# BETaaS

**Building the Environment for the Things as a Service**

Grant Agreement no.: **317674**

Call identifier: **FP7-ICT-2011-8**

---

## D4.1 – BETaaS platform - Initial release

<table>
<thead>
<tr>
<th>Deliverable:</th>
<th>D4.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title:</td>
<td>BETaaS platform - Initial release</td>
</tr>
<tr>
<td>Due date:</td>
<td>30 September 2013</td>
</tr>
<tr>
<td>Actual submission date:</td>
<td>10 October 2013</td>
</tr>
<tr>
<td>Lead contractor for this deliverable:</td>
<td>Hewlett-Packard Italiana</td>
</tr>
<tr>
<td>Dissemination Level:</td>
<td>PU</td>
</tr>
<tr>
<td>Version:</td>
<td>1.0</td>
</tr>
</tbody>
</table>

## Abstract

This document describes the initial release of the BETaaS platform, where a preliminary subset of features have been properly developed and integrated, taking into account the initial results coming from the research and design work packages. The document is an addendum to the actual BETaaS software platform, which is the major deliverable outcome.
## Document History

<table>
<thead>
<tr>
<th>Version</th>
<th>Issue Date</th>
<th>Total pages</th>
<th>Modified pages</th>
<th>Modifications implemented</th>
<th>Author</th>
<th>Organization</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>12/07/2013</td>
<td>-</td>
<td>-</td>
<td>ToC, initial platform components definition and assignment</td>
<td>Alessandro Mamelli</td>
<td>HP</td>
</tr>
<tr>
<td>0.2</td>
<td>17/07/2013</td>
<td>-</td>
<td>-</td>
<td>Added the Coding Guidelines Section</td>
<td>INCS</td>
<td>INCS</td>
</tr>
<tr>
<td>0.3</td>
<td>27/07/2013</td>
<td>-</td>
<td>-</td>
<td>Added section Software development guidelines</td>
<td>Alessandro Mamelli, Davide Sommacampagna</td>
<td>HP</td>
</tr>
<tr>
<td>0.4</td>
<td>09/08/2013</td>
<td>-</td>
<td>-</td>
<td>Added guidelines to deploy Blueprint service, using Karaf and Test with JUnit</td>
<td>Alessandro Mamelli, Davide Sommacampagna</td>
<td>HP</td>
</tr>
<tr>
<td>0.5</td>
<td>25/08/2013</td>
<td>-</td>
<td>-</td>
<td>Added section for Karaf global log configuration and distributed OSGi feature for Karaf</td>
<td>Alessandro Mamelli, Davide Sommacampagna</td>
<td>HP</td>
</tr>
<tr>
<td>0.6</td>
<td>10/09/2013</td>
<td>-</td>
<td>-</td>
<td>Added 3th party repository</td>
<td>Alessandro Mamelli, Davide Sommacampagna</td>
<td>HP</td>
</tr>
<tr>
<td>0.7</td>
<td>20/09/2013</td>
<td>-</td>
<td>-</td>
<td>Added Data Manager instructions</td>
<td>Alessandro Mamelli, Davide Sommacampagna</td>
<td>HP</td>
</tr>
<tr>
<td>0.8</td>
<td>30/09/2013</td>
<td>-</td>
<td>-</td>
<td>Added Installation and Testing</td>
<td>Alessandro Mamelli, Davide Sommacampagna</td>
<td>HP</td>
</tr>
<tr>
<td>0.9</td>
<td>14/10/2013</td>
<td>-</td>
<td>-</td>
<td>Added Platform implementation and demonstration application</td>
<td>WP4 Partners</td>
<td>WP4 Partners</td>
</tr>
<tr>
<td>1.0</td>
<td>17/10/2013</td>
<td>-</td>
<td>-</td>
<td>Final harmonization</td>
<td>Alessandro Mamelli</td>
<td>HP</td>
</tr>
</tbody>
</table>
Executive Summary

Within the BETaaS project, Work Package 4 “Platform implementation” (WP4) aims to implement the BETaaS software platform.

In particular, the main WP4 objectives are:

- To ensure the technical coordination of the platform development and integration work (integration methodology definition, incremental release steps, etc.)
- To select and validate the appropriate reference technological baseline on top of which the different architecture components will be developed and then properly integrated in the BETaaS platform
- To develop and integrate the components of the BETaaS platform
- To design and execute the integration tests of the BETaaS platform
- To provide a coherent software distribution that will integrate all the specific components into the different BETaaS platform releases

This document presents the results achieved in WP4 during the first phase of the project. The results of WP4 Tasks 4.1 (Development and integration coordination), WP4 Tasks 4.2 (Technological baseline definition), Task 4.3 (Platform development) and Task 4.4 (Integration tests design and execution) are presented. The document is the accompanying textual specification of the major deliverable result: the packaged developed software that implements the BETaaS platform. The document and the developed software constitute the overall deliverable output.

The work have been carried out in close cooperation and synchrony with WP1-2-3, properly integrating the delivered results in a coherent and uniform manner, and then finally releasing the overall BETaaS platform. The progress of these tasks will be one of the major drivers of the work package dedicated to validation trials (WP5).
# Table of Contents

**Executive Summary** ......................................................................................................................... 3

1. **Introduction** ............................................................................................................................... 7

2. **Table of acronyms** ...................................................................................................................... 8

3. **Platform software development management** ............................................................................ 9

   3.1. **Apache Maven** .................................................................................................................... 9
   
   3.1.1. **Setting software the Nexus repository for BETaaS and 3th party libraries** ..................... 9
   
   3.2. **The software development environment** ........................................................................... 10
   
   3.2.1. **The software repository** .............................................................................................. 10
   
   3.2.2. **The software development Redmine sub-project** .......................................................... 10
   
   3.3. **The software coding guidelines** .......................................................................................... 11
   
   3.3.1. **Lines Length and Indentation** ....................................................................................... 11
   
   3.3.2. **Implementation Comments** .......................................................................................... 11
   
   3.3.3. **Source File Header** ..................................................................................................... 11
   
   3.3.4. **Class and Interfaces Declaration** .................................................................................. 12
   
   3.3.5. **Naming** ....................................................................................................................... 12
   
   3.3.6. **Wrapping Lines** ......................................................................................................... 12
   
   3.3.7. **Variables** .................................................................................................................... 13
   
   3.3.8. **Scope** .......................................................................................................................... 13

4. **Platform technological baseline** ............................................................................................... 14

   4.1. **OSGI** ..................................................................................................................................... 14
   
   4.1.1. **Importing the BETaaS baseline software** ..................................................................... 15
   
   4.1.2. **Creating a new OSGI bundle with Eclipse and Maven** .................................................. 16
   
   4.1.3. **Dependencies that are already defined as superpom level** ........................................... 21
   
   4.1.4. **How to run and debug bundles on Eclipse** ................................................................. 21
   
   4.2. **Apache Karaf** ..................................................................................................................... 23
   
   4.2.1. **Installing Karaf** .......................................................................................................... 24
   
   4.2.2. **Running Karaf** ........................................................................................................... 24
   
   4.2.3. **Configuring Karaf** ....................................................................................................... 25
   
   4.2.4. **Using Apache Blueprint for deployment of OSGi services** ......................................... 27
   
   4.2.5. **Using Whiteboard pattern for OSGi** .......................................................................... 29
   
   4.2.6. **Deploy Bundles on Apache Karaf** .............................................................................. 31
   
   4.2.7. **Using DOSGi** .............................................................................................................. 33
   
   4.2.8. **Deploy Data Service on Apache Karaf** ....................................................................... 35
   
   4.2.9. **Provide bundles with configuration parameters** ......................................................... 35
   
   4.2.10. **Getting data from the Data Manager Service** ............................................................ 37
   
   4.2.10.1. **Getting a connection from the pool** ..................................................................... 38
   
   4.2.10.2. **Exchanging a Java JPA entity object** ................................................................... 38

5. **Platform implementation** ........................................................................................................... 40
5.1. **High level architecture** ................................................................. 40
5.2. **InstanceManager** ........................................................................ 40
  5.2.1. **Class diagrams** ....................................................................... 41
  5.2.2. **Public interfaces** ................................................................... 41
  5.2.3. **Code packaging** .................................................................... 42
5.3. **ServiceManager** ......................................................................... 42
  5.3.1. **Class diagrams** ..................................................................... 43
  5.3.2. **Public interfaces** .................................................................. 43
  5.3.3. **Code packaging** ................................................................... 45
5.4. **TaaS Resource Manager** ................................................................. 46
  5.4.1. **Class diagrams** ..................................................................... 46
  5.4.2. **Public interfaces** .................................................................. 48
  5.4.3. **Code packaging** ................................................................... 51
5.5. **ThingsAdaptor** ............................................................................ 51
  5.5.1. **Class diagrams** ..................................................................... 52
  5.5.2. **Public interfaces** .................................................................. 52
  5.5.3. **Code packaging** ................................................................... 52
5.6. **BigDataManager** .......................................................................... 53
  5.6.1. **Class diagrams** ..................................................................... 53
  5.6.2. **Public interfaces** .................................................................. 54
  5.6.3. **BETaaS BDM Service Core** .................................................. 54
  5.6.4. **BETaaS BDM TaaS Core** ..................................................... 55
  5.6.5. **Code packaging** ................................................................... 55
5.7. **VMMManager** .............................................................................. 55
  5.7.1. **Class diagrams** ..................................................................... 56
  5.7.2. **Public interfaces** .................................................................. 56
  5.7.3. **Code packaging** ................................................................... 57
5.8. **QoSManager** ................................................................................ 58
  5.8.1. **Class diagrams** ..................................................................... 58
  5.8.2. **Public interfaces** .................................................................. 59
  5.8.3. **Code packaging** ................................................................... 60
5.9. **ContextManager** .......................................................................... 60
  5.9.1. **Class diagrams** ..................................................................... 60
  5.9.2. **Public interfaces** .................................................................. 61
  5.9.3. **Code packaging** ................................................................... 63
5.10. **SecurityManager** ........................................................................ 63
   5.10.1. **Class diagrams** .................................................................. 63
   5.10.2. **Public interfaces** ................................................................ 64
   5.10.3. **Code packaging** ................................................................. 65
   5.10.4. **Additional notes on bundles activation steps**...................... 65
5.11. **TaaS Trust Manager** ................................................................... 66
   5.11.1. **Class diagrams** .................................................................. 66
   5.11.2. **Public interfaces** ................................................................ 68
   5.11.3. **Code packaging** ................................................................ 69
6. **Platform testing** ............................................................................... 70
   6.1. **Unit tests** .................................................................................. 70
6.2. **Integration tests** ............................................................................ 71
   6.2.1. **Integration test with Karaf** .................................................. 72
7. **Platform installation and configuration** ............................................ 74
  7.1.1. **Platform software requirements** ............................................ 74
  7.1.2. **Platform software download** ................................................ 74
  7.1.3. **Platform software installation** ................................................ 74

**Version 1.0**  
**Page: 5 of 81**
7.1.4. Platform baseline dependencies ................................................................. 75
7.1.5. Platform Adaptation layer installation ........................................................ 75
7.1.6. Platform TaaS layer installation ................................................................. 75
7.1.7. Platform Service layer installation ............................................................. 76
7.1.8. Platform instance installation ..................................................................... 76
7.1.9. Platform gateway build ............................................................................... 77
7.1.10. Platform instance build ............................................................................ 77

8. Demonstration application ............................................................................... 78
9. Conclusions ...................................................................................................... 80
10. References ....................................................................................................... 81
1. Introduction

The BETaaS platform aims at providing a runtime environment relying on a local cloud of gateways to support the deployment and execution of content-centric M2M applications. The proposed platform will seamlessly integrate existing heterogeneous M2M systems. The latter constitute the so-called underlying ‘physical’ layer, consisting of smart things connected to a global network infrastructure and providing basic services either directly or through intermediate gateways.

In such context, Work Package 4 “Platform implementation” (WP4) aims to implement the BETaaS software platform.

In particular, the main WP4 objectives are:

- To ensure the technical coordination of the platform development and integration work (integration methodology definition, incremental release steps, etc.)
- To select and validate the appropriate reference technological baseline on top of which the different architecture components will be developed and then properly integrated in the BETaaS platform
- To develop and integrate the components of the BETaaS platform
- To design and execute the integration tests of the BETaaS platform
- To provide a coherent software distribution that will integrate all the specific components into the different BETaaS platform releases

This document presents the results achieved in WP4 during the first phase of the project. The results of WP4 Tasks 4.1 (Development and integration coordination), WP4 Tasks 4.2 (Technological baseline definition), Task 4.3 (Platform development) and Task 4.4 (Integration tests design and execution) are presented. The document is the accompanying textual specification of the major deliverable result: the packaged developed software that implements the BETaaS platform. The document and the developed software constitute the overall deliverable output.

The work have been carried out in close cooperation and synchrony with WP1-2-3, properly integrating the delivered results in a coherent and uniform manner, and then finally releasing the overall BETaaS platform. The progress of these tasks will be one of the major drivers of the work package dedicated to validation trials (WP5).

The document consists of the following chapters:

- Chapter 3, “Platform software development management”, contains the description of the established coordination process of the software development and integration work, in order to effectively perform the technical management of the resources of the platform development team.
- Chapter 4, “Platform technological baseline”, contains the selected reference technological baseline on top of which the different components have been developed and then properly integrated in the BETaaS platform.
- Chapter 5, “Platform implementation”, contains the general specification of the BETaaS platform components implementation. Each component is described through its class diagrams, public interfaces and basic code packaging.
- Chapter 6, “Platform testing”, contains the description of the BETaaS platform testing methodology. It is focused on the unit and integration tests of the BETaaS platform components and basic building blocks.
- Chapter 7, “Platform installation and configuration”, contains the description of the BETaaS platform installation and configuration procedure.
- Chapter 8, “Demonstration application”, contains the description of the initial design of the first BETaaS demonstration application (Intrusion Detection), built on top of the BETaaS platform.
# 2. Table of acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Expanded form</th>
</tr>
</thead>
<tbody>
<tr>
<td>3GPP</td>
<td>The 3rd Generation Partnership Project</td>
</tr>
<tr>
<td>3GPP2</td>
<td>The 3rd Generation Partnership Project 2</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>A-SL</td>
<td>ITU Network application interface</td>
</tr>
<tr>
<td>BD</td>
<td>Big Data</td>
</tr>
<tr>
<td>BETaaS</td>
<td>Build the Environment for the Things as a Service</td>
</tr>
<tr>
<td>CoAP</td>
<td>Constrained Application Protocol</td>
</tr>
<tr>
<td>CON</td>
<td>CoAP Confirmable Message</td>
</tr>
<tr>
<td>CoRE</td>
<td>Constrained RESTful Environments</td>
</tr>
<tr>
<td>D/G/NCS</td>
<td>ETSI Device/Gateway/Network Communication Selection</td>
</tr>
<tr>
<td>D/G/NTM</td>
<td>ETSI Device/Gateway/Network Transaction Management</td>
</tr>
<tr>
<td>DA</td>
<td>ETSI Device Application</td>
</tr>
<tr>
<td>dIa</td>
<td>ETSI Device application interface</td>
</tr>
<tr>
<td>DSCL</td>
<td>Device Service Capability Layer</td>
</tr>
<tr>
<td>D-SL</td>
<td>ITU Device application interface</td>
</tr>
<tr>
<td>DTLS</td>
<td>Datagram Transport Layer Security</td>
</tr>
<tr>
<td>ETSI</td>
<td>European Telecommunications Standards Institute</td>
</tr>
<tr>
<td>G/NAE</td>
<td>ETSI Gateway/Network Application Enablement</td>
</tr>
<tr>
<td>GA</td>
<td>ETSI Gateway Application</td>
</tr>
<tr>
<td>GSCl</td>
<td>ETSI Gateway Service Capability Layer</td>
</tr>
<tr>
<td>G-SL</td>
<td>ITU Gateway application interface</td>
</tr>
<tr>
<td>GW</td>
<td>Gateway</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>IETF</td>
<td>Internet Engineering Task Force</td>
</tr>
<tr>
<td>IoT</td>
<td>Internet of Things</td>
</tr>
<tr>
<td>IoT-A</td>
<td>Internet of Things – Architecture</td>
</tr>
<tr>
<td>IoT-GSI</td>
<td>Internet of Things Global Standards Initiative</td>
</tr>
<tr>
<td>IoT-i</td>
<td>The Internet of Things Initiative</td>
</tr>
<tr>
<td>ISO/OSI</td>
<td>International Organization for Standardization - Open Systems Interconnection</td>
</tr>
<tr>
<td>ITU</td>
<td>International Telecommunication Union</td>
</tr>
<tr>
<td>ITU-T</td>
<td>ITU Telecommunication Standardization Sector</td>
</tr>
<tr>
<td>JCA-IoT</td>
<td>Joint Coordination Activity on Internet of Things</td>
</tr>
<tr>
<td>M2M</td>
<td>Machine to Machine</td>
</tr>
<tr>
<td>mIa</td>
<td>ETSI M2M application interface</td>
</tr>
<tr>
<td>mId</td>
<td>ETSI M2M to device interface</td>
</tr>
<tr>
<td>NA</td>
<td>ETSI Network Application</td>
</tr>
<tr>
<td>NGC</td>
<td>ETSI Network Generic Communication</td>
</tr>
<tr>
<td>NON</td>
<td>CoAP Non-Confirmable Message</td>
</tr>
<tr>
<td>NREM</td>
<td>ETSI Network Remote Entity Management</td>
</tr>
<tr>
<td>NSCL</td>
<td>ETSI Network Service Capability Layer</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
</tr>
<tr>
<td>SCL</td>
<td>ETSI Service Capability Layer</td>
</tr>
<tr>
<td>SLA</td>
<td>Service Level Agreement</td>
</tr>
<tr>
<td>SL-SL</td>
<td>ITU Service layer to service layer interface</td>
</tr>
<tr>
<td>TaaS</td>
<td>Things as a Service</td>
</tr>
<tr>
<td>TISPAN</td>
<td>Telecommunications and Internet converged Services and Protocols for Advanced Networking</td>
</tr>
<tr>
<td>TLS</td>
<td>Transport Layer Security</td>
</tr>
<tr>
<td>VE</td>
<td>IoT-A Virtual Entity</td>
</tr>
<tr>
<td>VE-ID</td>
<td>IoT-A Virtual Entity Identification</td>
</tr>
<tr>
<td>xGC</td>
<td>ETSI Generic Communication</td>
</tr>
<tr>
<td>xSEC</td>
<td>ETSI SECure</td>
</tr>
</tbody>
</table>
3. Platform software development management

In order to effectively perform the technical management of the resources of the platform development team, a proper coordination process of the software development and integration work has been established. It has defined the general integration methodology and the incremental release steps of the BETaaS platform, made available to the validation work-package. It has also setup collaboration tools such as tools for version management and bugs tracking. Finally, a coherent technological foundation software distribution has been provided, which integrates all the specific components into the different incremental releases (e.g. packaging all contributions of all software in a coherent, manageable and releasable set), and then made available to the validation work-packages for related use cases implementation.

3.1. Apache Maven

Apache Maven (http://maven.apache.org/) is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information.

Maven's primary goal is to allow a developer to comprehend the complete state of a development effort in the shortest period of time. In order to attain this goal there are several areas of concern that Maven attempts to deal with:

- Making the build process easy
- Providing a uniform build system
- Providing quality project information
- Providing guidelines for best practices development
- Allowing transparent migration to new features

It also allows to defining dependencies between projects, required libraries and plugins. It executes tests defined inside the code and it will also take care of downloading the jar files from repositories. There are websites where developers can check for available libraries, their version and the POM dependencies that should be included. It is strongly recommended to check websites like http://mvnrepository.com/ for libraries and their available versions.

In BETaaS we've also used a Maven plugin that takes care of packaging the code inside OSGi bundles (see section 4.1).

When using OSGi, like in BETaaS, it's important to remember that not all the Maven libraries are suitable for running as OSGi bundles. A developer should always check that an imported library is available as an OSGi bundle: in any case the OSGi Maven plugin tries to only get the compatible OSGI libraries.

3.1.1. Setting software the Nexus repository for BETaaS and 3th party libraries

In order to tell Maven to use the BETaaS Nexus (http://www.sonatype.org/nexus/) repository, is required a change in the settings.xml file.

Open the settings.xml inside the folder:

  C:/Users/{user}/.m2 For Windows systems
  $HOME/.m2 for Linux environments

If the file is not in these folders, just control under the folder CONF inside the installation of Maven.

Locate the tag <servers>. It can be commented for new Maven installations. In case, you can remove the comment or just set the following information:

```xml
<server>
```
The account is the one used only for downloading artefacts from the Nexus repository. Inside Jenkins (http://jenkins-ci.org/), the repository configuration uses a different account that is enabled to deploy. In this way, it is guaranteed that developers just download the latest stable artefact, and only when a build fully compiles Jenkins updates the bundles with the latest version.

### 3.2. The software development environment

The standard developer environment requires the following components:

- Java Runtime Environment 1.6 or higher
- Eclipse (version 4.2 or higher)
- The following Eclipse Plugins:
  - Maven Plugin M2E, install from the url: m2e http://download.eclipse.org/technology/m2e/releases
  - Apache Maven (version 3.0.3) from http://maven.apache.org/
- A Subversion client, e.g. Tortoise SVN downloadable from: http://tortoisesvn.net/

#### 3.2.1. The software repository

The Subversion (SVN) software repository is accessible by authorized users at the following address: http://www.betaas.eu:8081/svn/BETaaS

Any SVN client may be used to perform browsing, check-out and commit but the recommended tool is the SVN plug-in for Eclipse. Such plug-in allows advanced repository access, synching, file difference editing, conflicts resolution, etc.

The working build directory for any developer is the trunk folder, where the continuous integration process (http://en.wikipedia.org/wiki/Continuous_integration) through Jenkins occurs. When any commit on the trunk is performed by a developer, an automatic platform build process occurs on the trunk. If the build fails, a mail alert is delivered to the software development manager (HP) with the information about the build errors and the related owners and to the development team.

The other root folders under BETaaS (branches and tags) are only managed by the software development manager (HP), for special purposes (e.g. produce an official release of the BETaaS platform).

#### 3.2.2. The software development Redmine sub-project

BETaaS project management is assisted by the Redmine Web Application, available at the following address:

http://2.228.24.139:8080
Authentication is required. Once entered, it is possible to select one among the projects and sub-projects for which the user has been given access by the site administrator.

**BETaaS – Development** is the BETaaS Redmine sub-project focused on development activities. The user can jump to it by means of the drop-down box on the right top of the page, as shown below.

![Figure 1](image1)

Once a **BETaaS – Development** project is selected, the following tabs are available at the top of the page:

- **Overview** – an overview of the subproject and a summary of the open bugs and tasks
- **Activity** – a list of last Redmine users activities performed on the Development sub-project
- **Issues** – this is the place where registered bugs and tasks can be browsed
- **New Issue** – to register new bugs and tasks
- **Documents** – the folder where documents can be uploaded
- **Repository** – this is the Subversion repository browser

### 3.3. The software coding guidelines

This section provides essential guidelines to refer to when coding in Java in order to produce uniform code among different components\(^1\).

Code documentation is based on **Javadoc**\(^2\).

#### 3.3.1. Lines Length and Indentation

Lines should be no longer than **80 characters**. Indentation should be inserted as **two spaces**.

#### 3.3.2. Implementation Comments

Implementation comments are marked by the \/// sequence and should be used just to comment out sections of unused code or to document the way a specific implementation has been done (so that the comments shouldn’t belong to general documentation).

#### 3.3.3. Source File Header

Each Java source file should begin with the following header, by customizing the fields in bold:

```java
// BETaaS - Building the Environment for the Things as a Service
//
// Component: <component>
// Responsible: <responsible>
```

After the header, each file should contain the **package statement**, followed by the **import statements** and the class/interface declaration.

The file version is not included since it is managed by SVN.

---

\(^1\) See also [http://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html](http://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html)

### 3.3.4. Class and Interfaces Declaration

Each class and interface declaration should be written according to the following structure:

```java
/**
 * <Description of the class/interface, its intended use, its placement within the architecture (e.g. the component to which it belongs, the implemented interfaces from the API doc, etc.)>
 * @author <author>
 */

Class/Interface statement {

// PUBLIC SECTION
Constructors
Static variables
Final variables
Instance variables
Methods

// PROTECTED SECTION
Static variables
Final variables
Instance variables
Methods

// PRIVATE SECTION
Static variables
Final variables
Instance variables
Methods
}
```

Methods with the same visibility should be grouped **functionally**.

Each member variable should be documented as well as all methods. In particular methods documentation should include the following Javadoc tags:

- `@param`
- `@return`
- `@throws`

### 3.3.5. Naming

Packages must be named in lowercase letters (e.g. `eu.betaas.service.bigdatamanagement.core.services`). Interfaces and Classes name should be **nouns** in *CamelCase* letters (e.g. `ServiceProvider`).

Methods name should be in *CamelCase*, beginning with a **verb** in lowercase letter (e.g. `computeQoS`). Setter and getter methods should begin with `set` and `get` respectively (e.g. `getName()`, `setLevel()`), while getter methods for boolean variables should begin with `is` (e.g. `isVisible()`).

Member variables should be in *CamelCase* letters, starting with `m` (e.g. `mObjectList`). This helps distinguishing them from local scope variables, performing quicker automatic completion, etc.

Variables should be in *CamelCase* letters (e.g. `listIterator`).

Constants should be named in uppercase letters, with an “_” between consecutive words (e.g. `MAX_LIST_ELEMENTS`).

### 3.3.6. Wrapping Lines

When possible, long lines should be wrapped using the following rules:

- Break before operators
• Break before commas
• Align new lines to the beginning of the expression of the same level
• Divide logical conditions (e.g. within if statements) into different lines according to parenthesis levels. For example:

```cpp
if (condition1 &&
    (condition2 || (condition3 &&
                  condition4)))
```

• Prefer breaking higher level parenthesis than lower level ones. For example prefer:

```cpp
if (condition1 &&
    (condition2 || condition3))
```

instead of

```cpp
// TO AVOID
if (condition1 && (condition2 ||
                   condition3))
```

### 3.3.7. Variables

Variables should be declared at the beginning of the block (delimited by curly braces). One variable per line is preferred, along with its initialization and in-line brief documentation. For example:

```cpp
{
    int listSize=0; // description of x
}
```

### 3.3.8. Scope

Always try to give methods and variables the least possible visibility (from private to protected and public).
4. Platform technological baseline

In this chapter we describe the selected reference technological baseline on top of which the different components have been developed and then properly integrated in the BETaaS platform (this baseline have been fundamental in order to have guaranteed a successful integration).

The platform components have been built on top of relevant existing off-the-shelf and best of breed software platforms, with particular focus on open source frameworks and/or products. Although the BETaaS architecture and framework is general, a practical approach has been used in the software development and integration to ensure that the platform is able to operate with full capabilities on at least one concrete practical implementation.

All the fundamental topics are described in the following.

4.1. OSGI

The OSGI technology [http://www.osgi.org](http://www.osgi.org) has been selected as major cornerstone of the BETaaS platform implementation.

OSGI is a set of specifications that define a dynamic component system for Java. These specifications reduce software complexity by providing a modular architecture for large-scale distributed systems as well as small, embedded applications. They enable a development model where applications are dynamically composed of many different reusable components. The OSGI specifications enable components to hide their implementations from other components while communicating through services, which are objects that are specifically shared between components.

OSGI is based on a layered model that is depicted in the following figure:

![Figure 2](image-url)

The following contains a short definition of the terms showed in the previous figure:

- **Bundles**: bundles are the OSGI components made by the developers;
- **Services**: the services layer connects bundles in a dynamic way by offerings a publish-find-bind model for plain old Java objects;
- **Life-Cycle**: the API to install, start, stop, update and uninstall bundles;
- **Modules**: the layer that defines how a bundle can import and export code;
- **Security**: the layer that handles the security aspects;
Execution Environment: defines what methods and classes are available in a specific platform

In practice, bundles are actually jar files with a manifest file that specifies meta-information about the bundle itself (e.g.: library exported and imported version and name).

The most important OSGI implementations are Felix, Equinox and Knopflerfish.

The current version of the BETaaS platform supports version 4.2 of the OSGi specification inside an Apache Karaf Container (see section 4.2), and is compatible with all the three current major and most important OSGI implementations: Apache Felix, Eclipse Equinox and Knopflerfish. In particular, the initial focus has been on Equinox version 3.6.0.

In the following sections we describe the fundamental principles that have driven the BETaaS platform OSGI-based implementation.

4.1.1. Importing the BETaaS baseline software

In order to import the BETaaS baseline software, first checkout the BETaaS SVN repository, then, from Eclipse, imports an Existing Maven project.

Choose the “betaas” directory under the SVN trunk and import the superpom \(^3\) together with all the existing subprojects.

---

\(^3\) The superpom is the POM xml file inside the “betaas” folder.
Inside the Eclipse workspace you now will have many different folders, the one with the superpom and one for each of the subprojects (e.g.: betaas-service, betaas-taas, betaas-adaptation, etc.). If you are not interested in all the projects, you can avoid importing them on Eclipse, but you will need in any case to run a “mvn install” from the superpom directory.

### 4.1.2. Creating a new OSGI bundle with Eclipse and Maven

BETaaS is structured as four major projects linked together by a super POM. The super POM contains the common dependencies, used by the projects, with their version. Each project defines its dependencies and its bundles. When a component is created inside one of these projects, it uses the project’s POM as parent POM.

The four main projects containers are:

- **betaas-common** for components that do not fit the other three layer of BETaaS
- **betaas-service** for Service Layer
- **betaas-taas** for TaaS Layer
- **betaas-adaptation** for Adaptation Layer

Every software component, for example the Big Data Manager at Service layer, must import the parent project POM.

First create a new Maven project.
No special archetype is required so you can create a simple project without archetype. Be careful to specify a new folder inside its parent container (in our case since we will create the betas-service-bigdatamanager folder).

Edit the setting of the new projects according to the following recommendations:

**groupId**: eu.betaas

**artifactId**: betaas-component-subcomponent

In this case *component* can be one of: service, taas or adaptation; while the *subcomponent* can be one of the components represented in Figure 10 and section 4.1 and 4.2 of the architecture document D3.1.1. In our case it is the Big Data Manager of the Service layer. Moreover, the general principle is that each component of the architecture has a bundle at each level in which its functionalities are needed (and then specialized depending on the level itself), e.g.: Big Data Manager will have 2 bundles, one at level service and one at level taas (assuming that the Big Data Manager has no functionalities at level adaptation, otherwise the BDM bundles would be 3); ServiceManager and InstanceManager will have only 1 bundle each at level service, if they provide functionalities only at such level.

**packaging**: it is a jar, but later we will edit it to be bundle, once we have specified the OSGi Maven plugin.

**name**: it should be a human readable string that describes the component name.

**description**: it should be a human readable string that describes the component.
Also remember to add the dependency inside the parent's POM: in our case *betas-service* POM.

```xml
<modules>
  <module>betaas-service-bigdatamanager</module>
</modules>
```

This will tell to the parent project to build your project too.

The actual POM should look like this one:

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>eu.betaas</groupId>
    <artifactId>betas-service</artifactId>
    <version>0.0.1-SNAPSHOT</version>
  </parent>
  <groupId>eu.betaas</groupId>
  <artifactId>betas-service-bdm</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>BETaaS Service Big Data Manager</name>
  <description>BETaaS Service Layer Big Data Manager</description>
</project>
```

To build the project as OSGi bundle, a specific Maven plugin is required. Just add the following lines inside the POM file, to include the OSGi Maven plugin:
Remember to change PACKAGEQNAME to reflect your effective package qualify name and specify a `<Bundle-Activator>` if your bundle provides an activator class. This information should be inserted after the `<dependencies>` inside the `<project>` xml element. In our example `eu.betaas.service.bigdatamanager` will be enough.

The generic convention is: `eu.betaas.component.subcomponent`. They have to reflect the artefact ID used for the bundle. In our case the component is the service and the subcomponent is big data manager. The activator class is the entry point of the bundle and will be named `<subcomponent>Activator`. In our case `BigDataManagerActivator`.

While we just specified the plugin Maven has to use to produce the bundle file, we still miss the OSGI dependency. Without this, we will miss the required interface that our activator requires. Insert inside the `<dependencies>` tag the following lines:

```xml
<dependency>
  <groupId>org.apache.felix</groupId>
  <artifactId>org.osgi.core</artifactId>
</dependency>
```

Now it is possible to create the class `BigDataManagerActivator`. Just create a new class that extends the BundleActivator with the correct qualified name (`eu.betaas.component.subcomponent`).
At the beginning of the POM file also remember to check the following bundle attributes (some of these are already created for you by Eclipse):

```xml
<groupId>eu.betaas</groupId>
<artifactId>eu.betaas.service.bdm</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>bundle</packaging>
<name>BETaaS Service Big Data Manager</name>
<description>BETaaS Service Layer Big Data Manager</description>
```

The `groupId` identifies the entity which creates the artefact. The bundle file will have a file name in the form of `<artifactId>-<version>.jar`. Do not forget to specify the tag `<packaging>` which is not added automatically and is specific for the OSGi Maven plugin.

Finally the Maven project can be compiled from the shell line with an "mvn clean install" command, launched at the same level of the POM file. As alternative you can right click the project that you want to build and run as “maven install” from Eclipse. Remember to set exports and imports with `<Export-Package>` and `<Import-Package>`; in this way your bundle does not miss required dependencies and export the required libraries for other bundle to use. You can find them inside the `<plugins>` tags.

At this point the bundle has been compiled and you should have seen the following similar output:

```
[[INFO] ----------------------------- Reactor Summary:]
[INFO] BETaaS Super POM .................. SUCCESS [0.592s]
[INFO] BETaaS Common Components ........... SUCCESS [0.102s]
[INFO] BETaaS Service .................. SUCCESS [0.091s]
[INFO] BETaaS BigData Manager Service ....... SUCCESS [0.096s]
[INFO] BETaaS BigData Manager Core Service .... SUCCESS [7.837s]
[INFO] BETaaS Instance Manager .......... SUCCESS [2.371s]
[INFO] BETaaS Service Manager ........... SUCCESS [1.218s]
[INFO] BETaaS TaaS .................. SUCCESS [0.155s]
```
4.1.3. Dependencies that are already defined as superpom level

In order to avoid importing different libraries versions inside the platform, some common libraries dependencies are already defined inside the superpom. These libraries are:

```xml
<dependency>
    <groupId>org.apache.felix</groupId>
    <artifactId>org.osgi.core</artifactId>
    <version>${osgi.version}</version>
</dependency>

<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>${log4j.version}</version>
</dependency>

<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>${junit.version}</version>
    <scope>test</scope>
</dependency>
```

In order to import one of these libraries, there is no need to include the version, in fact Maven will use as version the one included in the superpom. For example to include the Log4j library, use the following dependency inside the POM file.

```xml
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
</dependency>
```

4.1.4. How to run and debug bundles on Eclipse

In order to run the OSGI console from Eclipse, just create a new run configuration (or a debug configuration if you need to debug a bundle).
Click on Run-> New Configuration from the Eclipse IDE and then select a new configuration under the OSGi Framework configuration. Select the bundles displayed on the above figure. The basic bundles are required to run Equinox console:

- `org.apache.felix.gogo.command`
- `org.apache.felix.gogo.runtime`
- `org.apache.felix.gogo.shell`
- `org.eclipse.equinox.console`
- `org.eclipse.osgi`

Save the configuration and then run it. Inside the Eclipse console the framework will run. You should see a "osgi>" symbol. To check that the framework is running and the bundles are loaded run a "ss" and press <enter>. This will display the current state of bundles in the following way:

```
osgi> ss
"Framework is launched."
```

<table>
<thead>
<tr>
<th>id</th>
<th>State</th>
<th>Bundle</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ACTIVE</td>
<td>org.eclipse.osgi_3.9.0.v20130529-1710</td>
</tr>
<tr>
<td>6</td>
<td>ACTIVE</td>
<td>org.apache.felix.gogo.command_0.10.0.v201209301215</td>
</tr>
<tr>
<td>7</td>
<td>ACTIVE</td>
<td>org.eclipse.equinox.console_1.0.100.v20130429-0953</td>
</tr>
<tr>
<td>8</td>
<td>ACTIVE</td>
<td>org.apache.felix.gogo.runtime_0.10.0.v201209301036</td>
</tr>
<tr>
<td>9</td>
<td>ACTIVE</td>
<td>org.apache.felix.gogo.shell_0.10.0.v201212101605</td>
</tr>
</tbody>
</table>

The output will show all the bundles in state ACTIVE.

Now to install and run your bundle you need to note where it is currently stored. If you compiled correctly the Maven project, you should have the jar file under your project’s target directory. The absolute path should be used to run the following command inside the console:
install file:PATH/TO/JAR/FILE/eu.betaas.service-bdm_0.0.1.SNAPSHOT.jar

Now running "ss" should generate the following input.

```
osgĩ> ss
"Framework is launched."
```

<table>
<thead>
<tr>
<th>id</th>
<th>State</th>
<th>Bundle</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ACTIVE</td>
<td>org.eclipse.osgi_3.9.0.v20130529-1710</td>
</tr>
<tr>
<td>6</td>
<td>ACTIVE</td>
<td>org.apache.felix.gogo.command_0.10.0.v201209301215</td>
</tr>
<tr>
<td>7</td>
<td>ACTIVE</td>
<td>org.eclipse.equinox.console_1.0.100.v20130429-0953</td>
</tr>
<tr>
<td>8</td>
<td>ACTIVE</td>
<td>org.apache.felix.gogo.runtime_0.10.0.v201209301036</td>
</tr>
<tr>
<td>9</td>
<td>ACTIVE</td>
<td>org.apache.felix.gogo.shell_0.10.0.v201212101605</td>
</tr>
<tr>
<td>13</td>
<td>RESOLVED</td>
<td>eu.betaas.service-bdm_0.0.1.SNAPSHOT</td>
</tr>
</tbody>
</table>

Your bundle is currently installed but is not running. To start the bundle run:

```
osgi> start 13
Bundle started successfully
```

In our case OSGi assigned 13 as id for the bundle. Fix the id accordingly with the specific id assigned by your system. Now the “ss” command will show the bundle as ACTIVE.

```
osgi> ss
"Framework is launched."
```

<table>
<thead>
<tr>
<th>id</th>
<th>State</th>
<th>Bundle</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ACTIVE</td>
<td>org.eclipse.osgi_3.9.0.v20130529-1710</td>
</tr>
<tr>
<td>6</td>
<td>ACTIVE</td>
<td>org.apache.felix.gogo.command_0.10.0.v201209301215</td>
</tr>
<tr>
<td>7</td>
<td>ACTIVE</td>
<td>org.eclipse.equinox.console_1.0.100.v20130429-0953</td>
</tr>
<tr>
<td>8</td>
<td>ACTIVE</td>
<td>org.apache.felix.gogo.runtime_0.10.0.v201209301036</td>
</tr>
<tr>
<td>9</td>
<td>ACTIVE</td>
<td>org.apache.felix.gogo.shell_0.10.0.v201212101605</td>
</tr>
<tr>
<td>13</td>
<td>ACTIVE</td>
<td>eu.betaas.service-bdm_0.0.1.SNAPSHOT</td>
</tr>
</tbody>
</table>

The bundle has now been installed and deployed correctly.

4.2. Apache Karaf

Apache Karaf is a small OSGi based runtime environment which provides a lightweight container onto which various components and applications can be deployed. It has been selected within BETaaS in order to mainly support the scalability requirements that have to be supported by the BETaaS platform, with particular focus on its test and deployment aspects.

Karaf supports different OSGi containers and also provides advanced features such as OSGi Blueprint and OSGi Ctx. It also provides a console and many features that can be deployed on its runtime environment.

Here is a short list of the main features supported by Karaf:

- **Hot deployment**: Karaf supports hot deployment of OSGi bundles by monitoring jar files inside the [home]/deploy directory. Each time a jar is copied in this folder, it will be installed inside the runtime. It is possible to update or delete it and changes will be handled automatically. In addition, the Karaf also supports exploded bundles and custom deployers (blueprint and spring ones are included by default).
- **Dynamic configuration**: services are usually configured through the ConfigurationAdmin OSGi service. Such configuration can be defined in Karaf using property files inside the [home]/etc directory. These configurations are monitored and changes on the properties files will be propagated to the services.
- **Logging System**: using a centralized logging back end supported by Log4J, Karaf supports a number of different APIs (JDK 1.4, JCL, SLF4J, Avalon, Tomcat, OSGi)
- **Provisioning**: provisioning of libraries or applications can be done through a number of different ways, by which they will be downloaded locally, installed and started.
- **Native OS integration**: Karaf can be integrated into a Operating System as a service so that the lifecycle will be bound to such Operating System.
- **Extensible Shell console**: Karaf features a nice text console where it is possible to manage the services, install new applications or libraries and manage their state. This shell is easily extensible by deploying new commands dynamically along with new features or applications.
- **Remote access**: use any SSH client to connect to Karaf and issue commands in the console
- **Security framework based on JAAS**: Karaf provides simple commands for managing multiple instances. It is possible to easily create, delete, start and stop instances of Karaf through the console.
- **Support of the latest OSGi 4.2 containers**: Apache Felix Framework 3.0 and Eclipse Equinox 3.6

Karaf features are also based on a layered model that is depicted in the following figure:

![Layered Model of Karaf](http://karaf.apache.org/index/community/download.html)

Figure 11

### 4.2.1. Installing Karaf

In order to start with Karaf, it is necessary to download the version 2.3.2 from the official website[^1]. The installation is quite simple and requires only unzipping the downloaded file. To start the framework, it is necessary to just launch `karaf` from its bin folder. You should have the following output:

```
C:\BETaaS\apache-karaf-2.3.2\bin>karaf.bat

Apache Karaf (2.3.2)
Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
Hit 'ctrl-d' or type 'osgi:shutdown' or 'logout' to shutdown Karaf.
karaf@root
```

Figure 12

### 4.2.2. Running Karaf

In order to run Karaf, just launch karaf.bat or karaf.sh.

If you see a message like:

[^1]: [http://karaf.apache.org/index/community/download.html](http://karaf.apache.org/index/community/download.html)
it means that inside your configuration `org.apache.karaf.management` you should change the `rmiServerHost` and `rmiRegistryHost` to `localhost` or to a different address than `0.0.0.0`.

### 4.2.3. Configuring Karaf

Before starting deployment with Karaf, some settings need to be configured. First the container should be chosen so that it will run Equinox.

Locate the file `config.properties` under the `etc` folder where Karaf is installed and change the following line:

```
karaf.framework=felix
```

to:

```
karaf.framework=equinox
```

Now we will make an example of leveraging the global log configuration file of Karaf. This file is `org.ops4j.pax.logging.cfg` in the folder: `{karaf.install.dir}/etc`.

Using this file makes the management of logging easier than placing a configuration file inside every bundle. As default, Karaf do not log on console but only on files. This behaviour can be changed within this file. For example we can specify to have enabled only ERROR log and also the stdout for the rootlogger.

```
# Root logger
log4j.rootLogger = ERROR, stdout
```

Now we can enable the output to console and set it to warning:

```
# CONSOLE appender not used by default
#log4j.appender.stdout.treshold=WARN
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.org.apache.log4j.ConsoleAppender
log4j.appender.stdout.org.apache.log4j.PatternLayout
log4j.appender.stdout.org.apache.log4j.PatternLayout.ConversionPattern=%d{ISO8601} | %d{yyyy-MM-dd HH:mm:ss} | %-5.5p | %-16.16t | %-32.32c{1} | %-32.32C %4L | %X{bundle.id} - %X{bundle.name} - %X{bundle.version} | %m%n
```

And we can setup our log:

```
log4j.logger.betaas=INFO, stdout, betaasfilelog
```

We setup a logger for every class that want to use it. Inside the Java code, while you instantiate your logger, just specify betas as parameter:

```
Logger.getLogger("betaas");
```

This will bind your logger to the BETaaS settings inside the configuration file.

Also add the required dependency on maven to have the library available.
If a new log appender for BETaaS is needed to be included in the configuration, just add a new log appender in the following way:

File appender for BETaaS Taas

```xml
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
</dependency>
```

In this example we added a new appender for the BETaaS Taas bundles. We also specified that the level is INFO and that every package that will instantiate the logger class inside the code by using the string `betas.taas` will log on this appender. Adding the following log initialization code inside a bundle will ensure that it will use the appender that we just specified.

```java
Logger.getLogger("betas.taas");
```

We just told to the global configuration file that every bundle that instantiates the logger to use this appender will be redirected to `betaastaasfilelog`, but we didn’t yet tell how this log should work. To do this we add, inside the global configuration file the following settings:

```properties
# ensure that only the desired class log on this appender
log4j.additivity.betaas.taas=false
```

We remove the addition to ensure that only the bundle that instantiates the log component with the code that we introduced before writes to this appender. Others BETaaS bundles that do not belong to ‘betas.taas’ can use other appenders.

To finish the settings for our log we need to specify how the `betaastaasfilelog` works:

```xml
log4j.appender.betaastaasfilelog=org.apache.log4j.RollingFileAppender
log4j.appender.betaastaasfilelog.layout=org.apache.log4j.PatternLayout
log4j.appender.betaastaasfilelog.threshold=INFO
log4j.appender.betaastaasfilelog.layout.ConversionPattern=%d{ISO8601} | %d{yyyy-MM-dd HH:mm:ss} | %X{bundle.id} - %X{bundle.name} - %X{bundle.version} | %m%n
log4j.appender.betaastaasfilelog.file=${karaf.data}/log/karaf_betaas_taas.log
log4j.appender.betaastaasfilelog.append=true
log4j.appender.betaastaasfilelog.maxFileSize=1MB
log4j.appender.betaastaasfilelog.maxBackupIndex=10
```

These settings will write the log inside the file `${karaf.data}/log/karaf_betaas_taas.log`.

In similar way we can define also logs for others BETaaS layers:

File appender for BETaaS Service

```xml
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
</dependency>
```

```java
Logger.getLogger("betas.service");
```

```properties
# ensure that only the desired class log on this appender
log4j.additivity.betaas.service=false
```

```xml
log4j.appender.serviceLogger=org.apache.log4j.RollingFileAppender
log4j.appender.serviceLogger.layout=org.apache.log4j.PatternLayout
log4j.appender.serviceLogger.threshold=INFO
log4j.appender.serviceLogger.layout.ConversionPattern=%d{ISO8601} | %d{yyyy-MM-dd HH:mm:ss} | %X{bundle.id} - %X{bundle.name} - %X{bundle.version} | %m%n
log4j.appender.serviceLogger.file=${karaf.data}/log/karaf_betaas_service.log
```
Finally, we need to setup a web console for Karaf. This feature can be installed by running Karaf and installing a feature. In fact Karaf deploys extensions by installing features.

```
karaf@root> features:install webconsole
```

Figure 13

Now the console will be available at:

```
http://localhost:8181/system/console
```

with karaf as both login and password (as default from installation).

### 4.2.4. Using Apache Blueprint for deployment of OSGi services

A powerful capability of Apache Karaf is to leverage the OSGi Blueprint through Apache Aries. This bundle is installed in the runtime container and allows the creation and deployment of OSGi services through an xml file. Once the interface of the service has been declared, with the xml file it can be easily deployed as a service without the need of registering the service with a bundle activator. Also, Maven Blueprint service archetype can be used to generate the structure of the service.

```
mvn archetype:generate
-DarchetypeGroupId=org.apache.karaf.archetypes
-DarchetypeArtifactId=karaf-blueprint-archetype
-DarchetypeVersion=2.3.2
-DgroupId=eu.betaas
-DartifactId=betaas.example.logger
-Dversion=0.0.1-SNAPSHOT
-Dpackage=betaas.example.logger.blueprint
```

In this case Maven will build the structure of a package that deploys a service with blueprint. Inside the POM file, you will need to add the BETaaS dependencies.

While this method easily creates the structure of a blueprint service OSGi bundle, many rework has to be done for integrating the project with the remaining BETaaS architecture.

If you prefer, you can generate the Maven project following the previous instructions and then remembering to add an export for your service class: without this you will not be able to use the service from another bundle.

```
<Export-Package>
  betaas.example.logger.blueprint
</Export-Package>
```
The central element of Blueprint is the xml file that tells how to build the service and inject class and parameters\(^5\). It is stored inside the OSGI-INF\[blueprint\] folder of the project (or you can create this path inside your project if you choose to not create it from a maven archetype).

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0" default-activation="lazy">
  <bean id="serviceBean" class="betaas.example.logger.blueprint.MyCustomLogServiceImpl" init-method="setLogger"/>
  <service ref="serviceBean" interface="betaas.example.logger.blueprint.ILogService"/>
</blueprint>
```

In our example we created a bean `serviceBean` from the class `MyCustomLogServiceImpl` that extends `ILogService`. In OSGi practice, a service is created from an abstract interface so that it can be extended by other bundle with different implementations. In this case we also specified an init method. This method is called when service is installed and run. For example it can be used to ensure that the service initializes the required class. Also Blueprint allows you to inject properties and dependencies.

In our case we are going to define the `ILogService` interface so that it provides the following methods:

```
package betaas.example.logger.blueprint;

public interface ILogService {
  public void setLogger();
  public String echo(String message);
}
```

The method `setLogger` is used to initialize the logger class, while `echo` print a message. This service is just an example to demonstrate how to generate service with OSGi Blueprint.

We implement our `MyCustomLogServiceImpl` in this way:

```
package betaas.example.logger.blueprint;

import java.util.Vector;
import org.apache.log4j.Logger;

public class MyCustomLogServiceImpl implements ILogService {
  private Logger logger;

  public void setLogger() {
    logger = Logger.getLogger("MyCustomLogServiceImpl");
  }

  public String echo(String message) {
    return logger.info("Echoing: "+message);
  }
}
```

\(^5\) For more example refer to [http://aries.apache.org/modules/blueprint.html](http://aries.apache.org/modules/blueprint.html)
private Vector<String> components;

public void setLogger(){
    logger = Logger.getLogger("betaas");
    components = new Vector<String>();
}

public String echo(String message) {
    System.out.println("Invoked echo Logger");
    logger.info(message);
    return "Echo processed: " + message;
}

We simply init the class to use a logger that will take the 'betaas' settings from the configuration file. Then you've to just compile your bundle with Maven and it will be ready to be deployed in a runtime environment, like Karaf, that supports Blueprint services.

4.2.5. Using Whiteboard pattern for OSGi

Let's see how to consume the service created with OSGi Blueprint. There are many ways in which OSGi services can be consumer. A plain implementation requires:

- A service implementation, like our ILogService bundle
- A class that implement a BundleActivator that start the service
- A bundle that consume the service

Fortunately OSGi Blueprint takes care of the first two parts. When our ILogService will start a Blueprint it will instantiate and register itself in the OSGi service registry, so that other bundles will be able to use it. The problem with OSGi is that services can disappear and appear at runtime and to leverage this capability in BETaaS we need to listen for service registration. An OSGi pattern that allows reducing the code and the complexity of application is called Whiteboard\(^6\). A bundle that consumes a service, do not simply search it inside the registry but it registers itself as consumer of the service. Whenever the service is started, the bundle is notified and can take actions; also when the service is stopped it will be informed. This pattern is also simpler than the listener pattern that requires services to control the listeners increasing the level of dependency between bundles. In our case the consumer will react only if the service is registered, and the service will not need to know who is going to consume it.

Again, we remind that is fundamental that a service bundle exports the service class and the consuming bundle imports these classes. In our case inside the POM file we specify the import of the service class.

We also import the `org.osgi.util.tracker` because we will need it to implement the whiteboard pattern.

We can create the class that will consume the service. It will extend a BundleActivator.

For example, we want our bundle to use the registered services every five seconds. We also plan to use a thread for this purpose.

```java
public class BigDataManagerActivator implements BundleActivator, Runnable

We will start the thread when the service will be detected. For this reason, we use a ServiceTracker.

```java
public void start(BundleContext context) throws Exception {
    tracker = new ServiceTracker<>(context, ILogService.class.getName(), null);
    tracker.open();
    thread = new Thread(this, "LogService Whiteboard");
    thread.start();
}
```

The thread every five seconds gets the list of registered services and logs a simple text.

```java
public synchronized void run() {
    Thread current = Thread.currentThread();
    int n = 0;
    while (current == thread) {
        Object[] providers = tracker.getServices();
        if (providers != null && providers.length > 0) {
            if (n >= providers.length)
                n = 0;
            ILogService cp = (ILogService) providers[n++];
            System.out.println("WHITEBOARD " + cp.echo("Test registered service"));
        }
        try { wait(5000); } catch (InterruptedException e) {} 
    }
}
```

In this case, for each service registered the ServiceTracker will execute the method echo. Be careful to check that the service is still not null and to use the generic interface `ILogService` and not a specific implementation. In our case, both interfaces are provided by the same bundle, but OSGi offers to other bundles the opportunity to implement the same service in a different way.
4.2.6. Deploy Bundles on Apache Karaf

Now that two bundles are available, once we’ve built and installed them with Maven we can take advantage of the Karaf deployer to load bundles into the runtime environment.

First, let’s start Karaf.

The syntax to install a bundle from Maven repository is:

```osgi
osgi:install mvn:GROUPID/ARTIFACTID/VERSION
```

In our case we can start to install the two bundles:

```osgi
osgi:install mvn:eu.betaas/betaas.example.logger/0.0.1-SNAPSHOT
```

![Figure 14](image)

Our logger service is now installed. We can also check that is registered under bundles. The ‘list’ command will show it. The bundle is installed but not yet started.

![Figure 15](image)

Now let’s start our consumer bundle.

```osgi
install mvn:eu.betaas/betaas-service-bigdatamanager/0.0.1-SNAPSHOT
```

The two bundles will be shown as installed.
Figure 16

Under the web console we can check that the two bundles are installed. Some details like export and imports are shown as well as if they are resolved or not.

Figure 17

To start our bundle logger, we need to specify its ID. In our case Karaf assigned 84, but it can change.

Figure 18

Also BETaaS Data Manager is now resolved because its dependencies are available. Our service will start to be used every five seconds by our Data Manager bundle.

Figure 19

If we stop the bundle 84, then we will see our service not logging anymore.
An important feature of Karaf is the support for Distributed OSGi (DOSGi). Under this specification, an OSGi service bundle can be shared among different containers. In fact DOSGi will take care of generating a web service for the service interface to enable a remote call of its methods. While this feature is useful to call bundles from external applications, by using DOSGi the OSGi framework can track service in remote containers, without any change in the way the traditional OSGi services work. In fact when a bundle requests the available services list, it will be able to retrieve remote bundles without any change in its code behaviour, DOSGi will take care of calling the remote bundles and returning their results to the local container. The only apparent negative side of DOSGi is that it requires to configure the address of the remote container that exposes bundles, but in order to overcome this limitation another feature available in Karaf can be used: Apache Zookeeper⁷. This component allows exploring dynamically bundles by acting as a central repository of the configurations. The containers do not need to know each other’s IP addresses but only contact the Zookeeper server to register their DOSGi bundles. In this way is possible to dynamically discover bundles and services.

If you want to install Zookeeper by using the feature of Apache Karaf that does not require downloading and installing any additional software, you can execute the following commands:

```
features:chooseurl cxf 2.7.2
features:chooseurl cxf-dosgi 1.5.0
features:install cxf-dosgi-discovery-distributed cxf-dosgi-zookeeper-server
```

⁷ [http://www.slideshare.net/sauravhaloi/introduction-to-apache-zookeeper](http://www.slideshare.net/sauravhaloi/introduction-to-apache-zookeeper)
You need also to configure the Zookeeper server. To configure the port used by the client to connect:

```
config:propset -p org.apache.cxf.dosgi.discovery.zookeeper.server clientPort 2181
```

In case you want just to configure your Apache Karaf to connect to a server, you’ve to use only these commands:

```
features:chooseurl cxf 2.7.2
features:chooseurl cxf-dosgi 1.5.0
features:install cxf-dosgi-discovery-distributed
```

In the clients that need to access Zookeeper, it must be configured the Zookeeper port and the IP address of the server:

```
config:propset -p org.apache.cxf.dosgi.discovery.zookeeper zookeeper.port 2181
config:propset -p org.apache.cxf.dosgi.discovery.zookeeper zookeeper.host=x.x.x.x
```

Now your Karaf is available for deploying DOSGi bundles with dynamic discovery. To expose the interfaces of a service using DOSGi, it must be added the following code in the activator class when registering a service:

```java
public void start(BundleContext bc) throws Exception {
    Dictionary<String,String> props = new Hashtable<String,String>();
    props.put("service.exported.interfaces", "*");
    props.put("service.exported.configs", "org.apache.cxf.ws");
    props.put("org.apache.cxf.ws.address", "http://localhost:18900/bdm-service");

    reg = bc.registerService(IBigDataDistributedCoreService.class.getName(),
                             new BDDistributedCoreServiceImpl(), props);
}
```

The previous code will expose the interface of the BDDistributedCoreServiceImpl service class through a web service. The WSDL can be checked at the url `http://localhost:18900/bdm-service?wsdl`. While the web service generated by DOSGi can be also used by non OSGi external applications, inside the framework the bundle service will be exposed without the need of calling the web service but just using the traditional service registry like a local service. The properties interfaces control which interface are exposed by DOSGi and act as a filter.

DOSGi can be also used for Blueprint service without using bundle activator; the properties can then be injected in the following way:

```xml
<bean id="serviceDistributedBDCoreBean" class="eu.betaas.service.bigdatamanager.core.service.impl.BDDistributedCoreServiceImpl"/>

<service ref="serviceDistributedBDCoreBean" interface="eu.betaas.service.bigdatamanager.core.service.IBigDataDistributedCoreService">
    <service-properties>
        <entry key="service.exported.interfaces" value="*"/>
    </service-properties>
</service>
```
In the case of Blueprint, the properties are injected by using the service-properties tag.

### 4.2.8. Deploy Data Service on Apache Karaf

In order to deploy the database service, you need two files from the SVN repository. The files are:
- `feature-taasdb.xml` (under trunk\betaas\betaas-configuration\features)
- `taasdb.cfg` (trunk\betaas\betaas-configuration\configuration)

The `cfg` file needs to be placed inside the current Karaf installation directory under `etc` folder. Also it should be edited to reflect the desired configuration for the database.

The `feature-taasdb.xml` installs the bundles required by the database service; in order to install the bundle run from Karaf shell:

```
features:addUrl file:{path-to-xml-file}/feature-taasdb.xml
```

If everything is fine, running:

```
features:list
```

it will return a list of available features that will include the Betaas Database.

<table>
<thead>
<tr>
<th>State</th>
<th>Version</th>
<th>Name</th>
<th>Repository</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[uninstalled]</td>
<td>[0.0.1-SNAPSHOT]</td>
<td>taas-database</td>
<td>repo-0</td>
<td></td>
</tr>
</tbody>
</table>

Then the database will be available and can be installed with:

```
features:install taas-database
```

A `feature file` is a provisioning file that tells to Karaf about a set of bundles that are required to be installed in order to implement a specific feature. This mechanism can be reused also to install other BETaaS functionalities.

Further details about how the database configuration works are described in the next section.

### 4.2.9. Provide bundles with configuration parameters

Keeping configuration parameters inside a bundle, in the form of configuration files, is not advisable because it requires opening the bundle and editing the information. Instead of this solution, Karaf offers a service called `configuration admin`. This service can be used together with Blueprint to inject configuration parameters in a bundle by reading files located in the Karaf `etc` directory. The feature `taas-database` uses this mechanism and provides the configuration to the database bundle through the file `taasdb.cfg`. This file defines the following parameters:

- `jdbc = jdbc:h2:file:C:/betaas/data/taasbmdmb;DB_CLOSE_DELAY=-1`
- `user = sa`
- `pwd = sa`
The blueprint xml file for the bundle is the following:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!--
Licensed to the Apache Software Foundation (ASF) under one or more
ccontributor license agreements. See the NOTICE file distributed with
this work for additional information regarding copyright ownership.
The ASF licenses this file to You under the Apache License, Version 2.0
(the "License"); you may not use this file except in compliance with
the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.

requires that install mvn:com.h2database/h2/1.3.163 is installed
-->
<blueprint default-activation="eager"
xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:jpa="http://aries.apache.org/xmlns/jpa/v1.0.0"
xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.1.0"
xmlns:tx="http://aries.apache.org/xmlns/transactions/v1.0.0">

<cm:property-placeholder persistent-id="taasdb" update-strategy="reload" />
<cm:default-properties>
<cm:property name="jdbc" value="jdbc:h2:mem:dbtaas" />
<cm:property name="user" value="sa" />
<cm:property name="pwd" value="sa" />
</cm:default-properties>
</cm:property-placeholder>

<bean id="serviceDatabaseBean"
class="eu.betaas.taas.bigdatamanager.database.service.impl.BDDatabaseService"
scope="singleton" init-method="setup" destroy-method="close">
<property name="jdbcurl" value="${jdbc}" />
<property name="user" value="${user}" />
<property name="pwd" value="${pwd}" />
</bean>

<service ref="serviceDatabaseBean" interface="eu.betaas.taas.bigdatamanager.database.service.IBigDataDatabaseService" />
</blueprint>
```

It can be noticed that there is a namespace `cm` that is in fact for the configuration manager. This Karaf service is the responsible of injecting configuration into the bundle. In our case we’ve defined that the configuration file will be called `taasdb` by setting the persistent-id:

```xml
<cm:property-placeholder persistent-id="taasdb" />
```
This instruction tells to the configuration manager to look into a file called *taasdb.cfg*. Also, we defined the properties default values, so that even if the file is missing a *non-null* value will be set to the bundle parameters.

In the blueprint definition of our service we injected the properties in the following way:

```xml
<property name="jdbcurl" value="${jdbc}"/>
<property name="user" value="${user}"/>
<property name="pwd" value="${pwd}"/>
```

By using the notation of `${param}`, the configuration manager will look into the configuration file for a parameter string with the name corresponding to *param* and then it will pass it to the setter associated with the property name. For example, the property *jdbcurl* will tell to Blueprint to pass the *jdbc* parameter to the setter method called *setJdbcurl*. It is important that each param has a corresponding setter method.

### 4.2.10. Getting data from the Data Manager Service

The required dependencies for the Data Manager Service are:

```xml
<Export-Package>
   eu.betaas.taas.bigdatamanager.database.hibernate.data,
   eu.betaas.taas.bigdatamanager.database.service.impl
</Export-Package>
```

Importing the *eu.betaas.taas.bigdatamanager.database.service.impl* will make available the service interface.

By importing *eu.betaas.taas.bigdatamanager.database.hibernate.data*, the JPA\(^8\) class representing the tables will be imported in to the bundle.

The first step, in order to use the data source, is to retrieve an instance of the Data Manager service. One possibility is getting the service reference from the bundle context (provided to the method).

```java
ServiceReference reference = context.getServiceReference(IBigDataDatabaseService.class.getName());
IBigDataDatabaseService service = (IBigDataDatabaseService) context.getService(reference);
```

The second method is to use Blueprint to retrieve a reference to a registered service that implements the DataManager Service interface.

```xml
<reference id="databaseService" availability="mandatory" activation="eager" interface="eu.betaas.taas.bigdatamanager.database.service.IBigDataDatabaseService"/>
```

This reference can then be injected in another bundle so that it will be available when it will start:

```xml
<bean id="serviceTaasLoggerBean" class="eu.betaas.taas.bigdatamanager.logger.service.impl.BDLocalLoggerService" init-method="setup">
   <property name="service" ref="databaseService"/>
</bean>
```

---

In this last example, Blueprint will inject the reference `databaseService` in the variable service of the bundle `BDLocalLoggerService`. In order to work, the interface of the service should provide a setter for the variable service that will be used by Blueprint. Also, in this example the bundle defines an `init-method` that will execute the method `setup()` after setting the variable service. The type of service variable defined in `BDLocalLoggerService` must be `IBigDataDatabaseService`

Once the service is provided to the bundle when it will start, we can use the service and its method. The service itself define two ways for connecting to the database: getting a connection from the connection pool or by exchanging a Java JPA entity object that represents a fragment of data from the database.

### 4.2.10.1. Getting a connection from the pool

The connection pool allows the component to avoid uncontrolled use of the database resources and reuse already opened connections.

The method to retrieve the connection from the pool is:

```java
public Connection getConnection() throws SQLException;
```

For example, to get a connection and execute a query the following code can be used:

```java
Connection conn = service.getConnection();
PreparedStatement ps = conn.prepareStatement("SELECT COUNT(*) FROM AGREEMENT_EPR_CONTAINER");
ResultSet rs = ps.executeQuery();
while (rs.next()){
    logger.info("Database connection result for COUNT "+rs.getString(1));
}
conn.close();
```

It is always important to ensure that the connection is closed once it has been used.

### 4.2.10.2. Exchanging a Java JPA entity object

The second method to manipulate data from the database is to use a JPA entity class. Some initial classes are already created, but more can be added in the future.

The class annotated as JPA entity are placed under the package:

```java
eu.betaas.taas.bigdatamanager.database.hibernate.data
```

In order to create the corresponding SQL scripts, JPA requires that the classes are defined under the persistence.xml file:

```xml
<class>eu.betaas.taas.bigdatamanager.database.hibernate.data.AgreementEprContainer</class>
<class>eu.betaas.taas.bigdatamanager.database.hibernate.data.ApplicationRegistry</class>
<class>eu.betaas.taas.bigdatamanager.database.hibernate.data.AppService</class>
<class>eu.betaas.taas.bigdatamanager.database.hibernate.data.PersistentAgreementContainer</class>
<class>eu.betaas.taas.bigdatamanager.database.hibernate.data.TrustManagerService</class>
```

Each class allows JPA to define a table and its data. It will take care of generating the underlying queries. The class defines a field for each table field so that they can be mapped together. In order to manipulate data the service will define three possible methods:

```java
saveClassName (ClassName) // to save a new record in the table
updateClassName (ClassName) // to update an existing value in the table
```
deleteClassName (ClassName) // to delete an existing record from the table
searchClassName (ClassName) // to search a record by a filed

Where class name can be one of the following:

AgreementEprContainer
ApplicationRegistry
AppService
PersistentAgreementContainer
TrustManagerService

So if you want to save a new data inside the table AppService, just use the method saveAppService that expects as input an instance of AppService class.
5. Platform implementation

In this chapter the general specification of the BETaaS platform components implementation is provided. Each component is described through its class diagrams, public interfaces and basic code packaging. An introductive section depicting the high level architecture of the BETaaS platform is also provided for the sake of clarity (as reported with in [1], where all the related details are described).

5.1. High level architecture

As reported in [1], the BETaaS architecture has been designed in such a way that it covers all the functionalities required and it is modular so, in some cases, for certain gateways it will be possible to avoid the deployment and usage of some components.

![Class diagram of BETaaS components](image)

**Figure 22**

The figure shows the main components and the relationships. The two main components can be considered the TaaSResourcesManager and the ThingsAdaptor, since they represent the things and the way to manage them for providing the functionalities required.

5.2. InstanceManager

The InstanceManager component is in charge of managing the operations that involve the gateway entities from the instance point of view. Then the main role of IM is to communicate with other IMs to request its own GW to be accepted as a part of a BETaaS instance, to let other gateways entering its own instance and to leave the instance it belongs to. It also communicates with the other components of its own GW so that the necessary synchronization operations are performed at the time an instance is joined or disjoined.
5.2.1. Class diagrams

![Class Diagram Image]

Figure 23

Instance Manager behaves differently depending on its configured properties.

In case it is a generic IM, it uses a IM*Tracker to detect when an IM* is reachable within the instance. After a IM* is detected, the join operations are performed (involving credentials management). A successful join ends with a trigger by IM to the other local components (through their provided interfaces) in order to start the gateways synchronization. Synchronization activities are anyway partially performed by leveraging DOSGi capabilities.

5.2.2. Public interfaces

By implementing the following public interfaces, the component makes available its functionalities to the other platform components and eventually to the platform clients:

```java
boolean joinInstance(String GWID, String credentials)
```

- **Input:**
  - the ID of the new gateway that requests to join the instance
  - the credentials provided by the joining gateway

- **Output:**
  - true if the gateway request is accepted, false otherwise
Functionality:
joinInstance is used by the IMs of other gateways to request the join to the instance. It is exposed only by IM*. Once the procedure has completed, the joining GW capabilities become available to all the other gateways.

```java
boolean disjoinInstance(String GWID, String credentials)
```

- **Input:**
  - the ID of the gateway that requests to disjoin the instance
  - the credentials provided by the disjoining gateway
- **Output:**
  - true if the gateway request is accepted, false otherwise

Functionality:
disjoinInstance is used by the IMs of other gateways to request the disjoin from the instance. It is exposed only by IM*. Once the procedure has completed, the disjoining GW capabilities are no more available to the instance.

5.2.3. Code packaging

The code for the component implementation is packaged inside the following structure:

- **package eu.betaas.service.instancemanager** contains the following classes and interfaces:
  - ContextProvider
  - GWRegistry
  - GWRegistryRow
  - IMStarHandler
  - InstanceManager
  - JoinThread
  - TrackerIMCustomizer

- **Package eu.betaas.service.instancemanager.api** contains the following classes and interfaces:
  - InstanceManagerExternalIF

- **Package eu.betaas.service.instancemanager.api.impl** contains the following classes and interfaces:
  - IMAPIImpl

- **Package eu.betaas.service.instancemanager.configuration** contains the following classes and interfaces:
  - Configuration
  - Gateway
  - InstanceManager

The resources/OSGI-INF/blueprint folder contains the blueprint information to startup OSGi services and to inject properties loaded from a configuration file.

5.3. ServiceManager

This is the component which acts as the platform interface to the extern for managing BETaaS services and applications. It gathers all the needed information about the installed services and applications, so that all the requested resources can be allocated with the required security and QoS specifications. It
leverages the platform capability to abstract things by means of Thing Services and also manages aggregated and extended services.

5.3.1. Class diagrams

![Class diagram]

**Figure 24**

5.3.2. Public interfaces

By implementing the following public interfaces, the component makes available its functionalities to the other platform components and eventually to the platform clients:

```java
boolean notifyServiceInstallation(String serviceID)
```

- **Input:**
  - The identifier of the service whose installation is being notified
- **Output:** true on notification success, false otherwise
- **Functionality:**
  - `notifyServiceInstallation` is called by TaaSRM once the installation procedure is successfully completed for a required service.
boolean notifyNewMeasurement(String serviceID, String data)

- Input:
  - The service identifier to which the measurement refers
- Output: true on notification success, false otherwise
- Functionality:  
  notifyNewMeasurement is called by TaaSRM when a new measurement is available from some ThingServices corresponding to a registered service having the specified ID.

boolean installApplication(String manifestFileName)

- Input:
  - The URL from where the manifest can be loaded by SM
- Output: true on installation success, false otherwise
- Functionality:  
  installApplication is called by external applications that require to be installed on the BETaaS platform. The installation mostly consists on the preparation of thing services necessary to fulfill the requirements of the application that are specified in the manifest. Installation involves also security and quality of service negotiation.

boolean uninstallApplication(String appID, String manifestFileName)

- Input:
  - The ID of the application that is requesting to be uninstalled
  - The URL from which the application manifest may be loaded by SM
- Output: true on uninstall success, false otherwise
- Functionality:  
  uninstallApplication is called by the application that requests the platform to deallocate all the resources previously requested during the installation.

String getThingServiceData(String appID, String serviceID)

- Input:
  - The ID of the application requesting the data
  - The ID of the service for which the data is requested
- Output: the data provided by the specified service
- Functionality:  
  getThingServiceData is used by external applications to pull data from a previously installed service. The kind of data and the time it refers to depend on the requirements specified in the manifest that was provided during the installation.

boolean register(String appID, String serviceID)

- Input:
  - ID of the application requesting the register procedure
  - ID of the service to which the application request to register
- Output: true on success, false otherwise
Functionality:
*register* is used by applications that want to register once to a previously installed service in order to automatically receive the corresponding data updates from BETaaS in push mode.

```java
boolean unregister(String appId, String serviceID)
```

**Input:**
- ID of the application requesting the unregister procedure
- ID of the service to which the application request to unregister

**Output:** true on success, false otherwise

Functionality:
*unregister* is called by external applications that previously registered to a service. After *unregister* is called data updates are no more sent to the application for the specified service.

```java
String getTaskData(String taskId)
```

**Input:**
- ID of BDM task to be executed

**Output:** the data provided by the BDM task

Functionality:
*getTaskData* is used by applications to request a data analysis to the platform. The specified ID is selected among a registry of available tasks provided by the platform to the applications.

### 5.3.3. Code packaging

The code for the component implementation is packaged inside the following structure:

- **package eu.betaas.service.servicemanager** contains the following classes and interfaces:
  - ServiceManager
  - ServiceManagerActivator

- **package eu.betaas.service.servicemanager.api** contains the following classes and interfaces:
  - ServiceManagerExternalIF
  - ServiceManagerInternalIF

- **package eu.betaas.service.servicemanager.api.impl** contains the following classes and interfaces:
  - ExternalAPIImpl
  - InternalAPIImpl

- **package eu.betaas.service.servicemanager.apps** contains the following classes and interfaces:
  - ApplicationRegistry
  - AppManifest
  - AppRegistryRow
  - AppService
  - ServiceRequirement

- **package eu.betaas.service.servicemanager.apps.messages** contains the following classes and interfaces:
5.4. TaaS Resource Manager

The TaaS Resource Manager is a component which works at the TaaS level and it can be considered the core of the BETaaS instance operation. It is in charge of managing the resources present in the instance, by mapping and allocating them properly, depending on external applications needs.

This component is also responsible of redirecting the invocations to Thing Services to the adaptation layer, retrieving data which will be provided to the applications. This intermediate step provides an abstraction layer to the applications, since the TaaS Resource Manager may change the real Thing Service to be invoked depending on the current context.

In the current implementation, the TaaS Resource Manager can perform basic management of resources and applications, using the Context Manager for obtaining candidate Thing Services and taking into consideration the inputs given by the Trust Manager and the QoS Manager. At invocation time, the re-allocation of resources is limited and there is no active monitoring and optimization of all the resources allocated in the instance. Although it is possible to create VMs in external clouds, still it is not used for providing VMs to applications and to the Big Data Manager.

5.4.1. Class diagrams

The following diagram shows the main classes involved in the TaaS Resources Manager of BETaaS, according to its OSGi compliant implementation.

The TaaSResourcesManager is the public API exposed as a service in the OSGi instance by means of Blueprint. All the methods provided are implemented by the TaaSResourcesManagerImpl class.

![Class diagram](image)

**Figure 25**

In this first diagram, we show the activator class required by OSGi, which will initialize all the catalogues and the clients (although not shown in the diagram for simplicity, all the clients are generated when the bundle is activated).
The diagram presents both the public API of the component (the interface TaasResourcesManager) and its implementation (TaasResourcesManagerImpl). They act as a façade pattern and provide all the methods required by other components located at the same or different layers. Also, it contains methods which can be used to synchronize several Taas Resources Managers running in the same instance but different gateways. As a way to facilitate the provision of requirements, we have created the class Feature, which will provide information about those features that applications require to the BETaaS instance.

While the ResourcesCatalog will store information about resources available in the instance (both things and virtual resources such as VMs), the ApplicationCatalog will contain information about those external applications which requested resources, the features requested and the mapping between those features and the Thing Services which will implement them. Both catalogues are implemented applying the singleton pattern, since only one catalogue of each kind should be active, avoiding potential synchronization and race issues.

When allocating resources, the ResourcesAllocator will be in charge of carrying out all the steps defined in the architecture to determine if it is possible to fulfill the external application requirements and, if possible, which Thing Services will be used for doing so. It will require using the catalogues and it will update the status of the resources which were allocated.

The EndPointsManager will be the class taking care of the invocation of Thing Services, by using those mappings contained in the catalogues.

Figure 26
If we take a closer look to the catalogues, we realize that their structure is not so simple. On one hand, we need to keep information about each ‘virtual’ resource and its physical implementation (by means of the Resource class and the ResourcesCatalog). On the other hand, we need to know some information about the external applications and their status (represented by the Application class and listed in the ApplicationsCatalog) as well as certain details about the features requested and how they can be mapped to the existing resources in the BETaaS instance, information which is contained in the FeatureService class.

In order to complete information about the Thing Services and other resources present in the instance, the component requires interacting with the VM Manager and the Big Data Manager in the TaaS layer. The interaction with the VM Manager will be performed through the TaaSVMClient class (in order to get some information about available VMs), while the TaaSBDMClient will be the one in charge of the interaction with the Big Data Manager, in order to retrieve some information about the external applications and the things registered in the instance. In all cases, the clients are implemented with the singleton pattern, so only one client will be active for the bundle.

The allocation of resources is not a simple task either, and it requires performing some steps in which the interaction with other components is mandatory.

Apart from the interactions already mentioned, the ResourcesAllocator will require to contact with the VM Manager, the Context Manager the Security Manager (here, represented by the Trust Manager directly) and the Service Manager. The TaaSVMClient will be in charge of the interaction with the VM Manager, the TaaSContextClient will interact with the Trust Manager, the TaaSServiceClient will interact with the Service Manager.

In the case of the invocation process, this is done by the EndPointsManager, which requires interacting with the catalogues and with several external components before performing the invocation (in case some of the Thing Services in the invocation list must be changed). The AdaptTAClient will interact with the Things Adaptor in order to check that a Thing Service is active and to retrieve data. The TaaSContextClient will be used to check some ‘soft security’ parameters (trust) before the invocation and the TaaSQoSClient will interact with the QoS Manager in order to confirm no SLA will be violated before the invocation. All these classes will enable the possibility to invoke Thing Services in pull mode each time an application request a feature.

### 5.4.2. Public interfaces

The public API of the component is defined in the TaaSResourcesManager java interface. By implementing the following public interfaces, the component makes available its functionalities to the other platform components and eventually to the platform clients:
< allocateResources>

- Input: a Feature object containing information about the feature required by the application (feature, location, mode, non-functional information...)
- Output: a String representing the identifier of the service which will implement the required feature
- Functionality: The component will carry out all the process of allocation, retrieving the list of candidates from the Context Manager and filtering the list according to security and quality parameters.

< freeLocalResources>

- Input: String with the identifier of an application
- Output: there is no output
- Functionality: The component will remove the application from the catalogue and all the resources allocated for the application will be released, changing their status.

< synchronizeThingServices >

- Input: String containing the identifier of gateway
- Output: a boolean indicating if the operation finished correctly
- Functionality: The component will communicate with other TaaS Resource Manager located in another gateway belonging to the same instance, in order to synchronize the catalogues.

< synchronizeThingServices >

- Input: ArrayList of Strings containing the identifiers of Thing Services already known
- Output: an ArrayList of Strings containing the identifiers of those Thing Services known by the target gateway
- Functionality: This method is for sending to other gateway the information about known Thing Services while, at the same time, we receive information about those Thing Services known by the remote gateway.

< removeThingServices >

- Input: String containing the identifier of a remote gateway
- Output: a boolean indicating if the operation finished correctly
- Functionality: The component will remove those Thing Services provided by the remote gateway, since they are expected to be unavailable.

< removeThingServices >

- Input: There is no input
- Output: a boolean indicating if the operation finished correctly
- Functionality: This method will clean the catalogues completely.
< registerThingsService >
• Input: String indicating the physical thing identifier and String indicating the Thing Service identifier, as its 'virtual' representation
• Output: a boolean indicating if the operation finished correctly
• Functionality: This method will add the provided Thing Service to the resources catalogue.

< removeThingsService >
• Input: String indicating the Thing Service identifier to be removed
• Output: a boolean indicating if the operation finished correctly
• Functionality: This method will remove the provided Thing Service from the resources catalogue.

< registerApplication >
• Input: String indicating the identifier of an external application to be registered
• Output: a boolean indicating if the operation finished correctly
• Functionality: The component will create an entry in the Applications catalogue for the provided application identifier.

< unRegisterApplication >
• Input: String indicating the identifier of an external application to be removed
• Output: a boolean indicating if the operation finished correctly
• Functionality: The component will remove the entry in the Applications catalogue for the provided application identifier and it will release any resources allocated.

< getData >
• Input: String indicating the identifier of the service which implements certain feature required by certain external application
• Output: an ArrayList of Strings containing the results of all the Thing Services which had to be invoked plus the operation to be applied to those results
• Functionality: The component will check which Thing Services have to be invoked, it will invoke them one by one (taking into account the re-evaluation of certain security and quality aspects) and it will return the results with the application that, according to the Context Manager, has to be applied to those results (i.e. logical OR operation or an average).

< getSecurityRank >
• Input: String indicating the identifier of the service which implements certain feature for a certain external application
• Output: an ArrayList of ArrayList of Strings (a matrix, in the end) containing the groups of equivalent Thing Services which can implement the feature required
• Functionality: After performing a filtering of Thing Services candidates for implementing a feature and the equivalent Thing Services (mainly because of a security evaluation), the component can provide this list to external components which may need it.
**5.4.3. Code packaging**

The code for the component implementation is packaged inside the following structure:

- package `eu.betaas.taas.taasresourcesmanager` contains the following classes and interfaces:
  - TaaSResourcesManagerActivator.java
- package `eu.betaas.taas.taasresourcesmanager.api` contains the following classes and interfaces:
  - TaaSResourceManager.java
- package `eu.betaas.taas.taasresourcesmanager.api.impl` contains the following classes and interfaces:
  - TaaSResourceManagerImpl.java, Feature.java
- package `eu.betaas.taas.taasresourcesmanager.catalogs` contains the following classes and interfaces:
  - Application.java, ApplicationsCatalog.java, FeatureService.java, Resource.java, ResourcesCatalog.java
- package `eu.betaas.taas.taasresourcesmanager.endpointsmanager` contains the following classes and interfaces:
  - EndpointsManager.java
- package `eu.betaas.taas.taasresourcesmanager.resourcesoptimizer` contains the following classes and interfaces:
  - ResourcesAllocator.java
- package `eu.betaas.taas.taasresourcesmanager.taasrmclient` contains the following classes and interfaces:
  - ServiceSMClient.java, TaaSBDMClient.java, TaaSCLMClient.java, TaaSQoSMClient.java, TaaSRMClient.java, TaaSTMClient.java, TaaSVMClient.java

**5.5. ThingsAdaptor**

This component represents the bridge between the actual Things (physical devices) and the BETaaS capabilities related to the things. It is able to discover things that connect (or have been connected) to the BETaaS instance, maintain data about them, monitor them, access them and provide those adaptation interfaces so other components can provide richer context capabilities (QoS, resources management, trust, context management).

It will eventually provide not only the network protocols, but also other basic needs depending on the driver used for communicating with the device (i.e. identification management). It unifies the information
received from heterogeneous Things (different manufacturers, different standards, etc.) connected to the BETaaS instance. The component transforms the raw data emitted by the things into a serializable object that provides enriched context to the ContextManager in order to parse this predefined object into a semantic format that is mapped into the BETaaS ontology.

### 5.5.1. Class diagrams

Briefly described in the diagram below is the Class diagram for the ThingsAdaptor. An implementation of the ThingsAdaptor class, shown here as ThingsAdaptorImpl, uses privately the ThingsDiscovery class that runs a continuous thread of the PortReader Runnable. The latter is essentially a job that runs for as long as the BETaaS instance is alive, checking for the Things data and publishing new Things if found.

![Class Diagram](image)

**Figure 28**

### 5.5.2. Public interfaces

By implementing the following public interfaces, the component makes available its functionalities to the other platform components and eventually to the platform clients:

- `<getThings>`
  - Input: none
  - Output: A list fo ThingsObject as List<ThingsObject>.
  - Functionality: Returning a list of the available Things to any requesting module

### 5.5.3. Code packaging

The code for the component implementation is packaged inside the following structure:

- package `eu.betaas.adaptation.thingsadaptor.api` contains the following Interface:
  - ThingsAdaptor
- package `eu.betaas.adaptation.thingsadaptor.api.impl` contains the following class:
  - ThingsAdaptorImpl
- package `eu.betaas.adaptation.thingsadaptor.service` contains the following class:
  - ThingsDiscovery
5.6. BigDataManager

As reported in [1], this component provides advanced storage capabilities for large amounts of data coming from things and their monitoring. It also performs data analysis in order to find more information about the data usage and other data processing which can be done. It can take advantage of the VMManager in case it needs to use VMs for especial storage purposes, when the capability is available.

5.6.1. Class diagrams

The following diagram describes the internal structure of the BigData Manager at TaaS level. It consists in few classes that provide services to store inside the database things data and then retrieve them.

![Class Diagram](image)

*Figure 29*
At service level the interface of the TaaS level is used to get the data and store it in a centralized database. The layer also offers service of storage on distributed file system and component to manage query for real time information retrieval and scheduler manager to execute complex procedures that requires time to generate output.

![Diagram](image)

**Figure 30**

### 5.6.2. Public interfaces

By implementing the following public interfaces, the component makes available its functionalities to the other platform components:

#### 5.6.3. BETaaS BDM Service Core

**public** `execTask`

- Input: `JsonObject`
- Output: `JsonObject`

Functionality: this function gets as input a Json object representing the parameters of the task. In this case the data returned is empty if the task, which is not real time, is not already available. Typically this function is used to retrieve data from scheduled processing happening at given times provided by the scheduler.

**public** `getTaskData`

- Input: `JsonObject`
- Output: `JsonObject`

Functionality: this function gets as input a Json object representing the parameters of the task. In this case the data returned is taken in real time from the database. This function is used by application that wants to get information recently collected by one or more gateways.
**public** getTaskAvailable

- **Input:** none
- **Output:** JsonObject

Functionality: this function returns the list of available tasks currently available to the system.

### 5.6.4. BETaaS BDM TaaS Core

**public** getThingsData

- **Input:** Timestamp
- **Output:** JsonObject

Functionality: this function gets as input a timestamp, telling to the TaaS BDM core the data age that need to be transferred from TaaS to Service layer. In this way the function collects data from the oldest to the time specified by the parameter and wraps it in a JsonObject that is sent to Service BDM Core. The timestamp limits the amount of data transferred and can be tuned to avoid excessive transfer of data.

**public** setThingBDM

- **Input:** JsonObject
- **Output:** JsonObject

Functionality: this function gets as input an object containing a data returned by a thing. Together with the data also the general thing information, such as type and location, are returned. The function store the thing information only the first time and every new data reported is saved in to the TaaS database.

### 5.6.5. Code packaging

The code for the component implementation is packaged inside the following structure:

- **package** eu.betaas.bigdatamanager.dataservice.database.services contains the following classes and interfaces:
  - `IDatabaseBDMService`, `DatabaseBDMService`

- **package** eu.betaas.service.bigdatamanager.core.services contains the following classes and interfaces:
  - `IBigDataDistributedCoreService`, `BDDistributedCoreServiceImpl`

- **package** eu.betaas.taas.bigdatamanager.core.services contains the following classes and interfaces:
  - `ITaasBigDataManager`, `TaasBigDataManagerService`

- **package** eu.betaas.taas.bigdatamanager.database.service contains the following classes and interfaces:
  - `IBigDataDatabaseService`, `BDDatabaseService`

### 5.7. VMManager

This component is in charge of managing Virtual Machines that are seen as resources in a BETaaS instance and can be used for different purposes. These VMs will be created in those gateways with enough capabilities for doing so but there is also the possibility to use external resources by means of standardized interaction protocols with external Clouds (such as OCCI). This component works at the BETaaS TaaS layer, since VMs are managed by the TaaS Resources Manager.
At this moment, the component implements the OCCI interaction with Open Nebula clouds, through REST messages. It has also integrated a specific client for OCCI with JSON in order to interact with one of the platforms which were exposed in the OPTIMIS project.

5.7.1. Class diagrams

The following diagram shows the main classes involved in the VMManager of BETaaS, according to its OSGi compliant implementation.

The TaaSVMManager is the public API exposed as a service in the OSGi instance by means of Blueprint. All the methods provided are implemented by the TaaSVMManagerImpl class.

Figure 31

The VMAllocatorManager is in charge of the resources allocation when a new VM is requested (taking into account available local hardware resources and external ones), while the TaaSVMManagerActivator is the class that OSGi uses for activating the bundle.

Finally, the OpenNebulaClient interface exposes the API to be used for managing resources in external clouds by using OCCI. This interface is implemented by the OpenNebulaClientImpl, which takes care of the operations to be performed.

The VMManager also has implementations of the data model which supports the OCCI interaction (such as those data types used: Disk, Nic, Compute, etc...) as well as the client for OCCI with JSON and other supporting classes for requests and VMs information (VMRequest, VMProperties and Service).

5.7.2. Public interfaces

The public API of the component is defined in the TaaSVMManager java interface. By implementing the following public interfaces, the component makes available its functionalities to the other platform components and eventually to the platform clients:

< createVM>
- Input: VMRequest containing the resources required for the VM (memory, CPU, storage, etc...)
- Output: a String identifying the VM created
- Functionality: The component will determine if it is possible to create the requested VM, allocating the appropriate resources and generating a new identifier.

< startVM>
- Input: String with the VM identifier
- Output: a Boolean indicating if the VM was started
- Functionality: The component will start the VM requested.

< stopVM>
- Input: String with the VM identifier
- Output: a Boolean indicating if the VM was stopped
• Functionality: The component will stop the VM requested.

<removeVM>
• Input: String with the VM identifier
• Output: a Boolean indicating if the VM was removed
• Functionality: The component will remove the VM requested.

<getVMInfo>
• Input: String with the VM identifier
• Output: a VMProperties object containing important information about the VM
• Functionality: The component will retrieve information about the VM requested.

<getVMs>
• Input: String with the identifier of an application
• Output: a list of Strings containing identifiers of VMs
• Functionality: The component will retrieve the list of all the VMs associated to a concrete application.

5.7.3. Code packaging
The code for the component implementation is packaged inside the following structure:
• package eu.betaas.taas.taasvmmanager contains the following classes and interfaces:
  o TaaSVMManagerActivator.java
• package eu.betaas.taas.taasvmmanager.api contains the following classes and interfaces:
  o TaaSVMManager.java
• package eu.betaas.taas.taasvmmanager.api.impl contains the following classes and interfaces:
  o TaaSVMManagerImpl.java
• package eu.betaas.taas.taasvmmanager.vmsallocator contains the following classes and interfaces:
  o VMsAllocatorManager.java
• package eu.betaas.taas.taasvmmanager.cloudsclients contains the following classes and interfaces:
  o OCCI_RestClient.java, OCCIClient.java, OCCIClientException.java, Service.java, VMProperties.java, VMRequest.java
• package eu.betaas.taas.taasvmmanager.opennebula.client contains the following classes and interfaces:
  o OpenNebulaClient.java, OpenNebulaClientImpl.java, OpenNebulaException.java
• package eu.betaas.taas.taasvmmanager.opennebula.datamodel contains the following classes and interfaces:
  o Compute.java, ComputeCollection.java, ComputeState.java, InstanceType.java, InstanceTypeCollection.java, Link.java, Network.java, NetworkCollection.java, ObjectFactory.java, Resource.java, Storage.java, StorageCollection.java, StorageType.java, User.java, UserCollection.java
• package eu.betaas.taas.taasvmmanager.opennebula.util contains the following classes and interfaces:
  o OCCIProcesser.java

5.8. QoSManager

QoSManager is in charge of managing all the functionalities of the platform related with Quality of Service. In particular the module is responsible for implementing the following functionalities:

• QoS negotiation: In order to fulfill the application needs, the QoS of the Thing Services is negotiated to stipulate a SLA. For this reason, a negotiation protocol is implemented by this functionality.

• QoS management: QoSManager is responsible for managing the resources, efficient management of things will be performed exploiting Equivalent Thing Services by means of a number of parameters, e.g. in terms of energy efficiency. In order to handle and exploit the dynamicity of thing status and their varying availability, QoS implements the management as a two-steps procedure which entails resource reservation at the time of the service negotiation and resource allocation at the time of the Thing Service invocation.

• QoS monitoring: The module will implement monitoring functionalities to monitor the status of the resources and the quality of the exposed Thing Services. Resource monitoring is necessary to detect malfunctions/failures and to collect information regarding the status of the Things. SLA monitoring, instead, monitors the QoS level obtained by each Thing Service to check that the negotiated SLAs are fulfilled.

The first release of the platform will implement the QoS negotiation functionality while QoS management and QoS monitoring functionalities are planned to be implemented in the next releases.

The QoS negotiation functionalities are provided through the implementation of the protocol WS-Agreement Negotiation. At this aim the framework WSAG4J\(^9\) will be integrated into the QoSManager in order to exploit its WS-Agreement Negotiation implementation.

The first implementation of the QoSManager results in the following OSGi bundles:

• WSAG4J negotiation bundle, it is the porting of the WSAG4J core as OSGi bundle into the BETaaS platform. This bundle implements the WS Agreement Negotiation protocol and performs basic operations needed for SLA management. The original WSAG4J code has been stripped and modified in order to integrate into the platform.

• QoS Manager core, it implements the interfaces of the QoSManager. Regarding the negotiation process it is responsible to expose the public interfaces for QoS negotiation which are implemented interacting with the WSAG4J negotiation bundle accordingly.

5.8.1. Class diagrams

The image below shows the class diagram of the QoSManager core for the first implementation of the platform. The main class is the QoSManager which is responsible for implementing the logic of the module. In order to expose services to other modules two interfaces, QoSManagerInternalIF and QoSManagerExternalIF, are defined to expose the services of the QoSManager to other bundles running on the same gateway and to external bundles running of other gateways, respectively. At the moment, the following implementations of these two interfaces are provided: InternalAPImpl and ExternalAPImpl.

In order to initiate the bundle, allocate data and initialize structures, an activator class, the QoSManagerActivator, is also implemented.

\(^9\)http://wsag4j.sourceforge.net
5.8.2. Public interfaces

The QoSManager implements only internal public interfaces, i.e. interfaces that provide services to other modules running on the same gateway. Even though some external interfaces are defined in QoSManagerInternalIF, they are not public interfaces, i.e. they are not exposed to be used by other bundles but only by different instances of QoSManager. In particular they are exploited by all the QoSManager instances to communicate with the one running on the GW*.

The following public interfaces are implemented by the QoSManager:

<getTemplate>
- Input: none
- Output: String containing the template to be used for negotiation in xml format
- Functionality: This function is called to fetch the template before starting the WS-Agreement Negotiation

<createAgreement>
- Input: String offer, it contains the offer in xml format for the WS-Agreement negotiation
- Output: String containing the Agreement EPR resulting from the negotiation. The negotiation implicitly succeeded if a valid EPR is returned, otherwise an invalid EPR is returned
- Functionality: This function is called to perform the WS-Agreement Negotiation

<registerServiceQoS>
- Input: String serviceId, the service ID, ArrayList<String> thingServicesList the list of the required Thing Services, ArrayList<ArrayList<String>> the list of equivalent Thing Services associated with each Thing Service
Output: `ArrayList<String>` containing the list of Thing Services selected among the list of equivalent Thing Services to provide the required thing services

Functionality: This function is called by the TaaSRM at the time of service invocation for selecting the Thing Services to be invoked

### 5.8.3. Code packaging

The code for the component implementation is packaged inside the following structure:

- **package `eu.betaas.service.qosmager.core`** contains the following classes and interfaces:
  - QoSManager, QoSManagerActivator

- **package `eu.betaas.service.qosmager.api`** contains the following classes and interfaces:
  - QoSManagerExternalIF, QoSManagerInternalIF

- **package `eu.betaas.service.qosmager.api.impl`** contains the following classes and interfaces:
  - ExternalAPIImpl, InternalAPIImpl

### 5.9. ContextManager

As reported in [1], this component manages all the semantic processes that take part in BETaaS. In particular, the module is responsible for implementing the following functionalities:

- Translation of the information retrieved/sent from/to the things into a semantic format that is mapped into the BETaaS ontologies. **The first release of the platform only processes semantically the data retrieved from the things while the semantic processing of the data sent to things will be implemented in the next releases.**

- Managing the context or circumstances in which things that are connected to BETaaS are embedded. The context information is mapped into the BETaaS Context Ontology.

- Managing of the subscriptions/unsubscriptions made by the applications that use BETaaS to the elements in the ontology. **The first release of the platform retrieves information in a Real Time Pull mode only. The subscription/unsubscription will be implemented in the next releases.**

- Generation of the Thing Services used to expose the things as services.

- Sending the BigDataManager all the information about the things registered in BETaaS.

### 5.9.1. Class diagrams

The ContextManager implements classes in two layers:

- The TaaS_ContextManager implements an interface with the TaaS Layer.
The Adaptation_ContextManager implements an interface with the Adaptation Layer.

The SemanticParserAdaptator in the Adaptation Layer works in a similar way as the ThingServiceManager in the TaaS Layer. This means that the SemanticParserAdaptator imports `eu.betaas.taas.contextmanager` in order to have access to the ontologies and avoid the replication of the class structure.

5.9.2. Public interfaces

By implementing the following public interfaces, the component makes available its functionalities to the other platform components and eventually to the platform clients:

```java
< boolean publishThing(ArrayList<ThingsData> oThingsDataList)>
```

- Input:
**ArrayList<ThingsData> oThingsDataList**, where oThingsDataList is a list of arrays of the serializable object, ThingsData, with the following parameters:

- private boolean bInput: true when the thing is an actuator;
- private boolean bOutput: true when the thing is a sensor;
- private boolean bDigital: true when the thing is a digital device;
- private boolean bAnalogue: true when the thing is an analog device;
- private String sMaker: name of the maker of the thing;
- private String sID: thing ID within a gateway;
- private String sType: type of the thing;
- private String sMeasurement: measurement taken by the thing;
- private String sUnit: units of the measurement taken;
- private String sEnvironment: it defines the type of scenario where the thing is used: home/city;
- private String sRoom: it defines the type of scenario where the thing is used (only relevant when environment=home);
- private String sFloor: it defines the location of the thing (only relevant when environment=home);
- private String sCityName: it defines the location of the thing (only relevant when environment=city);
- private String sLatitude: it defines the location of the thing (only relevant when environment=city);
- private String sLongitude: it defines the location of the thing (only relevant when environment=city);

**Output:**

- boolean bCorrect, returns true if everything is OK, and false in case of failure.

**Functionality:** it is used by the ThingsAdaptor to inform the ContextManager that a new thing has been connected to the GW. The information related to the thing is sent to the ContextManager (it has been previously pre-processed by the ThingsAdaptor) to translate these data into a semantic format that is mapped into the BETaaS ontologies.

**< ThingServiceList get ContextThingServices(String sParameter, String sLocation, String sMode)>**

**Input:**

- String sParameter, parameter describing the type of thing that it is going to be used, i.e, "presence".
- String sLocation, parameter describing the location where the thing is, i.e. "home".
- String sMode, parameter describing how the data is retrieved from things, i.e. "pull".

**Output:**

- ThingServiceList, object containing the list of matching Thing Services, the lists of equivalent Thing Services, the logical operator to be applied to combine this final list:
  - ArrayList<String> CMThingServicesList;
  - ArrayList<Object> CMThingServicesListEq;
  - String sOperator;
• Functionality: it is used by the TaaSResourceManager to ask the ContextManager to find the Thing Services that suit the context needed for the resources of an application.

### 5.9.3. Code packaging

The code for the component implementation is packaged inside the following structure:

- package eu.betaaS.adaptation.contextmanager contains the following classes and interfaces:
  - `eu.betaaS.adaptation.contextmanager.api`:
    - SemanticParserAdaptator, SemanticParserAdaptatorImpl
- package eu.betaaS.taas.contextmanager contains the following classes and interfaces:
  - `eu.betaaS.taas.contextmanager.api`:
    - ThingsServiceManager, ThingsServiceManagerImpl
  - `eu.betaaS.taas.contextmanager.onto.classesExt.semantic.jenaSdb`:
    - JenaSdbEx
  - `eu.betaaS.taas.contextmanager.onto.classesExt.semantic.data.xml.sparqlResultSet`:
    - SparqlResult, SparqlResultSet, SparqlVariable
  - `eu.betaaS.taas.contextmanager.onto.classesExt.commonUtils`:
    - ConfigBundleOSGi, ConfigBundleOSGiImpl
  - `eu.betaaS.taas.contextmanager.onto.core`:
    - OntoBetaas
  - `eu.betaaS.taas.contextmanager.onto.classesExt.db.h2`:
    - H2Db

### 5.10. SecurityManager

This component is responsible in providing security and trust in the BETaaS platform. The functionality includes trust monitoring and calculation of the thing services, key or certificate management as well as authentication and authorization for BETaaS gateways and applications. This component mainly interacts with the TaaSResourceManager in all of its functionalities. It also interacts with the ThingsAdaptor in order to monitor and calculate the trust of a thing service.

#### 5.10.1. Class diagrams

The following picture briefly describes the class diagram of the current version of the SecurityManager. At the moment, the SecurityManager consists of 6 main bundles, namely core, common, certificate, authentication, requirements, and TaaSTrustManager bundles. The main bundle that provides the main SecurityManager interfaces to the other components is the core bundle that currently is connected with the common, certificate, and authentication bundles. But for the demo purpose, only the interfaces in the requirements and TaaSTrustManager bundles will be provided to the other components, i.e. TaaSResourceManager. Later on, the requirements bundle will be merged into the core bundle, and the interfaces in the TaaSTrustManager bundle will be accessed through the core bundle.
5.10.2. Public interfaces

This sub-section only gives explanation of the public interface that will be used in the demo. By implementing the following public interfaces, the component makes available its functionalities to the other platform components and eventually to the platform clients:

**<getSecurityRank>**
- Input: A list of things services groups (a group similar thing service in different rooms) as \( \text{List<\text{List<String>}} \), and a set of security specs as \( \text{Map<String, String>} \)
- Output: A list of things services groups after the security filtering and ranking as \( \text{List<\text{List<String>}} \)
- Functionality: Returning a list of things services groups after the security filtering and ranking based on the provided security specs.

**<getTrust>**
- Input: A thing service ID as \( \text{String} \)
- Output: The requested thing service ID’s trust score as \( \text{double} \)
- Functionality: Returning trust score of a particular thing service ID.

**<getTrust>**
- Input: A list thing service IDs as \( \text{ArrayList<String>} \)
• Output: A list of the requested thing service ID’s trust scores as ArrayList<double>
• Functionality: returning trust scores of a list of thing service IDs.

5.10.3. Code packaging
The code for the component implementation is packaged inside the following structure:

- package eu.betaas.taas.securitymanager.requirements.service contains the following interfaces and classes:
  - ITingsRequirementsService, ThingsRequirementsService
- package eu.betaas.taas.trustmanager.api contains the following interfaces and classes:
  - TaaSTrustManager, TaaSTrustManagerImpl
- package eu.betaas.taas.securitymanager.core.service contains the following interfaces and classes:
  - IInitGWStarService, InitGWStarService
  - IJoinInstanceService, JoinInstanceService
  - ISecGWCommService, SecGWCommService
- package eu.betaas.taas.securitymanager.certificate.service contains the following interfaces and classes:
  - IGatewayCertificateService, GWCertificateService
  - IGatewayStarCertificateExtService, GWStarCertificateExtService
  - IGatewayStarCertificateIntService, GWStarCertificateIntService
- package eu.betaas.taas.securitymanager.authentication.service contains the following interfaces and classes:
  - IAppsAuthenticationService, AppsAuthenticationService
  - IGatewayEcmqvExtService, GWEcmqvExtService
  - IGatewayEcmqvIntService, GWEcmqvIntService

5.10.4. Additional notes on bundles activation steps:
For the demo:
• TrustManager bundle (and ThingsAdaptor) is required for the requirements bundle to execute the getSecurityRank method properly. But the requirements bundle can still run (be active) properly without those two bundles.

The other bundles (not for demo at the moment):
• All securitymanager bundles require the common bundle to be resolved. The common bundle should be installed with wrap: option.
• The dependencies among the bundles:
  - Core bundle depends on certificate, authentication, and common bundles.
  - Authentication bundle depends on certificate and common bundles.
  - Certificate bundle depends on common bundle.
• All bundles, except core, require the following dependencies to be compiled and run properly:
  - org.bouncycastle/bcprov-jdk15on/1.49
  - org.bouncycastle/bcprov-ext-jdk15on/1.49
  - org.bouncycastle/bcpkix-jdk15on/1.49
5.11. TaaS Trust Manager

This subcomponent is part of the Security Manager, since it provides one of the main extended functionalities related to ‘soft security’ mechanisms. It is in charge of evaluating the behavior of things services in order to determine whether they are trustworthy or not. This evaluation depends on the trust parameters already defined in the extended capabilities deliverable.

In the current implementation, the Trust Manager is able to calculate QoS fulfillment, battery load and stability in provided data. These calculations are supported by time series forecasting operations (calculated with simple and double exponential smoothing) and hypothesis tests (variance tests, runs tests, etc…), which are implemented by the corresponding supporting classes. The trust aggregation based on a fuzzy model is also implemented, so a final trust value can be obtained.

5.11.1. Class diagrams

The following diagram shows the main classes involved in the TaaS Trust Manager of BETaaS, according to its OSGi compliant implementation.

The TaaSTrustManager is the public API exposed as a service in the OSGi instance by means of Blueprint. All the methods provided are implemented by the TaaSTrustManagerImpl class.

In this first diagram, we show the activator class required by OSGi, which starts the TrustTaaSThread, the object which will represent the thread for calculating trust periodically. This class is implemented as a singleton pattern, so there will be only one thread calculating trust for the whole instance. The ‘real’ implementation of the thread is represented by the TaaSMonitoringThread, which represents the task scheduled according to the time interval defined. The scheduler at TrustTaaSThread will activate TaaSMonitoringThread from time to time, which will re-calculate trust by using the ThingServiceTrustCalculator class. The thread will also check if some trust threshold has been violated, in order to raise an alert (notifications at this point are not implemented, but are expected to be used with the TaaS Resources Manager).

In the diagram we also represent the public API exposed by the subcomponent (TaaSTrustManager), as well as the class implementing the interface: TaaSTrustManagerImpl. This class will use the TrustTaaSThread for subscribing alerts and registering thing services (this operation adds priority to their trust evaluation, increasing the number of evaluations for that Thing Service). It will also use the TaaSBDMClient, which is able to retrieve and store information about trust in the Big Data Manager.
component. This client (as any other in the component) is implemented with a singleton pattern as well, in order to optimize resources and make easier its creation (it requires some information provided by the activator class).

As the subcomponent complexity is high, we have generated a different class diagram more focused on those classes involved in the trust calculation, shown in the next figure.

![Class Diagram]

Figure 37

The ThingServiceTrustCalculator is in charge of orchestrating the trust calculation by invoking the different classes which calculate individual parameters and, later on, it will use the FuzzyAggregator for aggregating all the values and obtaining a global trust value. In this implementation, due to simplicity, it also implements the battery load parameter, but this functionality will be contained in a separate class as its complexity increases (in the next version).

The QoSFullFulfillmentCalculator will retrieve information about a Thing Service from the QoS Manager (through the TaaSQoSClient class), so it will be possible to know in how many cases the Thing Service fulfills its agreements. It will aggregate the information with an opinion model and it will provide a trust evaluation.

In order to determine stability in the data provided, the DataStabilityCalculator will retrieve the data already generated by a Thing Service from the TaaSBDMClient (as it will store some historical data) and, after executing some hypothesis tests, it will provide a trust evaluation.

In the case of the ExponentialSmoothingCalculator, it is able to retrieve some historical information (i.e. about previous battery load or time to invoking Thing Services) and to provide some forecast with several methods. In this case, the most appropriate ones are simple and double exponential smoothing (since for the triple one it is mandatory to have seasonality in the data and we still do not calculate execution time). The forecast value can be used to determine if that value is adequate or not.
Finally, the FuzzyAggregator class is in charge of loading fuzzy models and evaluate them by receiving a map of parameters – values. It is implemented in such a way it could use different fuzzy models, but at this moment we will only use one for the global trust value, although we expect to use more with other individual parameters in the future.

5.11.2. Public interfaces

The public API of the component is defined in the TaasTrustManager java interface. By implementing the following public interfaces, the component makes available its functionalities to the other platform components and eventually to the platform clients:

< getTrust>
- Input: String containing the identifier of a Thing Service
- Output: a double value representing the global trust value
- Functionality: The component will retrieve the trust value calculated in the last assessment of the given Thing Service.

< getTrust>
- Input: ArrayList with the identifiers of all the Thing Services to be evaluated
- Output: an ArrayList of double values containing the trust value for each Thing Service in the same order as the input list
- Functionality: The component will retrieve the trust values calculated in the last assessment of all the given Thing Services.

< registerThingsService >
- Input: String containing the identifier of a Thing Service
- Output: a Boolean indicating if the Thing Service was registered
- Functionality: The component changes the Thing Service to a priority list so its trust will be reevaluated more often, as trust is supposed to be more up to date than for those Thing Services which are deployed but not in use.

< removeThingsService >
- Input: String containing the identifier of a Thing Service
- Output: a Boolean indicating if the Thing Service was removed
- Functionality: The component will remove the Thing Service requested from the priority list, so its trust will be reevaluated less times, as it is not in use and we can save resources.

< subscribeThreshold >
- Input: String containing the identifier of a Thing Service and a double value representing the threshold
- Output: a Boolean indicating if the Thing Service and its threshold was registered
- Functionality: The component will add an internal alert in such a way that, if after any trust assessment the trust value is lower than the threshold, notifications can be sent in order to take corrective actions.
• Input: String containing the identifier of a Thing Service
• Output: a Boolean indicating if the Thing Service was removed from the alerts list
• Functionality: The component will remove the Thing Service requested from the alerts list, so its trust will not be checked against a threshold in future assessments.

5.11.3. Code packaging
The code for the component implementation is packaged inside the following structure:

• package eu.betaas.taas.securitymanager.taastrustmanager contains the following classes and interfaces:
  o TaaSTrustManagerActivator.java
• package eu.betaas.taas.securitymanager.taastrustmanager.api contains the following classes and interfaces:
  o TaaSTrustManager.java
• package eu.betaas.taas.securitymanager.taastrustmanager.api.impl contains the following classes and interfaces:
  o TaaSTrustManagerImpl.java
• package eu.betaas.taas.securitymanager.taastrustmanager.taasthread contains the following classes and interfaces:
  o TrustTaaSThread.java
• package eu.betaas.taas.securitymanager.taastrustmanager.taasaggregator contains the following classes and interfaces:
  o FuzzyAggregator.java, ThingServiceTrust.java, ThingServiceTrustCalculator.java
• package eu.betaas.taas.securitymanager.taastrustmanager.taasthreadcalculator contains the following classes and interfaces:
  o DataStabilityCalculator.java, ExponentialSmoothingCalculator.java, QoSFulfillmentCalculator.java
• package eu.betaas.taas.securitymanager.taastrustmanager.taasproxy contains the following classes and interfaces:
  o TaaSBDMClient.java, TaaSQoSClient.java
6. Platform testing

In this chapter the BETaaS platform testing methodology overview is provided. It is focused on the unit and integration tests of the BETaaS platform components and basic building blocks (OSGI bundles), i.e. the validation of the different intra and inter-components communications messaging through the proper invocation of implemented software interfaces, for early diagnosis of potential integration malfunctioning and/or unexpected behaviors, and then finally lead to a successful completion of the whole integration process, which will then enable the platform release.

6.1. Unit tests

Unit tests are independent from bundles implementation; their purpose it to test the generic methods that a bundle uses. For this purpose JUnit based testing (http://junit.org/) can be used.

The following class is a basic example of JUnit test.

```java
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertTrue;

import org.junit.*;
import betaas.example.logger.blueprint.MyCustomLogServiceImpl;

public class TestClass {

    private MyCustomLogServiceImpl serviceTested;

    @Before
    public void setUp(){
        serviceTested = new MyCustomLogServiceImpl();
        serviceTested.setLogger();
    }

    @Test
    public void TestServiceMissing(){
        assertFalse(serviceTested.isRegistered("test1"));
    }

    @Test
    public void TestServiceRegistered(){
        serviceTested.registerComponent("test1");
        assertTrue(serviceTested.isRegistered("test1"));
    }
}
```
In order to run the test, the dependencies should be added to bundles pom files.

```xml
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.5</version>
  <scope>test</scope>
</dependency>
```

It will be enough to include a JUnit dependency and specify its scope for testing. In order to run the test with Maven a Test class should include Test in the name. Maven will automatically search and execute the test contained in these classes. The output will show the outcome of tests.

```
Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.018 sec - in TestClass
```

## 6.2. Integration tests

In order to test bundles integration a more complex framework is needed. For OSGi the reference test framework is called [pax-exam](https://ops4j1.jira.com/wiki/display/paxexam/Pax+Exam). This test library can execute a container and deploy the required bundles before executing tests. Because it will not be possible to instantiate all bundles due to a lack of resources, the integration tests can leverage a Mock library to simulate the behaviour of bundle dependency. We recommend using EasyMock ([http://easymock.org/](http://easymock.org/)) for this purpose.

The required dependencies for pax-exam are:

```xml
<dependency>
  <groupId>org.ops4j.pax.exam</groupId>
  <artifactId>pax-exam</artifactId>
  <version>1.0.0</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.ops4j.pax.exam</groupId>
  <artifactId>pax-exam-container-default</artifactId>
  <version>1.0.0</version>
  <scope>test</scope>
</dependency>
```
These dependencies are used to integrate pax-exam with JUnit and run a container such as Equinox from the tests. These dependencies are used to support Karaf deployment inside the integration tests. Until the release of Karaf 3.x these dependencies should not be used.

For EasyMock the required dependencies are:

```xml
<dependency>
  <groupId>org.easymock</groupId>
  <artifactId>easymock</artifactId>
  <version>3.0</version>
  <scope>test</scope>
</dependency>
```

6.2.1. Integration test with Karaf

Testing OSGi bundles requires a framework able to start the container and the required dependencies for a bundle that requires testing. OSGi uses the PAX-Exam test that allows a container to be started, configured and managed from the test class. Karaf supports PAX-Exam and provides the required dependencies in order to implement such tests using this container.

For Karaf and pax-exam the required dependencies are (the actual version depends on the current version of Karaf used to run BETaaS framework):

```xml
<dependency>
  <groupId>org.apache.karaf</groupId>
  <artifactId>apache-karaf</artifactId>
  <version>2.3.3</version>
  <type>zip</type>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.apache.karaf.tooling.exam</groupId>
  <artifactId>org.apache.karaf.tooling.exam.container</artifactId>
  <version>2.6.0</version>
  <scope>test</scope>
</dependency>
```

PAX-Exam requires a configuration class and the test classes. Configuration class tells which container to start and which bundles need to be available before starting the test, and allows injecting custom configuration to run tests.

The class configuration looks like the following:

```java
@RunWith(JUnit4TestRunner.class)
@ExamReactorStrategy(AllConfinedStagedReactorFactory.class)
public class IntegrationTestTaasBdm {
```
```java
@Configuration
public Option[] config() {
    return new Option[] {
        karafDistributionConfiguration().frameworkUrl(
            maven().groupId("org.apache.karaf").artifactId("apache-karaf").type("zip").versionAsInProject()).
            karafVersion("2.3.3").name("Apache Karaf BETaaS testing"));
    }
}

@Test
public void test() throws Exception {
    assertTrue(true);
}
```

PAX-Exam requires some annotations for the testing class, in order to tell if the container should be confined and if it should be restarted at each test of it and if can be reused, reducing time for the software building engine to run integration tests. The Karaf libraries allows to instantiate the Karaf Container from an embedded zip file and run it, in the steps prior to the tests. Among many important feature of PAX-exam, there is the possibility of providing configuration files valid only for testing purposed and setting some useful options. For an example of karaf PAX-exam, refer to: http://karaf.apache.org/manual/latest-2.3.x/developers-guide/writing-tests.html For an example of EasyMock tests for OSGi, refer to: http://easymock.org/EasyMock3_0_Documentation.html
7. Platform installation and configuration

In this chapter the BETaaS platform installation and configuration procedure is provided.

7.1.1. Platform software requirements

In order to run the BETaaS platform, the following software components have to be available and installed:

- Java Runtime Environment 1.6 or higher
- Apache Karaf 2.3.x
- Apache Maven (version 3.0.3 or higher) from [http://maven.apache.org/](http://maven.apache.org/)
- A Subversion client, e.g. Tortoise SVN downloadable from: [http://tortoisesvn.net/](http://tortoisesvn.net/)

7.1.2. Platform software download

In order to run the BETaaS platform, the related software components have to be downloaded using the SVN client and then check out the SVN repository at: [http://www.betaas.eu:8081/svn/BETaaS/](http://www.betaas.eu:8081/svn/BETaaS/)

It is recommended to also install Maven and define the Nexus repositories in the Maven settings.xml file. In this way all the libraries, dependencies and bundles can be downloaded directly by Karaf during the provisioning.

```xml
<server>
  <id>betaas-obr</id>
  <username>betaas-dev</username>
  <password>B3t4s$</password>
</server>
<server>
  <id>betas-thirdparty</id>
  <username>betaas-dev</username>
  <password>B3t4s$</password>
</server>
```

7.1.3. Platform software installation

There are two steps that need to be performed after downloading the repository of BETaaS with SVN:

- Installing and setting the configuration files
- Installing the features files to enable features in Karaf

The configuration files are under the folder 'SVN_FOLDER\software\trunk\betaas\betaas-configuration\configuration'. Their extension is .cfg and they need to be placed in the etc folder of Karaf. In this way, when a bundle is started it can access the properties stored inside the files. In the following sections the description of the configuration parameters for each component will be also provided.

Configuration files setup is not the only required installation operation. In fact Karaf can deploy a bundle and its dependency by using a provisioning mechanism based on features files. These files are defined for BETaaS inside the folder 'SVN_FOLDER\software\trunk\betaas\betaas-configuration\features'. The features files can be placed in a folder path where they will be later provided to Karaf with the 'features:addUrl' command.
7.1.4. Platform baseline dependencies

The common dependencies that are required by the BETaaS platform bundles are provided by a feature file that installs the minimum requirements in terms of bundles and also some basic features like database drivers and features (i.e. web console and Java persistency API). To make available this feature, run the Karaf command:

```
features:addUrl  file:{path-to-xml-file}/feature-betaas-baseline.xml
```

And then run the installation:

```
features:install betas-baseline
```

In this way the basic dependencies for a BETaaS gateway bundles are installed.

7.1.5. Platform Adaptation layer installation

To install the Adaptation layer in a gateway, first copy and configure the files:

- `feature-betaas-adaptation.xml` (under `trunk\betaas\betaas-configuration\features`)
- `adaptation.cfg` (trunk\betaas\betaas-configuration\configuration)

```
features:addUrl  file:{path-to-xml-file}/ feature-betaas-adaptation.xml
```

And then run the installation:

```
features:install betaas-adaptation
```

7.1.6. Platform TaaS layer installation

To install the TaaS layer in a gateway and the related database service, two files from the SVN repository are needed. The files are:

- `feature-betaas-taas.xml` (under `trunk\betaas\betaas-configuration\features`)
- `taas.cfg` (trunk\betaas\betaas-configuration\configuration)

The `taas.cfg` file needs to be placed inside the current Karaf installation directory under `etc` folder. It should also be edited to reflect the desired configuration. It also contains the configuration of the database, that has to be changed accordingly to the actual environment.

The `PATH_TO_DATABASE` need to be configured: it specifies the path where the local H2 database ([http://www.h2database.com/html/main.html](http://www.h2database.com/html/main.html)) will be installed:

```
jdbc = jdbc:h2:file:PATH_TO_DATABASE;DB_CLOSE_DELAY=-1
user = sa
pwd = sa
```

The `feature-betaas-taas.xml` installs the bundles required by the database service. In order to install the bundle run from Karaf shell:
features:addUrl file:{path-to-xml-file}/feature-betaas-taas.xml

Then the database will be available and can be installed with:

features:install betaas-taas

7.1.7. Platform Service layer installation

To install the Service layer in a gateway, first copy and configure the files:

- feature-betaas-service.xml (under trunk\betaas\betaas-configuration\features)
- service.cfg (trunk\betaas\betaas-configuration\configuration)

Make available the feature in Karaf:

features:addUrl file:{path-to-xml-file}/feature-betaas-service.xml

The PATH_TO_DATABASE need to be configured also for the Service layer: it specifies the path where the central database that will be installed at instance level:

```
jdbc = jdbc:h2:file:PATH_TO_DATABASE;DB_CLOSE_DELAY=-1
user = sa
pwd = sa
```

And then run the installation:

features:install betaas-service

7.1.8. Platform instance installation

To install a BETaaS instance in a gateway, first copy and configure the files:

- feature-betaas-instance.xml (under trunk\betaas\betaas-configuration\features)
- instance.cfg (trunk\betaas\betaas-configuration\configuration)

Then it’s necessary to configure if the current gateway is a star gateway and its address.

```
star = no
address = x.x.x.x
```

Then add the feature to Karaf:

features:addUrl file:{path-to-xml-file}/feature-betaas-instance.xml

And run the installation:

features:install betaas-instance
7.1.9. Platform gateway build

To build a gateway, install all the components to the Service layer. The BETaaS instance can be configured only in the star gateway, while the others gateways do not have such need. In each gateway, if you plan to connect it to another gateway, install the distributed OSGi and the Karaf client.

```
features:chooseurl cxf 2.7.2
features:chooseurl cxf-dosgi 1.5.0
features:install cxf-dosgi-discovery-distributed
```

In clients that need to access Zookeeper, configure the Zookeeper port and the IP address of the server:

```
config:propset -p org.apache.cxf.dosgi.discovery.zookeeper zookeeper.port 2181
config:propset -p org.apache.cxf.dosgi.discovery.zookeeper zookeeper.host=x.x.x.x
```

7.1.10. Platform instance build

To build an instance, install all the components in at least in one gateway with the Karaf Zookeeper server feature, then connect the other gateways by configuring Zookeeper client to connect the server.

```
features:chooseurl cxf 2.7.2
features:chooseurl cxf-dosgi 1.5.0
features:install cxf-dosgi-discovery-distributed cxf-dosgi-zookeeper-server
```

Configure the Zookeeper server in the star gateway. To configure the port that clients will use to connect:

```
config:propset -p org.apache.cxf.dosgi.discovery.zookeeper.server clientPort 2181
```
8. Demonstration application

In this chapter the initial design of the first BETaaS demonstration application (Intrusion Detection), built on top of the BETaaS platform is provided. It will be described with more details in the WP5 deliverables.

The demonstration application consists of four main blocks:

- the BETaaS instance creation (that includes two gateways);
- the things connection;
- the Intrusion Detection application installation;
- the data retrieval.

First, the instance is made up by starting the two gateways. By means of the IM component, gateways automatically discover each other and perform the necessary operations to start belonging to the same instance and sharing the same resources.

Then, things (presence sensors) are connected to the platform. The adaptation layer detects them and starts the procedure to instruct CM, TaaSRM and BDM in order to create the corresponding thing services.

After that, the Intrusion Detection application is started. It automatically requests the platform to be installed on it, i.e. it requests all the needed resources to be allocated. This request is performed by the application that is external to the platform, and sent to the SM that represents the platform primary interface to the platform clients. In turn, SM starts the installation procedure requesting the TaaSRM to find the available thing
services to be used to fulfil the application requirements. The context management (CM) capability will be used for that. A negotiation is then performed with SECM and QoSM to establish the suitable resources that will accommodate the application security and service quality specifications.

Once the installation is completed, the application starts to periodically request presence data at home and the result will be shown on a dynamic Web page. Finally, the user may ask for some kind of presence analysis (e.g. the last presence detected by all the used thing services) by means of the web application. This step involves the BDM component that is in charge of storing and retrieving the historical data.
9. Conclusions

This document presents the results achieved in WP4 during the first phase of the project, i.e. the initial release of the BETaaS software platform. The results of WP4 Tasks 4.1 (Development and integration coordination), WP4 Tasks 4.2 (Technological baseline definition), Task 4.3 (Platform development) and Task 4.4 (Integration tests design and execution) are presented. The document is the accompanying textual specification of the major deliverable result: the packaged developed software that implements the BETaaS platform. The document and the developed software constitute the overall deliverable output.

The work have been carried out in close cooperation and synchrony with WP1-2-3, properly integrating the delivered results in a coherent and uniform manner, and then finally releasing the overall BETaaS platform. The progress of these tasks will be one of the major drivers of the work package dedicated to validation trials (WP5).
10. References

[1] Building the Environment of the Things as a Service (BETaaS) - EU FP7 Project, Deliverable D3.1.1 – BETaaS Architecture