project main objectives since it has a major impact in the *GetHistoricalData* service which makes use of this log file to retrieve past information. The logging potential of the *Log4j* API is immense, I will just show an example of the log file but the concept behind *Log4j* can is described in their manual [5].

```
2016-05-17 14:16:18 DEBUG EventHandlerSystem:65 porto-sensor-1 temperature 1
10°C Subscriber1;
2016-05-17 14:16:18 DEBUG EventHandlerSystem:65 porto-sensor-1 temperature 1
10°C Subscriber1;
2016-05-17 14:16:18 DEBUG EventHandlerSystem:65 porto-sensor-1 temperature 1
10°C Subscriber1;
2016-05-17 14:16:18 DEBUG EventHandlerSystem:65 porto-sensor-1 temperature 1
10°C Subscriber1;
2016-05-17 14:21:50 DEBUG EventHandlerSystem:65 porto-sensor-1 temperature 1
10°C Subscriber1;
2016-05-17 14:21:58 DEBUG EventHandlerSystem:65 porto-sensor-1 temperature 1
10°C Subscriber1;
2016-05-17 14:21:58 DEBUG EventHandlerSystem:65 porto-sensor-1 temperature 1
10°C Subscriber1;
```

Figure 51 - Log File Output

As depicted in Figure 51, the log file is quite simple. It consists of the date and time that event was processed along all the other information relative to the event. *DEBUG* is the severity level defined in the *log* function when we use *logger.debug(message)*. Next is the class that emitted the log followed by the *Producer uid*, *event type*, *severity(1-7)* and the *payload*. In this case we are simulating a temperature sensor which is sending the value of ten degrees Celsius. Finally one of the most important features of the log is the list of *Consumers* that were notified. This list is written in the file using the *Consumer uid* followed by a semicolon followed by another *Consumer uid*, etc.

```
# Root logger option
log4j.rootLogger=DEBUG, stdout, eventsFile

# Redirect log messages to console
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p
%c{1}:%L - %m%n

# Rirect log messages to a log file
log4j.appender.eventsFile=org.apache.log4j.RollingFileAppender
log4j.appender.eventsFile.File=log4j-eh.log
log4j.appender.eventsFile.MaxFileSize=5MB
log4j.appender.eventsFile.MaxBackupIndex=10
log4j.appender.eventsFile.layout=org.apache.log4j.PatternLayout
log4j.appender.eventsFile.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-
5p %c{1}:%L %m%n
```

Figure 52 - Log4j Properties File

Figure 52 is illustrating one of this project resource files which is the *Log4j* properties file which is also described in the System Description, Section 41,6.

```
/**
     * Singleton instance for the EventHanderSystem class.
     */
    private static final EventHandlerSystem instance = null;

/**
     * This function returns the singleton of this class.
     *
     * @return The singleton instance for this class.
     */
    public static EventHandlerSystem getInstance() {
        if (instance == null) {
            return new EventHandlerSystem();
        } else {
            return instance;
        }
    }
```

Figure 53 - Get Instance Functions Code

The *getInstance* function is embedded in the use of the Singleton pattern and is applied to the *EventhandlerSystem* and *Database classes.*

```
/**
     * Returns the consumer with specified uid.
     *
     * @param uid The uid of the consumer.
     * @return The consumer with the specified uid or null if it doesn't
exist.
     */
    public Consumer getConsumer(String uid) {
        for (Consumer consumer : consumer_list) {
            if (consumer.getUid().compareTo(uid) == 0) {
                return consumer;
            }
        }
        return null;
    }

/**
     * Adds a consumer to the current list of registered subscribers.
     *
     * @param c A consumer object.
     * @param db db Database singleton object.
     */
    public void addConsumer(Consumer c, Database db) {
        db.insertConsumer(c);
        consumer_list.add(c);
    }
/**
     * Returns the producer with specified uid.
     *
     * @param uid Producer uid.
     * @return The producer with the specified uid or null if it doesn't
exist.
     */
    public Producer getProducer(String uid) {
        for (Producer producer : producer_list) {
            if (producer.getUid().equalsIgnoreCase(uid)) {
                return producer;
            }
        }
        return null;
    }

/**
     * Adds a producer to the current list of registered producers.
     *
     * @param p Producer object.
     * @param db Database singleton object.
     */
    public void addProducer(Producer p, Database db) {
        db.insertProducer(p);
        this.producer_list.add(p);
    }
```

Figure 54 - Add Consumer/Producer Code

These functions are implemented in the *ConsumerController* and *ProducerController* classes and are used in the *registerConsumer/Producer* functions of the *Registry* service. They simply remove a given entity with the specified *uid*.

```java
/**
     * Removes a producer from the current list of registered producers.
     *
     * @param uid Producer uid.
     * @return True if it successfully removed the producer, or false if it
     * didn't.
     */
    public boolean removeProducer(String uid, Database db) {
        for (Producer producer : producer_list) {
            if (producer.getUid().compareTo(uid) == 0) {
                this.producer_list.remove(producer);
                db.removeProducer(producer);
                return true;
            }
        }
        return false;
    }

/**
     * Removes a consumer from the current list of registered subscribers.
     *
     * @param uid The uid of the consumer to be removed.
     * @return True if it successfully removed the consumer, or false if it
     * didn't.
     */
    public boolean removeConsumer(String uid, Database db) {
        for (Consumer consumer : consumer_list) {
            if (consumer.getUid().compareTo(uid) == 0) {
                this.consumer_list.remove(consumer);
                db.removeConsumer(consumer);
                return true;
            }
        }
        return false;
    }
```

Figure 55 - Delete Producer/Consumer Code

Both these methods and called in the *unregisterEntity* function of the *Registry* service. Using only the *uid* the application is capable of removing any *Consumer/Producer*.

```
/**
     * Returns all the events that match the specified filter.
     *
     * @param filter Filter object.
     * @param db Database singleton object.
     * @return A list of events
     */
    public ArrayList<LogData> getHistoricalData(Filter filter, Database db) {
        return (ArrayList<LogData>) db.getEvents(filter);
    }
```

Figure 56 - EventController GetHistoricalData Code

This function is embedded in the *EventController* class although all the logic is in the *Database* class it is important to make this reference since this is the function called by the *GetHistoricalData REST* interface.

4.2.2.14 Database

```
private static Database database_instance;

    /**
     * Singleton function returning the current instance of this class.
     *
     * @return A synchronized singleton instance for this class.
     */
    public static synchronized Database getInstance() {
        if (database_instance == null) {
            return new Database();
        } else {
            return database_instance;
        }
    }
```

Figure 57 - Database Singleton Code

Similarly to the *EventhandlerSystem* class, the *Database* makes use of the *Singleton Pattern*. The main difference resides in the *getInstance()* function which uses the *synchronized* function modifier. The *synchronized* mode is used so that it is not possible for two invocations on the same object to interleave. When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object.

Second, when a synchronized method exits, it automatically establishes a happens-before relationship with *any subsequent invocation* of a synchronized method for the same object. This guarantees that changes to the state of the object are visible to all threads [22].

```
/**
     * Contructor for the Database class.
     * <p>
     * Uses the database properties class to read the connection properties
and
     * set all variables accordingly.
     * </p>
     */
    private Database() {

        DBProperties dbProp = new DBProperties();
        this.username = dbProp.getUsername();
        this.passwd = dbProp.getPassword();
        this.db_url = dbProp.getDb_url();
        this.db_driver = dbProp.getDb_driver();
    }
```

Figure 58 - Database Class Contructor Code

In order to set the *Database* properties described in Figure 58 I created the *DBProperties* class which simply reads the *db.properties* file that contains the username, password, database url and driver used to do the connection. This ensures that the *Eventhandler* application can connect to all types of databases independently but in order for it to properly function besides of a *MySql* database a few changes must be made to the code, perhaps creating a *Strategy Pattern* similarly to the arrowhead connection, this is an improvement that is mentioned is Section 5.4. Figure 59 is the example of a *db.properties* file which is also described in Section 4.1.6.

```
#Cister
dburl=jdbc:mysql://192.168.50.231:3306/eventhandler
username=root
password=cister
driver=com.mysql.jdbc.Driver
```

Figure 59 - Database Properfies File

```
/**
       * Opens the connection to the database.
       */
      public void openConnection() {

            try {
                  Class.forName(this.db_driver);
                  this.con = DriverManager.getConnection(this.db_url,
this.username, this.passwd);
                  this.is_connected = true;
            } catch (ClassNotFoundException | SQLException e) {

      Logger.getLogger(Database.class.getName()).log(Level.SEVERE, null, e);
            }
      }

      /**
       * Closes the connection to the database.
       */
      public void closeConnection() {
            try {
                  this.con.close();
                  System.out.println("Closed connection to db " +
this.db_url);
                  this.is_connected = false;
            } catch (SQLException ex) {

      Logger.getLogger(Database.class.getName()).log(Level.SEVERE, null, ex);
            }
      }
```

Figure 60 - Open/Close Database Connection Code

Both of these functions deal with database connection state. One important feature is that both of them set the *is_connected* value to either true or false depending if the connection is being opened or closed. This allows the application to know before doing any operation at the database layer if there is currently an open connection.

```
/**
	 * Inserts an event in the database.
	 *
	 * @param event
	 *            Event object.
	 */
	@Override
	public void insertEvent(Event event, List<Consumer> subs) {
		try {
			LogData data = new LogData();
			data.setConsumers(subs);

			// the mysql insert statement
			String query = "insert into events (date, producer_id,
event_type, severity, payload, subscribers)"
					+ " values (?, ?, ?, ?, ?, ?)";

			java.util.Date date = new Date();
			Timestamp timestamp = new Timestamp(date.getTime());

			PreparedStatement preparedStmt =
this.con.prepareStatement(query);

			preparedStmt.setTimestamp(1, timestamp);
			preparedStmt.setString(2, event.getFrom());
			preparedStmt.setString(3, event.getType());
			preparedStmt.setInt(4,
event.getDescription().getSeverity());
			preparedStmt.setString(5, event.getPayload());
			preparedStmt.setString(6, data.writeConsumers());

			// execute the preparedstatement
			preparedStmt.execute();
		} catch (SQLException ex) {

	Logger.getLogger(Database.class.getName()).log(Level.SEVERE, null, ex);
		}
	}
```

Figure 61 - Insert Event Function Code

In Figure 61 is an example of a function that inserts an object(*Event)* in the database. I will only present this example of an insertion since the only difference from inserting a *Consumer* or a *Producer* is the *query* string and the setters methods. For example, for inserting a *Producer* the *query* string could be:

```
"insert into producer (producer_id, name, type)" + " values (?, ?, ?)";
```

This is *MySql* syntax so currently the application only supports this type of database.

```
public List<LogData> getEvents(Filter f) {

            List<LogData> data = new ArrayList<>();
            LogData entry;
            Event logEvent;
            String[] subs;
            ArrayList<String> logSubs;

            try {

                  Timestamp timestampBegin = new
Timestamp(f.getStartDateTime().getMillisecond());
                  Timestamp timestampEnd = new
Timestamp(f.getEndDateTime().getMillisecond());
                  String query = "SELECT * FROM events WHERE event_type='" +
f.getType() + "' " + "AND meta_id='"
                              + f.getDescription().getSeverity() + "' " +
"AND producer_id='" + f.getFrom() + "' "
                              + "AND date BETWEEN " + timestampBegin + " AND
" + timestampEnd;

                  Statement st = this.con.createStatement();
                  ResultSet rs = st.executeQuery(query);

                  while (rs.next()) {

                        entry = new LogData();
                        logEvent = new Event();

                        String producerID = rs.getString("producer_id");
                        String eventType = rs.getString("event_type");
                        int severity = rs.getInt("meta_id");
                        String payload = rs.getString("payload");
                        String uids = rs.getString("subscribers");

                        Metadata m = new Metadata();
                        m.setSeverity(severity);

                        logEvent.setDescription(m);
                        logEvent.setFrom(producerID);
                        logEvent.setType(eventType);
                        logEvent.setPayload(payload);

                        entry.setConsumers(uids);
                        entry.setEvent(logEvent);

                        data.add(entry);
                  }
                  return data;

            } catch (SQLException ex) {

      Logger.getLogger(Database.class.getName()).log(Level.SEVERE, null, ex);
                  return null;
            }

      }
```

Figure 62 - Get Events Function Code

This is the last function that will be presented in this document and it is also one of the most important ones. The *getEvents* function contains most of the logic required to retrieve a list of events using a filter so, this function will be used by the *GetHistoricalData* to get past information regarding events from the database. It receives a filter which allows the application to search for events considering: two timestamps (begin and end dates), the event type, severity and payload. Currently the filter must have all its properties with values (not null), in case we don't want it to be like this we could simply check what parameters are *null* and ignore them.

4.3 Tests

This chapter will present the tests that were performed during and at the end of the development process. It will also present the results of those tests, as well as some conclusions drawn from these results.

4.3.1 Functional Tests

Functional testing is primarily used to verify that a piece of software is providing the same output as required by the end-user or business. Typically, functional testing involves evaluating and comparing each software function with the business requirements. Software is tested by providing it with some related input so that the output can be evaluated to see how it conforms, relates or varies compared to its base requirements. Moreover, functional testing also checks the software for usability, such as by ensuring that the navigational functions are working as required [18].

Table 23 - Functional Test 1 - Register Consumer/Producer

ID	**1**
PRIORITY	High
TEST CASE	Register Consumer/Producer
PROCEDURE	1. An Event Consumer/Producer application uses the Registry

	service of the *Eventhandler* providing the correct data.
OBTAINED OUTCOME	*Eventhandler* successfully register the Event Consumer in its system.
EXPECTED OUTCOME	*Eventhandler* successfully register the Event Consumer in its system.

Table 24 - Functional Test 2 - Register Consumer/Producer failure

ID	**2**
PRIORITY	High
TEST CASE	Register Consumer/Producer failure
PROCEDURE	1. An Event Consumer/Producer application uses the registry service of the *Eventhandler* providing data which does not match the required criteria.
OBTAINED OUTCOME	*Eventhandler* notifies the application that the data sent is incorrect.
EXPECTED OUTCOME	*Eventhandler* notifies the application that the data sent is incorrect.

Table 25 - Functional Test 3 - Unregister Consumer/Producer

ID	**3**
PRIORITY	High
TEST CASE	Unregister Consumer/Producer
PROCEDURE	1. An Event Consumer/Producer application accesses the unregister function to remove its uid from the database.
OBTAINED OUTCOME	*Eventhandler* notifies the application that it was successfully removed from the system.

EXPECTED OUTCOME	*Eventhandler* notifies the application that it was successfully removed from the system.

Table 26 - Functional Test 4 - Query Function

ID	**4**
PRIORITY	Medium
TEST CASE	Query the *Eventhandler*
PROCEDURE	1. An Event Consumer/Producer application uses the query functions of the *Eventhandler* to gather information about applications that match the specified criteria(filter). 2. There is a match for the specified filter and a HTTP Code 200 will be returned. 3. No matches for the given criteria, a HTTP Code 201 is returned.
OBTAINED OUTCOME	A response with HTTP Code 200 or 201.
EXPECTED OUTCOME	A response with HTTP Code 200 or 201.

Table 27 - Functional Test 5 - Publish Events

ID	**5**
PRIORITY	High
TEST CASE	Publish Events
PROCEDURE	1. An Event Producer used the Publisher service of the *Eventhandler* to send an event.
OBTAINED OUTCOME	A response with HTTP Code 200.
EXPECTED OUTCOME	A response with HTTP Code 200.

Table 28 - Functional Test 6 - Notify Events

ID	**6**
PRIORITY	High
TEST CASE	Notify Events
PROCEDURE	1. After receiving an event the *Eventhandler* must create a list of interested event Consumers and notify them using the Notify service.
OBTAINED OUTCOME	The *Eventhandler* should receive an HTTP code 200 as a response.
EXPECTED OUTCOME	Received an HTTP code 200.

Table 29 . Functional Test 7 - Store Events

ID	**7**
PRIORITY	High
TEST CASE	Store Events
PROCEDURE	1. When an event is received its data should be stored in a database or a log file.
OBTAINED OUTCOME	The event data was stored in the database and local file.
EXPECTED OUTCOME	The event data should be store in the database and local file.

4.3.2 Unit Tests

A unit test is a quality measurement and evaluation procedure applied in most enterprise software development activities. Generally, a unit test evaluates how software code complies with the overall objective of the software/application/program and how its fitness affects other smaller units. Unit tests may be performed manually - by one or more developer - or through an automated software solution.

When tested, each unit is isolated from the primary program or interface. Unit tests are typically performed after development and prior to publishing, thus facilitating integration and early problem detection. The size or scope of a unit varies by programming language, software application and testing objectives [18].

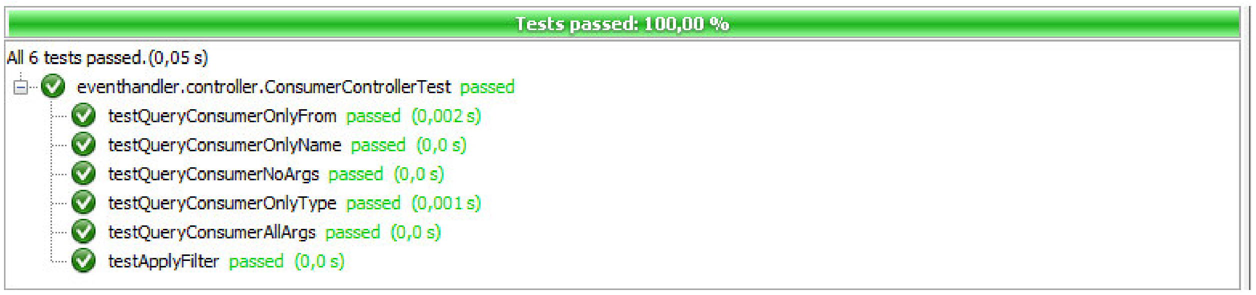
In this case I created one unit tests for each implemented service. The results of these tests are illustrated bellow.

Query Consumer/Apply Filter Unit Test

In order to properly verify if the function *query* is behaving normally with the possibility that several attributes may not be correctly formed, I've created the unit tests represented in Figure 63. To perform these tests, we insert in a list two different *Consumers*. One *Consumer* with the *Filter* set to "temperature" and the *Producer* uid to "sensor1" and the other to "pressure" and "sensor2". This function must be able to deal with queries of several types: query with all the arguments (name, type, producer uid) having values and a query with any of the other values being empty. The *query* unit tests all passed and so we conclude that the functions are operating as expected. The other function being tested is the *applyFilter*, this method receives an *event* and processes a list of *Consumers* to find if some of them are interested in receiving this *event* information. In this test only one of the *Consumers* should match with the *event* and as expected this function is also behaving as expected.

```
@Before
    public void setUp() {
        Consumer c1 = new Consumer();
        Filter f1 = new Filter("pressure", "sensor1");
        c1.setFilter(f1);

        c1.setName("consumer1");
        c1.getFilter().setType("pressure");
        c1.getFilter().setFrom("sensor1");

        Filter f2 = new Filter("temperature", "sensor2");
        Consumer c2 = new Consumer();
        c2.setFilter(f2);
        c2.setName("consumer2");

        controller = new ConsumerController();
        controller.addConsumer(c1);
        controller.addConsumer(c2);

        event = new Event();
        event.setFrom("sensor1");
        event.setType("pressure");
    }

    @Test
    public void testQueryConsumerAllArgs() {
        assertEquals(1, controller.queryConsumer("consumer1", "pressure",
"sensor1").size());
    }

    @Test
    public void testQueryConsumerOnlyName() {
        assertEquals(1, controller.queryConsumer("consumer1", "",
"").size());
    }

    @Test
    public void testQueryConsumerOnlyType() {
        assertEquals(1, controller.queryConsumer("", "pressure", "").size());
    }

    @Test
    public void testQueryConsumerOnlyFrom() {
        assertEquals(1, controller.queryConsumer("", "", "sensor1").size());
    }

    @Test
    public void testQueryConsumerNoArgs() {
        assertEquals(2, controller.queryConsumer("", "", "").size());
    }

    @Test
    public void testApplyFilter() {
        assertEquals(1, controller.applyFilter(event).size());
    }
```

Figure 63 – Query Consumer/Apply Filter Unit Test



```
Tests passed: 100,00 %
All 6 tests passed.(0,05 s)
eventhandler.controller.ConsumerControllerTest passed
    testQueryConsumerOnlyFrom passed (0,002 s)
    testQueryConsumerOnlyName passed (0,0 s)
    testQueryConsumerNoArgs passed (0,0 s)
    testQueryConsumerOnlyType passed (0,001 s)
    testQueryConsumerAllArgs passed (0,0 s)
    testApplyFilter passed (0,0 s)
```

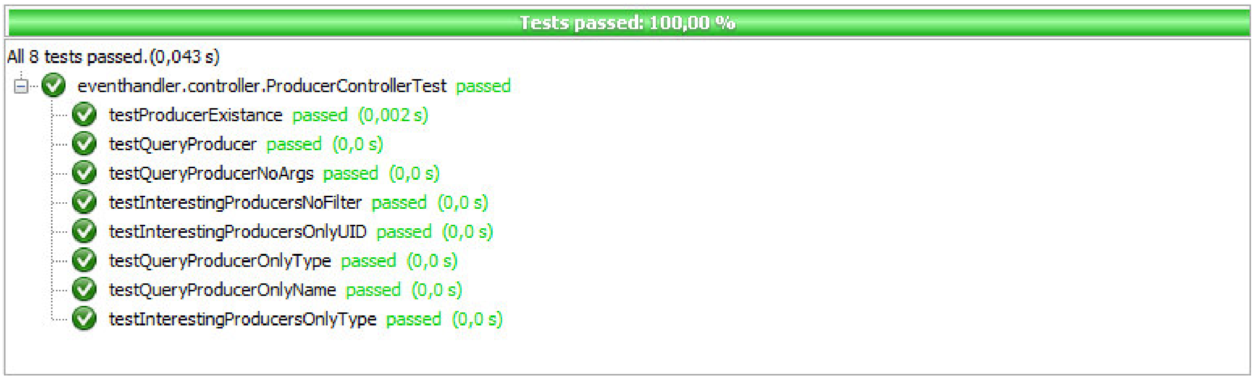
Figure 64 - Query Consumer Unit Tests Results

Query/Interesting Producer Unit Test

The Query/Interesting *Producers* unit test is very similar to the unit test in Figure 65, the main difference being that in the case of the *Producers* the *query* function only needs to deal with two parameters (*name, type*) and not three. The same behavior is expected from this *query* and the function results were as expected. The other method being test is the *interestingProducers* in which upon receiving a *Consumer* it should return true if there are any *Producers* that are interesting for this subscriber or false if there isn't. It is expected in all of the three *Consumers* (c1, c2, c3) that the result returned is true despite all of them having some of the values not defined.

```
@Before
    public void setUp() {
        Producer p1 = new Producer();
        p1.setUid("cister-sensor-123");
        p1.setName("Floor1S");
        p1.setType("temperature");
        controller.addProducer(p1);
        c2.setFilter(new Filter());
        c2.getFilter().setFrom("cister-sensor-123");
        c3.setFilter(new Filter());
        c3.getFilter().setType("temperature");
    }

    @Test
    public void testProducerExistance() {
        assertTrue(controller.existsProducer("cister-sensor-123"));
    }

    @Test
    public void testQueryProducer() {
        assertEquals(1, controller.queryProducer("Floor1S",
"temperature").size());
    }

    @Test
    public void testQueryProducerOnlyName() {
        assertEquals(1, controller.queryProducer("Floor1S", "").size());
    }

    @Test
    public void testQueryProducerOnlyType() {
        assertEquals(1, controller.queryProducer("", "temperature").size());
    }

    @Test
    public void testQueryProducerNoArgs() {
        assertEquals(1, controller.queryProducer("", "").size());
    }

    @Test
    public void testInterestingProducersNoFilter() {
        assertTrue(controller.interestingProducers(c1));
    }

    @Test
    public void testInterestingProducersOnlyUID() {
        assertTrue(controller.interestingProducers(c2));
    }

    @Test
    public void testInterestingProducersOnlyType() {
        assertTrue(controller.interestingProducers(c3));
    }
```

Figure 65 – Query/Interesting Producers Unit Test



```
Tests passed: 100,00 %
All 8 tests passed.(0,043 s)
eventhandler.controller.ProducerControllerTest passed
    testProducerExistance passed (0,002 s)
    testQueryProducer passed (0,0 s)
    testQueryProducerNoArgs passed (0,0 s)
    testInterestingProducersNoFilter passed (0,0 s)
    testInterestingProducersOnlyUID passed (0,0 s)
    testQueryProducerOnlyType passed (0,0 s)
    testQueryProducerOnlyName passed (0,0 s)
    testInterestingProducersOnlyType passed (0,0 s)
```

Figure 66 - Unit Test 2 Results

4.3.3 <u>Performance Tests</u>

Performance testing refers to the testing of software or hardware to determine whether its performance is satisfactory under any extreme and unfavorable conditions, which may occur as a result of heavy network traffic, process loading, under clocking, overclocking and maximum requests for resource utilization.

Most systems are developed under the assumption of normal operating conditions. Thus, even if a limit is crossed, errors are negligible if the system undergoes stress testing during development [18].

In order to create these tests, I used the Java Mission Control that is embedded in the Java Development Kit. This tool is described in the following chapter.

4.3.3.1 <u>Java Mission Control</u>

Oracle Java Mission Control (JMC) is a tool suite for managing, monitoring, profiling, and troubleshooting Java applications. Oracle Java Mission Control has been included in standard Java SDK since version 7u40. JMC consists of the JMX Console and the Java Flight Recorder. More plug-ins can easily be installed from within Mission Control.

Java Mission Control uses JMX to communicate with remote Java processes. The JMX Console is a tool for monitoring and managing a running JVM instance. The tool presents live data about memory and CPU usage, garbage collections, thread activity, and more. It also includes a fully featured JMX MBean browser that you can use to monitor and manage MBeans in the JVM and in your Java application.

Machine used for testing

The machine specs used to test the performance of the *Eventhandler* is quite important to reference since it will influence most of the results. The computer used has an *Intel(R) Core(TM) i5-4960K CPU @ 3.5GHz* processor and sixteen Gigabytes of RAM (Random Access Memory). All the graphs presented represent tests driven in this hardware.

Machine CPU usage

Eventhandler Startup

When the *Eventhandler* application is started we can see that the peek CPU usage in this machine is 34% and 6% when it is idle.

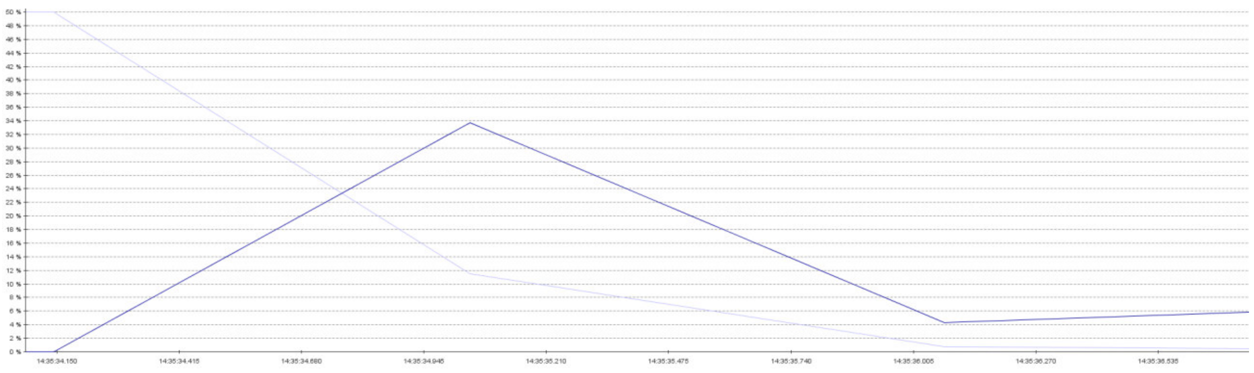


Figure 67 – Eventhandler Startup CPU Usage (yy-CPU%;xx-Time(hh:mm:ss)

Registering a Consumer

Regarding the registry of a *Consumer* the machine CPU usage peeks at 61% while the JVM CPU usage peeks at 6%. When it comes to a time consuming factor the entire operation took approximately one second, this includes the usage of the REST interfaces, registering the *Consumer* in memory and in the MySQL database.

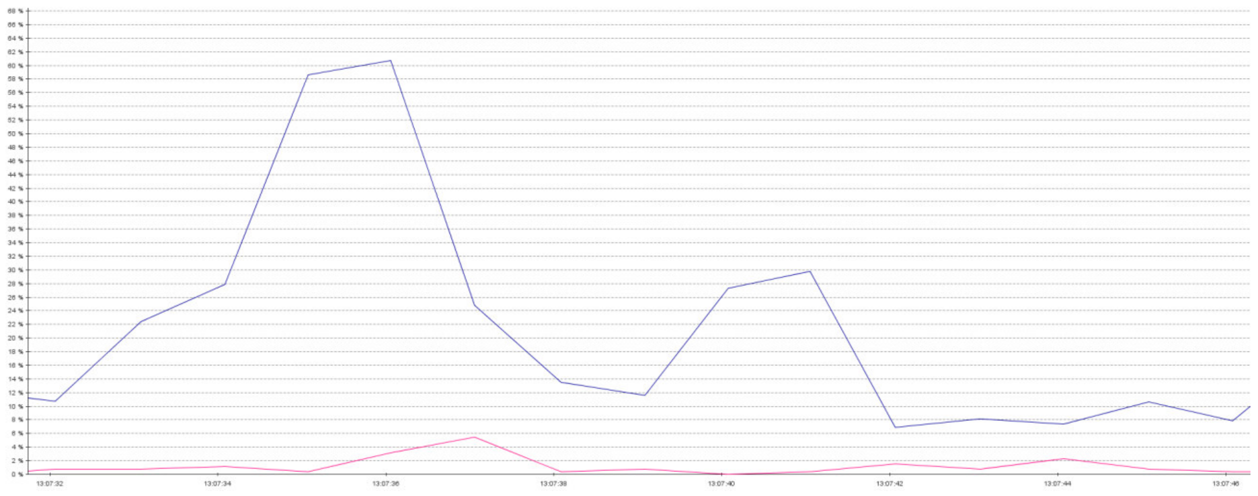


Figure 68 - Registering a Consumer CPU Usage (yy-CPU%;xx-Time(hh:mm:ss)

Registering a Producer

Similarly, to the previous test the machine CPU usage peeks at 61%. The main differences are in the time consumed and in the JVM CPU usage, which are respectively less than a second, approximately 0.5 seconds, and the JVM peeks at 4%.

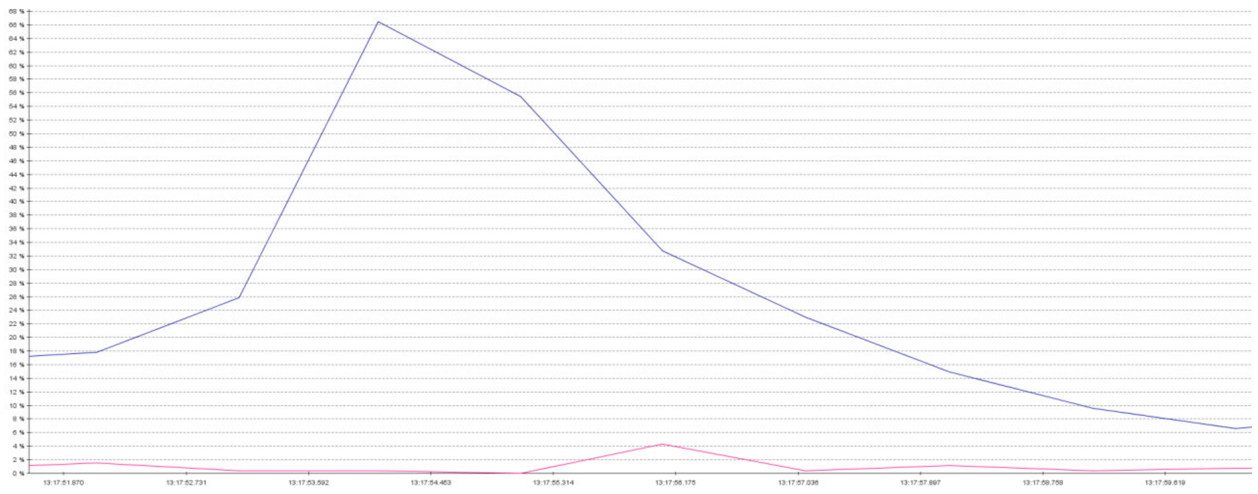


Figure 69 - Registering a Producer CPU Usage (yy-CPU%;xx-Time(hh:mm:ss)

The complete cicle

In this example all of the *Eventhandler*'s core features are being tested with the exception that both a *Conusumer* and a *Producer* are already registered. These features include:

Publishing one hundred events using the *Publish* service Eventhandler System - Arrowhead

- (*Producer)*.
- Logging the incoming events in a file and in a MySQL database (*Eventhandler)*.
- Notifying interested *Consumers* using their *Notify* service (*Eventhandler)*.
- Printing the event payload confirming that the entire operation was successful (*Consumer)*.

Note that the following graphs only provide information regarding the operations executed in the *Eventhandler* application and not operations executed in both the *Consumer* and *Producer* applications.

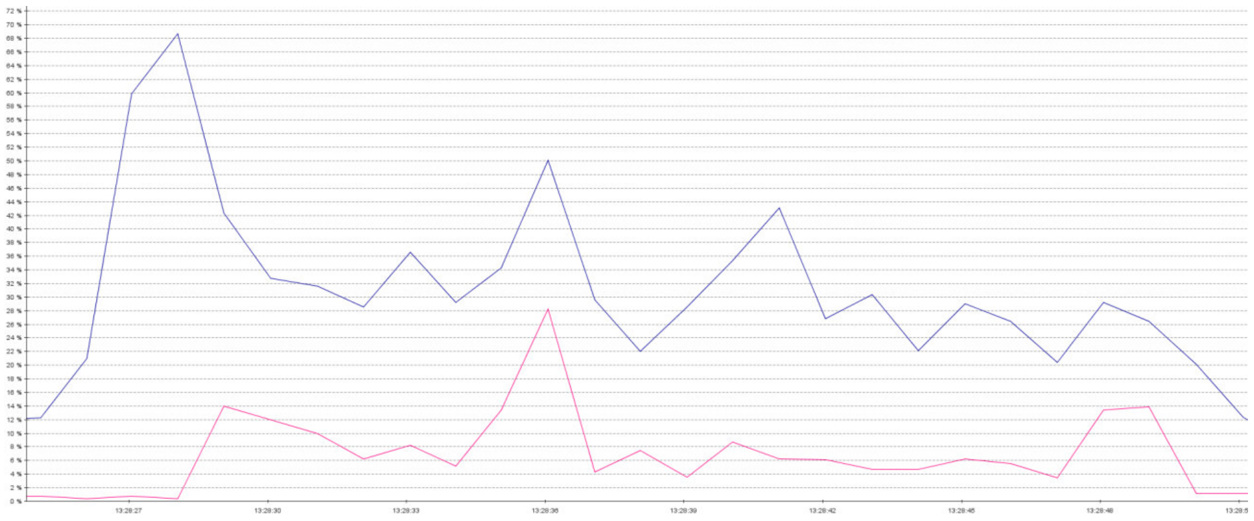


Figure 70 - Complete Cycle CPU Usage (yy-CPU%;xx-Time(hh:mm:ss)

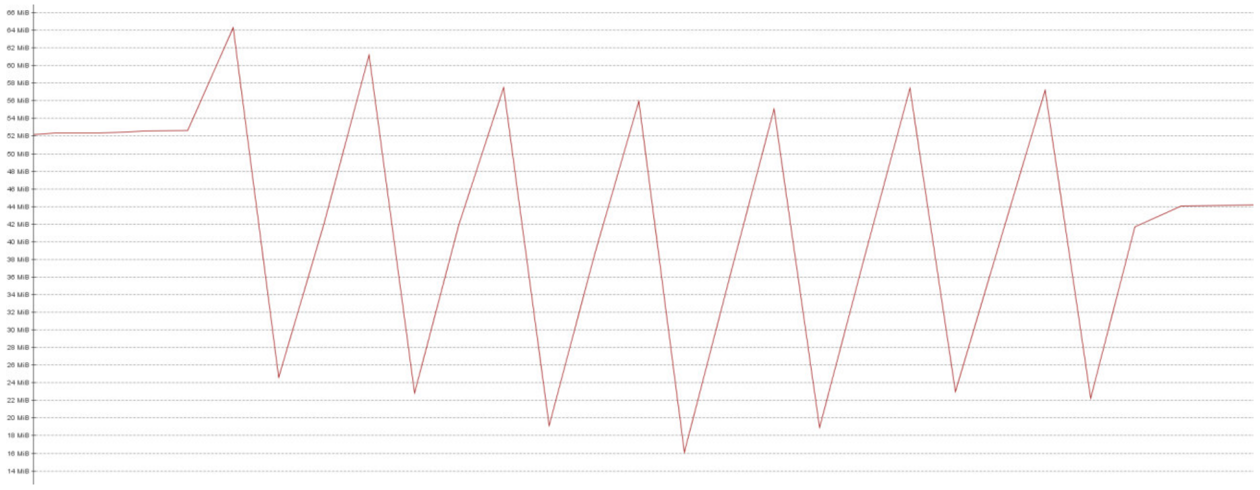


Figure 71 - Complete Cycle Java Heap (yy-java heap memory(MegaBytes) xx- Time(hh:mm:ss)

5 Conclusions

In this chapter a summary of this report is presented along with what goals were achieved, the main difficulties encountered in all the project phases and what improvements can be made concerning the application in the future.

To conclude a final appreciation is made to this project's conclusion.

5.1 Report Summary

The objective of this report is to describe and explain the planning and development of the *Eventhandler* for the Arrowhead European project. This application was proposed in the ambit of the discipline of Project/Internship of the degree in Informatics Engineering of the Instituto Superior de Engenharia do Porto. It contains five major chapters which include an *Introduction* to the project, the *Scope* in which this project was introduced, the *Work Environment* that explains the planning phase of this project along with the working methodologies and the technologies used. The *Development of the Solution* chapter illustrates the code created to achieve the applications main features, along with brief descriptions explaining the purpose behind each developed function or service.

Finally, the *Conclusions* chapter where this sub section is inserted, provides a report summary plus the goals achieved, difficulties encountered and future improvements to the application.

5.2 Goals Achieved

The main objectives proposed at the beginning of this internship were successfully implemented. Although in my opinion some features have room for improvement. Despite taking a longer period than planned the *Eventhandler* is operational and can perform according to the earlier projections. Its documentation was reviewed and accepted by the project partners and the code is available as open source.

5.3 Difficulties

Many difficulties encountered while developing this project were related to Arrowhead Framework implementation. Firstly, I misinterpreted some documents in the Arrowhead SVN which lead in some cases to wasting a lot of time trying to decompile code or trying to develop functionalities that were already developed by other people. Another problem was related to the technologies used, since during my classes in ISEP I never touched most of the core technologies in this project (Maven, Jersey, REST) I needed to read a lot about this concept and I also needed to develop some examples of code which took most of the first month of work. Lastly, the integration of the *Eventhandler* with the Arrowhead Framework was not easy because it involved reading and understanding many lines of code already developed, but in the end it was most gratifying that the integration was working according to plan.

5.4 Improvements

There is still a lot of work to be done regarding security features and multithreading. Security is of most importance in the context of this work since in the Internet of Things concept all different kinds of devices can be plugged to the internet and in this case, connected to the *Eventhandler*. So, using a security protocol like *AAA* (Authentication, Authorization and Accounting) is one of the many possibilities or using *SSL* (Secure Socket Layer) in our *HHTP* servers.

Being in the context of the distributed programming the application is not currently as robust as to the point of handling for example with network failures. It is possible to create other solution that would make the *Eventhandler* a much more robust software but in most cases it would also imply a deficit in performance. These improvements are in my opinion important to be discussed in the future taking in account se the performance deficit that may cause to the application.

One of the other concerns may be the usage of different databases besides *MySql* and as explained in Section 4.2.2.14 there are several solutions for this problem.

Since the *Eventhandler*, in a grand scale scenario, may be connected to a large number of devices multithreading is essential to provide performance. This increase is obtained by taking advantage of a multicore machine, in which multiple incoming evens can be processed at the same time.

5.5 Final Appreciation

I found really challenging at first to be involved in an international project, mostly because it evolved tight schedules, several changes of plans regarding the system features and having to interact with so many people. In the end I think it offered me the opportunity to develop skills that are currently very demanded by the labor market. I was also able to learn a lot of new technologies and working methodologies which will enhance my opportunities to find a job in the near future.

Overall, this internship proved out very gratifying in both acquiring experience and developing personal skills.

6 Bibliography

[1] Klisics, M. (2013). Arrowhead IDD Service Discovery DNS-SD v1.0.0.

[2] Klisics, M. (2013). Arrowhead IDD Authorisation Control REST_WS v1.0.0.

[3] Klisics, M. (2013). Arrowhead IDD Orchestration Management REST_WS v1.0.0.

[4] Eliasson, J. (2015). Arrowhead SD Historian v0.1 URL

[5] Log4j user manual. https://logging.apache.org/log4j/1.2/manual.html

[6] Eventhandler Registry Service Description – Arrowhead/WP8/Docs/Arrowhead SD EventHandlerRegistry v2.0.docx

[7] Eventhandler PublishEvents Service Description – Arrowhead/WP8/Docs/ Arrowhead SD EventHandlerPublish v2.0.docx

[8] Mats Johansson (2014), forge.soa4d.org/plugins/mediawiki/wiki/arrowhead/index.php/Design_Documentation_Guidelines, Arrowhead Framwork Definition v1.0.docx

[9] http://www.businessinsider.com/how-the-internet-of-things-market-will-grow-2014-10

[10] https://hbr.org/2014/10/the-sectors-where-the-internet-of-things-really-matters

[11] forge.soa4d.org

[12] eclipse.org/jetty

[13] jersey.java.net

[14] Eliasson, J. (2015). Arrowhead SD Historian v0.1 URL

[15] maven.apache.org

[16] www.wikipedia.org

[17] www.ichanical.com/agile-software-development-everything-you-need-to-know/

[18] www.techopedia.com/definition/19509/functional-testing

[19] www.ibm.com

[20] www.stackoverflow.com

[21] www.docs.oracle.com

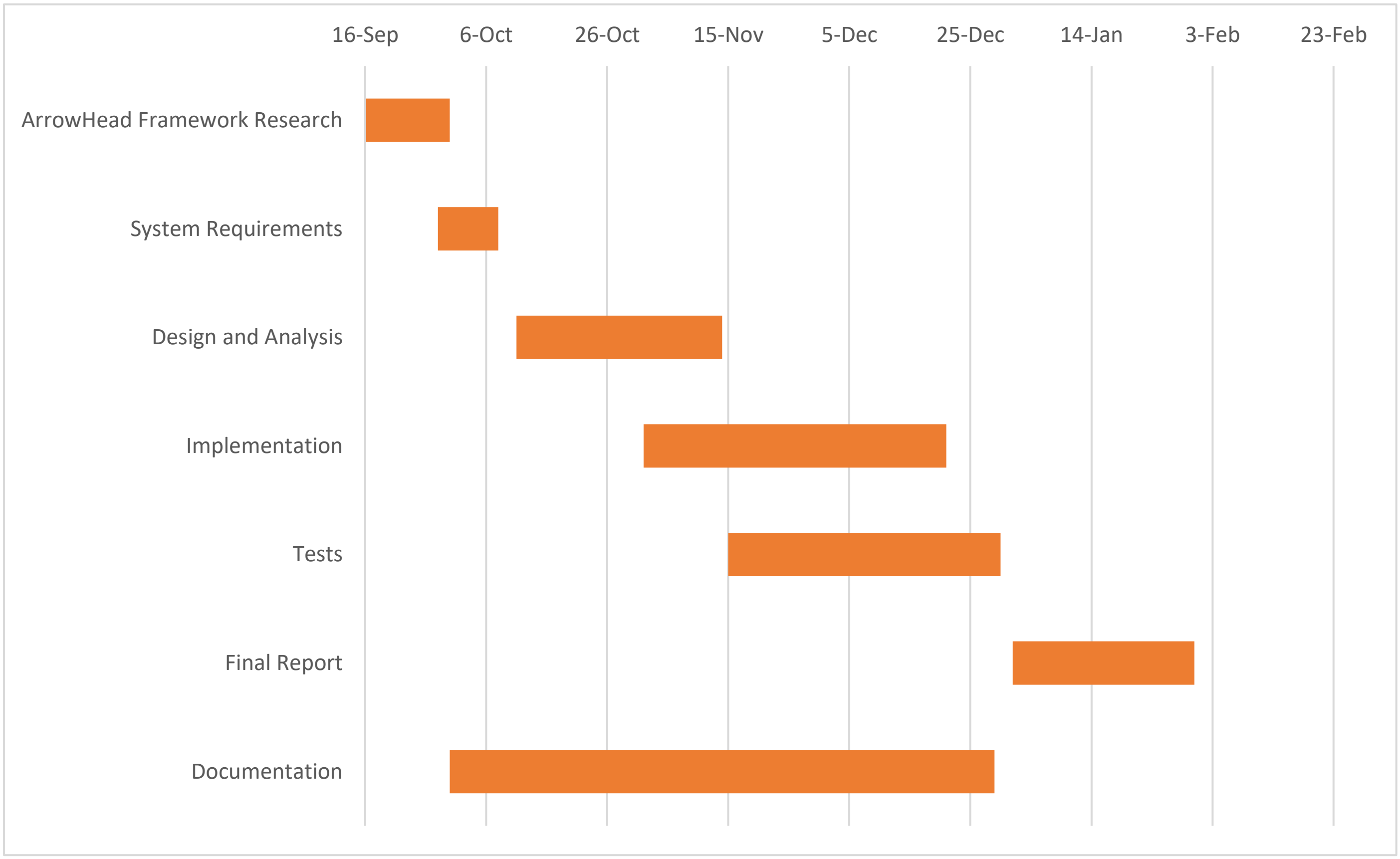
[22] www.prosysopc.com

[23] Gary Orenstein, IoT, at http://venturebeat.com/2016/08/27/ge-and-cisco-face-off-over-industrial-iot/

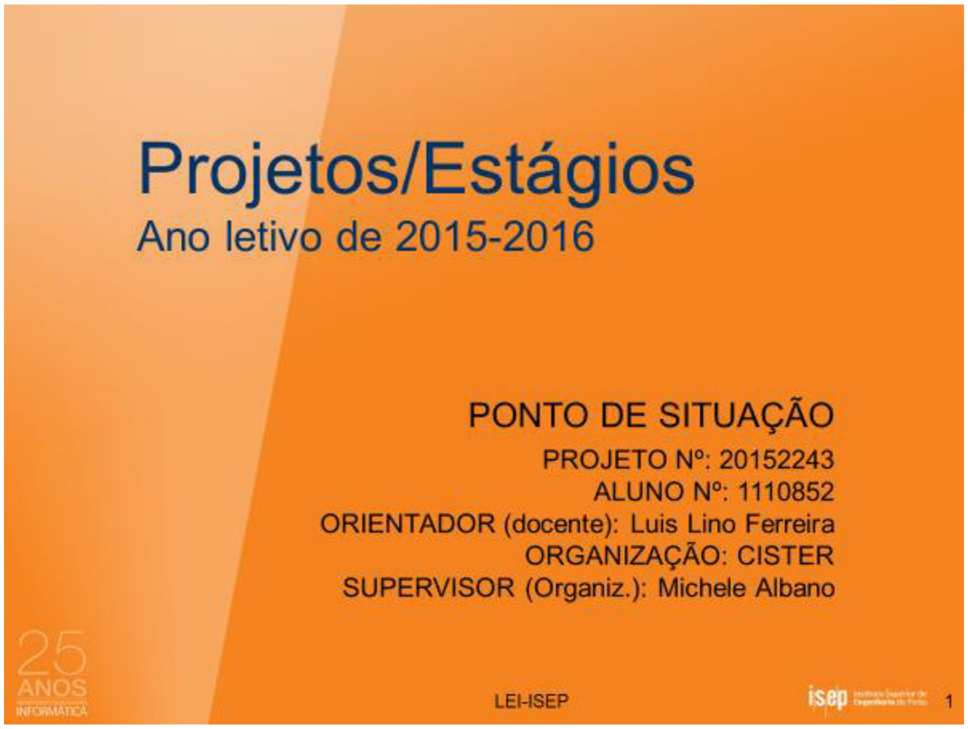
[24] Arrowhead Communication Profile, https://forge.soa4d.org/plugins/mediawiki/wiki/arrowhead-f/index.php/Core_Systems_Communication_Profiles

7 Appendixes

7.1 Gantt Diagram



7.2 Project Status



Projetos/Estágios

Ano letivo de 2015-2016

PONTO DE SITUAÇÃO

PROJETO Nº: 20152243

ALUNO Nº: 1110852

ORIENTADOR (docente): Luis Lino Ferreira

ORGANIZAÇÃO: CISTER

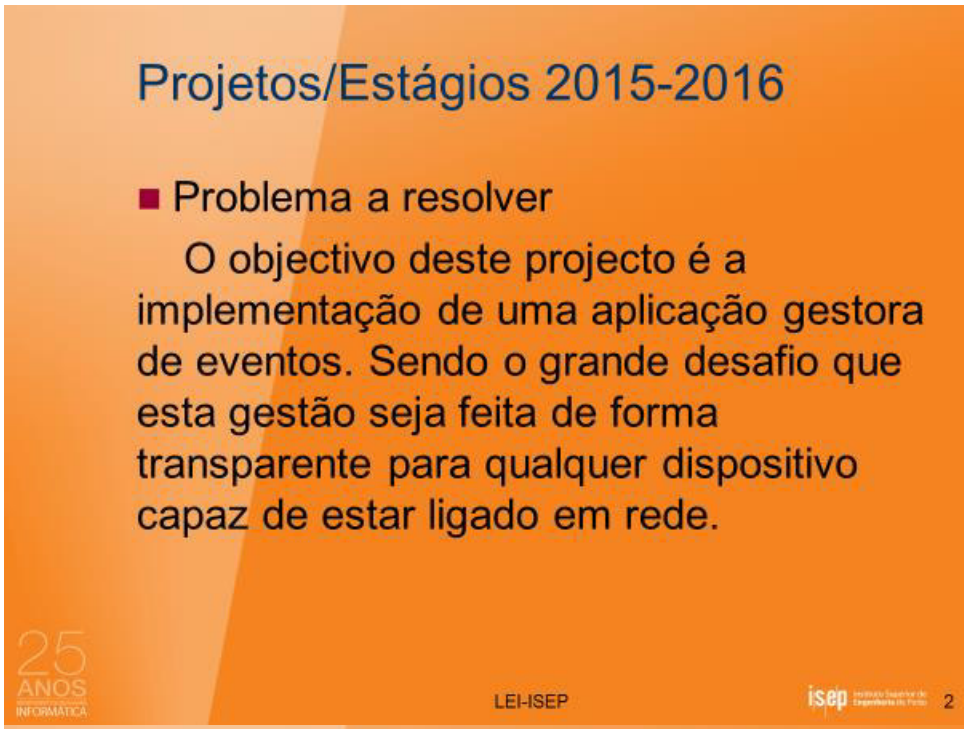
SUPERVISOR (Organiz.): Michele Albano

25 ANOS INFORMÁTICA

LEI-ISEP

isep

1



Projetos/Estágios 2015-2016

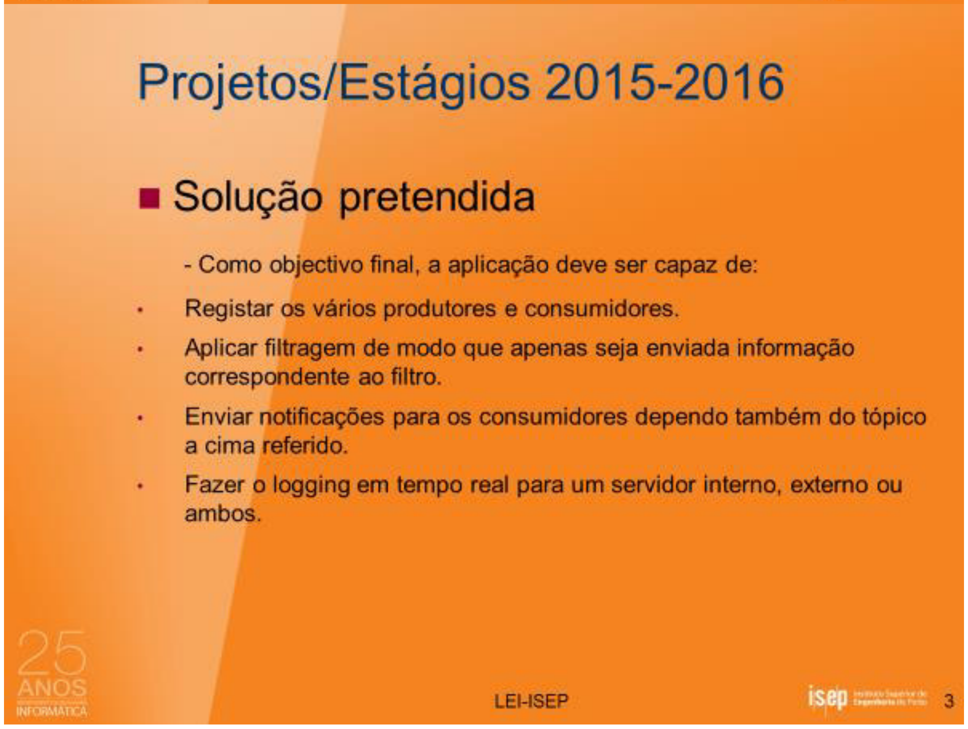
- Problema a resolver

O objectivo deste projecto é a implementação de uma aplicação gestora de eventos. Sendo o grande desafio que esta gestão seja feita de forma transparente para qualquer dispositivo capaz de estar ligado em rede.

25 ANOS INFORMÁTICA

LEI-ISEP

isep 2



Projetos/Estágios 2015-2016

- Solução pretendida

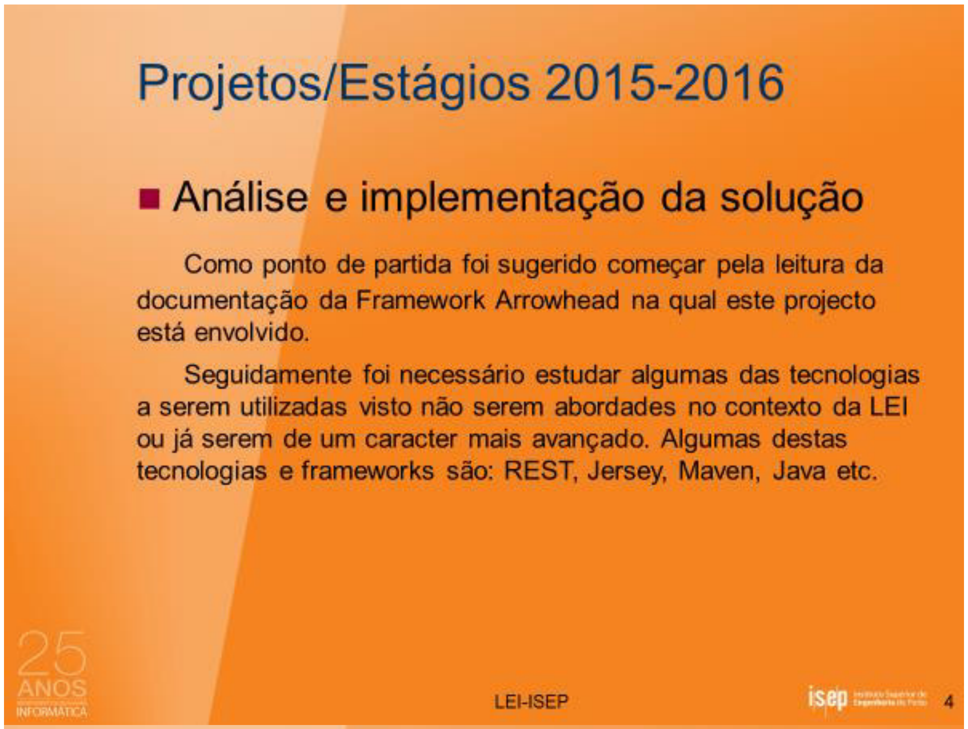
- Como objectivo final, a aplicação deve ser capaz de:

- Registar os vários produtores e consumidores.
- Aplicar filtragem de modo que apenas seja enviada informação correspondente ao filtro.
- Enviar notificações para os consumidores dependo também do tópico a cima referido.
- Fazer o logging em tempo real para um servidor interno, externo ou ambos.

25 ANOS INFORMÁTICA

LEI-ISEP

isep 3



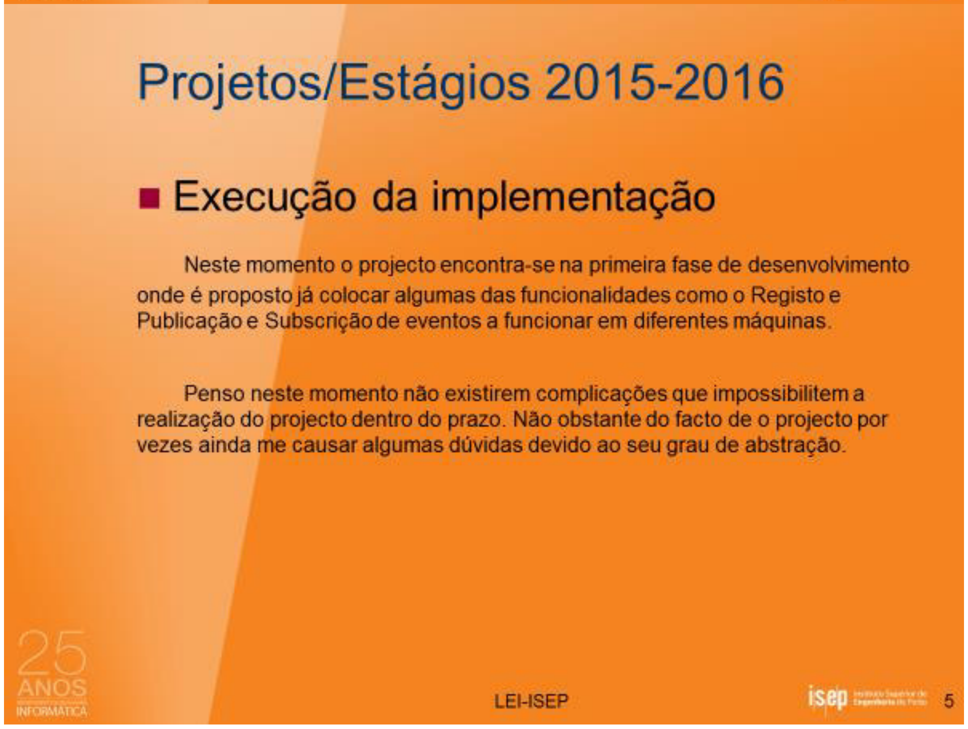
Projetos/Estágios 2015-2016

- Análise e implementação da solução

Como ponto de partida foi sugerido começar pela leitura da documentação da Framework Arrowhead na qual este projecto está envolvido.

Seguidamente foi necessário estudar algumas das tecnologias a serem utilizadas visto não serem abordades no contexto da LEI ou já serem de um caracter mais avançado. Algumas destas tecnologias e frameworks são: REST, Jersey, Maven, Java etc.

25 ANOS INFORMÁTICA LEI-ISEP 4



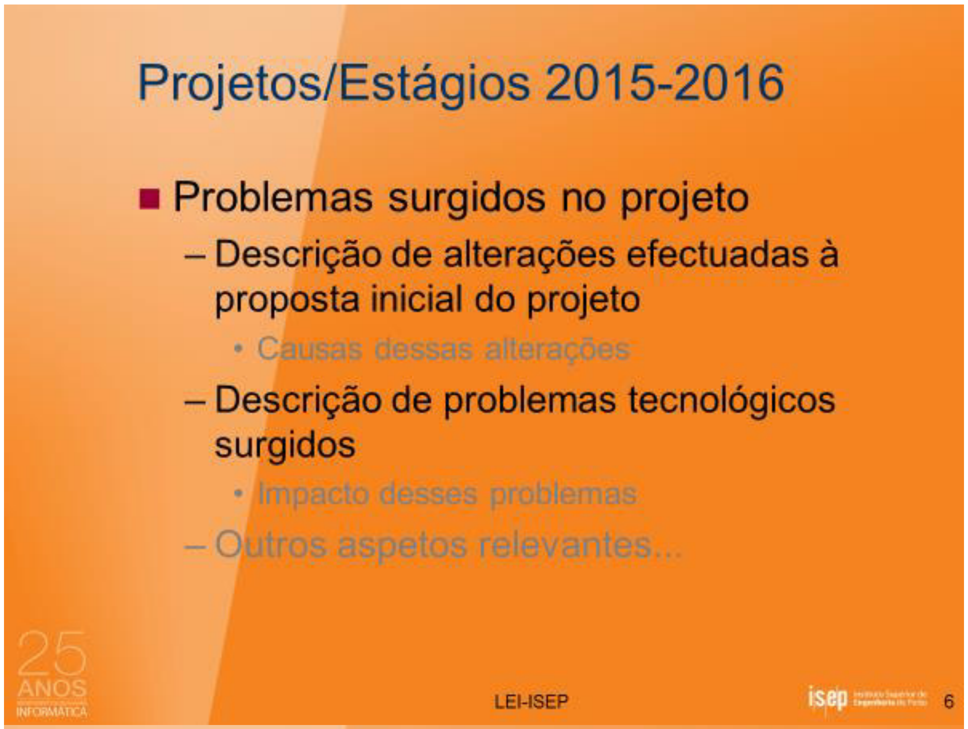
Projetos/Estágios 2015-2016

- Execução da implementação

Neste momento o projecto encontra-se na primeira fase de desenvolvimento onde é proposto já colocar algumas das funcionalidades como o Registo e Publicação e Subscrição de eventos a funcionar em diferentes máquinas.

Penso neste momento não existirem complicações que impossibilitem a realização do projecto dentro do prazo. Não obstante do facto de o projecto por vezes ainda me causar algumas dúvidas devido ao seu grau de abstração.

25 ANOS INFORMÁTICA LEI-ISEP 5



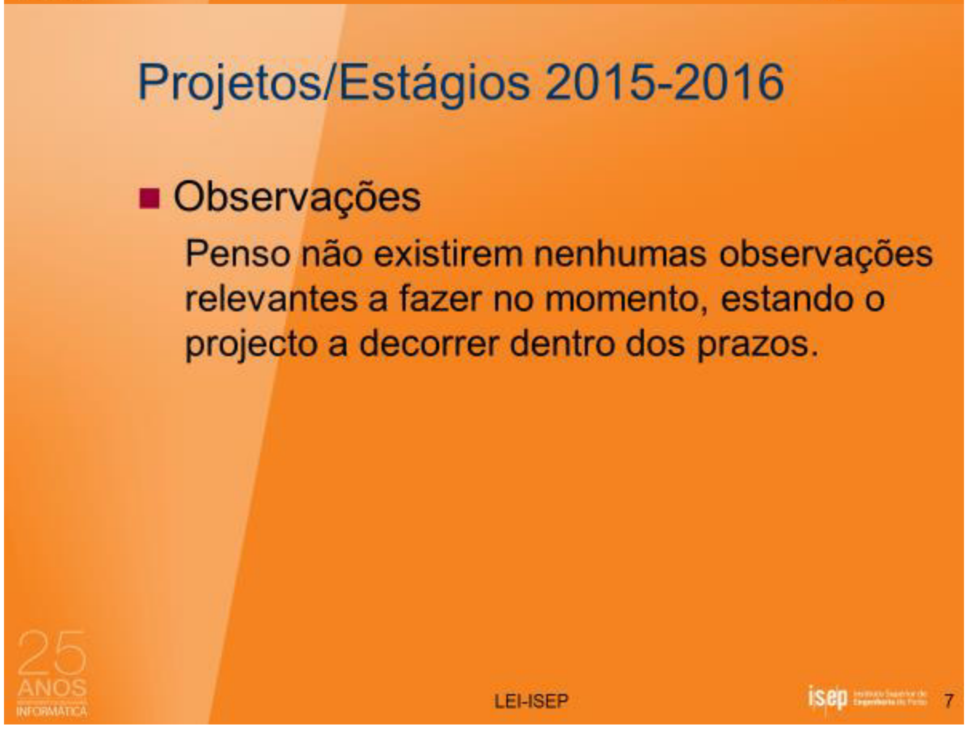
Projetos/Estágios 2015-2016

- Problemas surgidos no projeto
  - Descrição de alterações efectuadas à proposta inicial do projeto
    - Causas dessas alterações
  - Descrição de problemas tecnológicos surgidos
    - Impacto desses problemas
  - Outros aspetos relevantes...

25 ANOS INFORMÁTICA

LEI-ISEP

isep 6



Projetos/Estágios 2015-2016

- Observações

  Penso não existirem nenhumas observações relevantes a fazer no momento, estando o projecto a decorrer dentro dos prazos.

25 ANOS INFORMÁTICA

LEI-ISEP

isep 7