SeaClouds Project

D5.4.1 Initial version of sw platform

Project Acronym	SeaClouds
Project Title	Seamless adaptive multi-cloud management of service-based applications
Call identifier	FP7-ICT-2012-10
Grant agreement no.	610531
Start Date	1st October 2013
Ending Date	31st March 2016

Work Package	WP5 Integration, infrastructure delivery and GUI
Deliverable code	D5.4.1
Deliverable Title	Initial version of sw platform
Nature	Prototype
Dissemination Level	Public
Due Date:	M12
Submission Date:	10th October 2014
Version:	1.0
Status	Final
Author(s):	Miguel Barrientos (UMA), Jose Carrasco (UMA), Javier Cubo (UMA), Francesco D'Andria (ATOS), Adrián Nieto (UMA), Román Sosa (ATOS)
Reviewer(s)	Francesco D’Andria (ATOS)

Dissemination Level

Project co-funded by the European Commission within the Seventh Framework Programme		
PU	Public	X
PP	Restricted to other programme participants (including the Commission)	
RE	Restricted to a group specified by the consortium (including the Commission)	
CO	Confidential, only for members of the consortium (including the Commission)	

Table of Contents

List of Figures

List of Tables

Executive Summary

The objective of this deliverable is to give an overview of the first SeaClouds integrated prototype as outcome of the implementation work done in the technical work packages WP3 and WP4, where the main activities so far are concentrated in Deliverables D3.1, D4.1 and D4.2.

The document overviews the tools implemented by M12 and describe all the necessary process to install and configure the system.

1. Introduction

The deliverable D5.4.1 is the first deliverable of D5.4.X saga. It overviews the SeaClouds software tools which have been implemented during the course of the first twelve months.

We have tried to keep the deliverable short, in order not to repeat any information already available from previous deliverables. These deliverables have been referenced where appropriate. Anyway, some important aspects related to the implementations are implicitly described in this document.

The structure of the document is as follows.

First, in Section 3, we provide a list of services/functionalities implemented at M12 and brief descriptions of them. Section 4 describes how the code is organized, and Section 5 explains how to configure and install, as well as how to use the prototype. Last, Section 5 concludes the deliverable.

1.1 List of Acronyms

Acronym	Definition
SaaS	Software-as-a-Service
PaaS	Platform-as-a-Service
IaaS	Infrastructure-as-a-Service
QoS	Quality of Service
QoB	Quality of Business
SLA	Service Level Agreement
TOSCA	Topology and Orchestration Specification for Cloud Applications
CAMP	Cloud Application Management for Platforms
GUI	Graphical User Interface
API	Application Programming Interface
APP	Application
DB	Database
WP	Work Package
DAM	Deployable Application Model
YAML	YAML Ain't Another Markup Language

Table 1. Acronyms

2. Services and functionalities

Currently, several SeaClouds goals are accomplished. In Figure 1 we present the services and components, as well as the interaction among the components of the SeaClouds platform which should be implemented at the end of the project life-cycle.

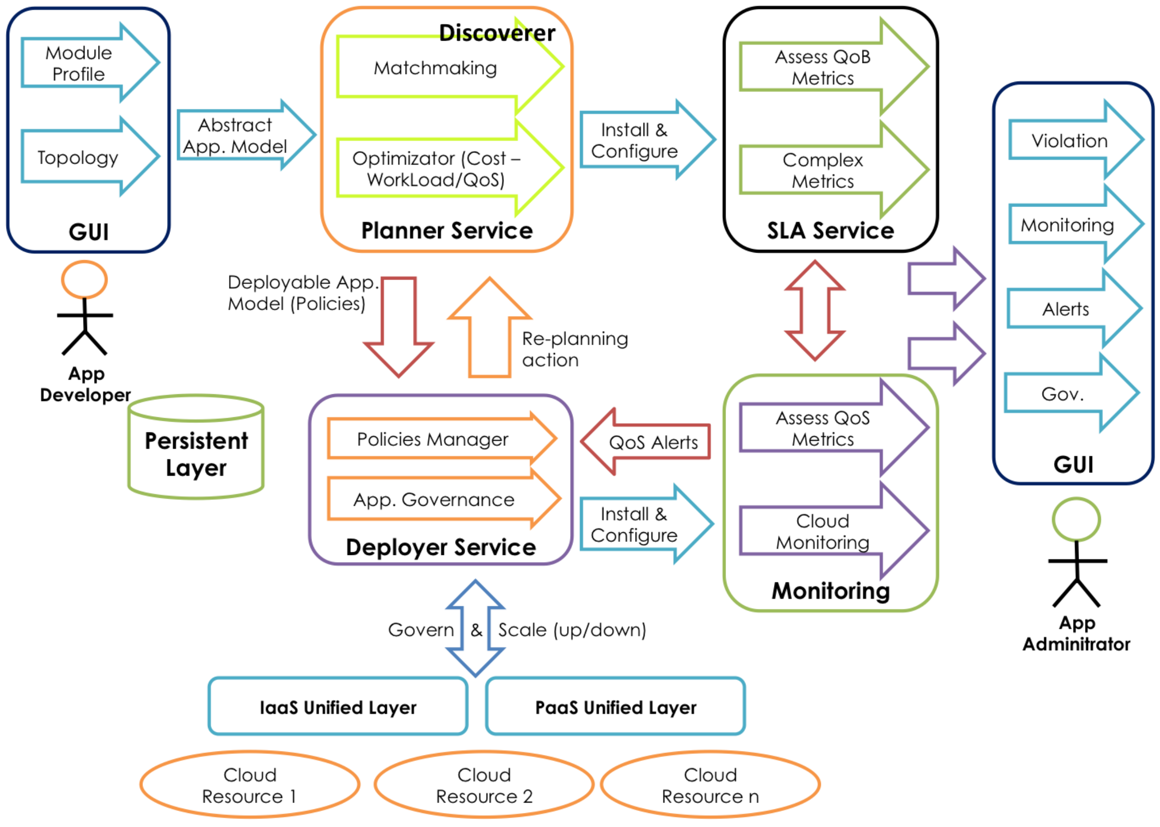


Figure 1: SeaClouds components/tools to be developed

The Figure 1 represents the steps necessary to carry out an application deployment from the initial stage where the Application Developer (end-user) provides the Application Model consisting of the Module Profile and the Topology representing the connections among the modules of the cloud application to be deployed (other elements as the SLA restrictions and policies are considered by SeaClouds), as is described in detail in Deliverable D3.1 [1] related to the design-time. After the Abstract Application Model has been specified, SeaClouds starts the Discoverer and Planner process (note that the Discoverer is included into the Planner service in the figure). It consists of two processes: Matchmaking and Optimizer, also explained in Deliverable D3.1 [1].

Then, as result of the Planner, a Deployable Application Model (DAM), which specifies the cloud services used to distribute the application, is generated by the Planner. The DAM generated allows the deployment of the application's module over heterogeneous IaaS and PaaS, using the Deployer component, more detailed in Deliverable D4.1 [2]. Once the application are deployed and managed by the Deployer, this notifies to the Monitor for initializing the monitoring of the applications, which connects with the SLA service to manage the violations of the QoS and QoB, and properties.

A GUI as Dashboard is also used for the interaction with the Application Administrator. Note that a Persistent Layer should be considered to maintain a continue store (e.g., the QoS violations).

Since the SeaClouds project is in its first year, in Figure 2, only the SeaClouds functionalities implemented in this first iteration of the project, month M12, are depicted (green sticks), marking the functionalities not considered for the review demo (red cross) and those which are under implementation (TODO).

Along this document we will explain the different components implemented for this initial version of the software platform of SeaClouds. We refer to the technical documents, D3.1, D4.1 and D4.2 (for the API) [3] to know more about every component or service. In fact, the services or functionalities not implemented yet or under implementation has not been included in this document.

For example, the Matchmaking process is a work in progress, and we pretend to present a simple matchmaking in the demo review (depending on the results on some issues we are analyzing as regards the algorithms we are implementing), although some implementation details have to be concreted, so we have decided to include the description of the Matchmaking in the next deliverable, and here only in the next section a brief section will be dedicated to the Discoverer and the Planner.

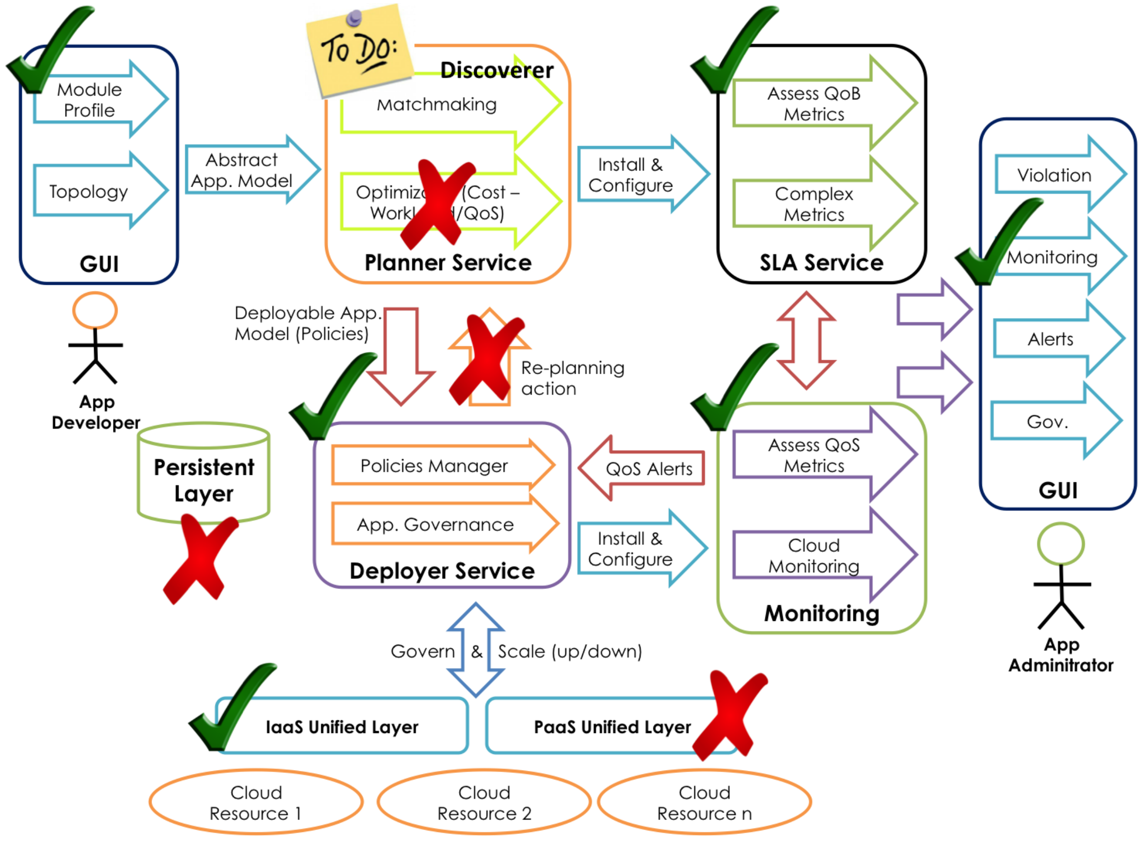


Figure 2: SeaClouds components/tools available at M12

2.1 Discoverer and Planner

In the Deliverable D3.1, we have already presented the functionality of these two components, thus here we only mention the main functionalities of both.

Together with the description of the user input (the Application Model), the Planner component needs to know which are the possible services available from cloud providers. Multiple cloud resources can be offered by cloud providers at different levels of abstraction. This is the main taks of the Discoverer.

On the other hand, the Planner component is in charge of providing an Abstract Deployment Plan (ADP) that defines where each application module will be deployed (and used to generate the final Deployable Plan, see next section).

Given a set of modules with their requirements, the topology of the application and a set of cloud resources, the Planner will generate an ADP that meet the requirements specified by the user. The ADP includes the concrete services associated with each Base module and the policies to manage the scaling mechanism of each module.

The generation of the ADP can be performed in two steps (see Deliverable D3.1 for more details): 1) Matchmaking: this first step aims to identify the cloud resources that are suitable to allocate each module, and 2) Optimization: once a set of suitable cloud services have been identified for each modules, an optimization process can be performed.

2.2 Deployer

In the previous Deliverable D4.1, we introduced the Deployer as the SeaClouds component responsible for the application module distribution across selected cloud providers. In this section, we describe the current status of the multideployment description and the support technologies, following the figure that represents the initial architecture of the deployer, proposed in the Deliverable D4.1**.**

In the architecture it is depicted how the deployer is composed by several elements. The main element of our SeaClouds Deployer is the Deployer Engine. The Deployer Engine receives a Deployable Application Plan through its Deployer API and executes the DAM. As the Deployer Engine is cloud-agnostic it is able to deploy applications to the different cloud provider services using multiple Cloud Adapters. Once the application has been deployed, the Deployer Engine uploads the live model which contains the data structure (components and relationship between these) in order to maintain topology of the application.

Currently, for the first draft we use Brooklyn as Deployer Engine to accomplish the multideployment of the applications component and the Live Model generation and management. The applications components could be deployed over different cloud providers simultaneously (using jClouds [4] like Cloud Adapters). Therefore, the application topology status is stored in the Live Model using several XML files. Although, these files are stored over the local file system, they could be deployed over a BlobStore (or PaaS Store service) of the any cloud provider supported by the Cloud Adapters. In any case, the Deployer Engine is charge of modify the Live Model to maintain the integrity with the live real distribution status.

Regarding the supported technologies, PHP application support has been added to Brooklyn in order to allow the deployment, management and monitoring of our early case study, Nuro Application which was developed using PHP (Section 5 of the Deliverable 4.1).

When we add PHP support to Brooklyn, we are adding PHP application description to the Deployable Application Model automatically, so currently this technology could be used from the DAM.

2.3 Monitor

In this section we will describe the status of the current implementation regarding the Monitor component in SeaClouds. Monitor component is in charge of retrieving the monitoring data from deployed applications, making it available to the rest of the components in SeaClouds. Also, it is responsible for the control and enforcement of QoS properties and SLA, as well as forwarding violations on these properties to the interested subscribed modules. The retrieved data and related information must be completely available through the Monitor API.

Following the monitor architecture introduced in the Deliverable 4.1, the Monitor functionality is centralized on the Monitoring Manager, which acts as a registry for new applications deployed and it's also the mechanism to manage the data generated from the Monitoring agents by retrieving it, exposing it through the Monitor API and storing it for later usage and analysis.

Regarding the Monitoring Agents, and as an initial approach, we make use of Brooklyn as the main Monitoring Agent, making use of the management capabilities and mechanisms incorporated in this tool, such as sensors, data feeds, enrichers and policies. These allows us to define new data retrieving mechanisms for any application managed by SeaClouds that will be gathered as QoS properties on the Monitor platform. An overall view of the kind of metrics retrieved by Brooklyn can be found in the Annex C of Deliverable 4.1. These metrics cover different kind of application and infrastructure monitoring giving a general description of the managed cloud system.

In the following releases of the Monitor component, several other Monitoring Agents will be included, making use of the abstraction layer that these Agents propose in order to retrieve a wider family of metrics from the rest of the levels of the application.

2.4 Dashboard

The current functionality implemented presents a very early version of the proposal of the Unified Dashboard, in which SeaClouds is working (and whose objective is to present more in detail for the next Deliverable, D4.5 Unified dashboard and revision of Cloud API). The dashboard will be used to access to the platform services. It is designed to provide a clear and simple way to interact with SeaClouds in all stages of the lifecycle of an application, starting from the application module definition (module profile) to the analysis of application's SLA violations.

In order to establish a common GUI across the whole SeaClouds platform, we need to specify the framework that we're going to use during the development process of the dashboard. The organization of the Dashboard is divided into several sections which could be found in Deliverable 4.2. In this stage, it is possible to deploy and monitor the applications deployed using Brooklyn as Deployer Backend.

2.5 SLA Service

The SLA Service enables the Service Level Agreements (SLA) management of business oriented policies. The SLA Service is an implementation of the WS-Agreement specification. In this section we will describe the status of the current implementation of the SLA Service in SeaClouds.

In Deliverable D4.1 the architecture of the SLA Service was presented. The main responsibilities of the SLA service are:

- Generating and storing WS-Agreement templates and agreements.
- Assessing that all the agreements (SLA guarantees) are respected assessing the business penalties.

Currently, for the first draft, the basic capabilities of Repository, SLA Manager and Assessment components have been developed. This includes the storing and retrieval of WS-Agreement entities, with some search capabilities, and a preliminary assessment of existing agreements. In terms of integration, Brooklyn is used as Monitoring Service to accomplish the constraints evaluation.

3. Code description

This section describes the code implemented in the initial version of the SeaClouds software platform, related mainly to the run-time environment, consisting on the deployer, monitor and SLA components. Also, implementations related to the dashboard have been performed, although in an early stage; so the details about it were already described in Deliverable D4.2. As regards the matchmaking process, as aforementioned, it will be described in next deliverables.

3.1 Deployer: Brooklyn Concepts

The central concept in a Brooklyn deployment is that of an entity. An entity represents a resource under management, either base entities (individual machines or software processes) or logical collections of these entities.

Fundamental to the processing model is the capability of entities to be the parent of other entities (the mechanism by which collections are formed), with every entity having a single parent entity, up to the privileged top-level application entity.

Entities are code, so they can be extended, overridden, and modified. Entities can have events, operations, and processing logic associated with them, and it is through this mechanism that the active management is delivered.

The main responsibilities of an entity are:

- Provisioning the entity in the given location or locations.
- Holding configuration and state (attributes) for the entity.
- Reporting monitoring data (sensors) about the status of the entity.
- Exposing operations (effectors) that can be performed on the entity.
- Hosting management policies and tasks related to the entity.

3.1.1 Sensors and Effectors

Sensors (activity information and notifications) and effectors (operations that can be invoked on the entity) are defined by entities as static fields in the Entity subclass.

Sensors can be updated by the entity or associated tasks, and sensors from an entity can be subscribed to by its parent or other entities to track changes in an entity's activity.

Effectors can be invoked by an entity's parent remotely, and the invoker is able to track the execution of that effector. Effectors can be invoked by other entities, but this functionality has to be used sparingly to prevent too many managers.

An entity consists of a Java interface (used when interacting with the entity) and a Java class. For resilience, it is recommended to store that the entity's state in attributes is stored. If internal fields can be used, then the data will be lost on Brooklyn restart upon Brooklyn restarting, and may cause problems if the entity is to be moved to a different Brooklyn management node.

3.1.2 Configuration

All entities contain a map of config information. This can contain arbitrary values, typically keyed under static ConfigKey fields on the Entity sub-class. These values are inherited, so setting a configuration value at the application level will make it available in all entities underneath unless it is overridden.

Configuration is propagated when an application "goes live" (i.e. it becomes "managed", either explicitly or when its start() method is invoked), so config values must be set before this occurs.

Documentation of the flags available for individual entities can normally be found in the javadocs.

The @SetFromFlag annotations on ConfigKey static field definitions in the entity's interface is the recommended mechanism for displaying configuration options. Currently, the camelCase notation is used to define the ConfigKey Flags, such as "appUrl", but we intend to use the C notation, which is used by TOSCA and CAMP, e.g.: "app_rule".

3.2 Deployer: PHP Support

In this section, we describe how have been developed the PHP support in the deployment engine currently used by SeaClouds, Brooklyn.

3.2.1 Implementing an Entity

All entity implementations inherit from AbstractEntity, often through one of the following:

- SoftwareProcess: if it is a software process
- VanillaJavaApp: if it is a plain-old-java app
- JavaWebAppSoftwareProcess: if it is a JVM-based web-app
- WhirrEntity: if it is a service launched using Whirr
- DynamicGroup: if it is a collection of other entities

Software-based processes tend to use drivers to install and launch the remote processes onto locations which support that driver type.

For example, AbstractSoftwareProcessSshDriver is a common driver superclass, targetting SshMachineLocation (a machine to which Brooklyn can ssh).

The various SoftwareProcess entities (as aboveand some of the exemplars listed at the end of this page) have their own dedicated drivers.

Entities model the deployment elements which are used during the application livecycles. The methods on the entity interface are its effectors; the interface also defines its sensors. However, the entities need mechanisms to manage the real elements in the deployment environment where they are being executed, the drivers. For example, an entity could display an operation to restart a service.

Maybe, this entity does not know the mechanisms to carry out this operation, which could depend on the runtime environment. In this regard, the entity uses a driver which knows and contains the implementation necessary to execute the operations in the final runtime environment, establish the sensor connections, etc. Typically, the implementation of the drivers is based on Ssh technology which allows commands and programs to be executed in the final runtime, remotely and a standardized way.

Apache Web server (httpd) is modeled using an entity which displays some operations (effectors) such as install, configure, etc. Although this entity contains the logic of several operations, it does not maintain knowledge over the final system. So, the driver describes the mechanisms necessary to interact with the real component. In this case, the server httpd is installed in an AWS (http://aws.amazon.com/ec2/) virtual machine with Centos (https://www.centos.org/). The driver contains the Ssh commands to carry out the operations, e.g. when the entity is installed it then uses the driver operation to send and run the bash commands to install the httpd in the deployment environment selected, Centor VM in this case. So, the server installation management is, encapsulated, wrapped, hidden and centralized by the driver.

Entities are created through the management context (rather than calling the constructor directly). This returns a proxy for the entity rather than the real instance, which is important in a distributed management plane.

Finally, there is a collection of traits, such as Resizable, in the package brooklyn.entity.trait. These provide common sensors and effectors on entities, supplied as interfaces. Choose one (or more) as appropriate.

3.2.2 Key Steps to Implement an Entity

- Create your entity interface, extending the appropriate selection from above, to define the effectors and sensors.
- Include an annotation like @ImplementedBy(YourEntityImpl.class) on your interface, where YourEntityImpl will be the class name for your entity implementation.
- Create your entity class, implementing your entity interface and extending the classes for your chosen entity super-types. Naming convention is a suffix "Impl" for the entity class.
- Create a driver interface, again extending as appropriate (e.g. SoftwareProcessDriver). The naming convention is to have a suffix "Driver".

- Create the driver class, implementing your driver interface, and again extending as appropriate. The naming convention is to have a suffix "SshDriver" for an ssh-based implementation. The correct driver implementation is found using this naming convention, or via custom namings provided by theBasicEntityDriverFactory.
- Wire the public Class getDriverInterface() method in the entity implementation, to specify your driver interface.
- Provide the implementation of missing lifecycle methods in your driver class (details below)
- Connect the sensors from your entity (e.g. overriding connectSensors() of SoftwareProcessImpl).. See the sensor feeds, such as HttpFeed and JmxFeed.

Any JVM language can be used to write an entity. However the use of pure Java is encouraged for entities in the Brooklyn core.

3.2.3 PHP Entities

Currently, SeaClouds uses Brooklyn as a deployer, which is in charge of the distribution of the applications' components over different locations (clouds, local or remote VM, etc).

In this regard, we intend to add several features to Brooklyn in order to adapt its capabilities to the goals of SeaClouds. At the moment, Brooklyn supports Java applications only, but, although Java is one of the most used languages, SeaClouds includes support for other languages, for example, PHP, as one of its goals.

In this document, we describe the architecture to add for adding PHP support to Brooklyn. We have based our development on the Brooklyn structure, as we have mentioned above, to take of advantage of current Brooklyn management mechanisms.

Figure 3. Overview of the PHP integration class diagram describes the class diagram used to extend Brooklyn. We have created one abstraction level which contains several abstract classes and interfaces, PhpWebApp*, to define the common behavior and features (sensors, effectors, general methods, etc) of PHP Web Applications. These classes have to be extended and implemented to add the final PHP support to Brooklyn. In this case we have described the Apache httpd for the PHP system support.

The abstraction level extends the generic classes of Brooklyn which allows several general mechanisms to be reused. For example, SoftwareProcess and SoftwareProcessDriver represent, respectively, a generic process entity and its driver. So, the PhpWebSoftwareProcess interface models a generic Php Web App. In order to follow the baseline described in the preceding sections, we have also defined a generic PHP driver.

The Apache classes are based on the abstraction level and contain the effector and sensor implementations, define a final deployment PHP system, and how it is to be managed. We will explain these classes in the following sections.

Please note that at the moment, we have only added Apache for PHP support and if in the case we add another system, e.g., JBoss (http://jbossweb.jboss.org/modules/php.html), we will also base it on the abstraction level.

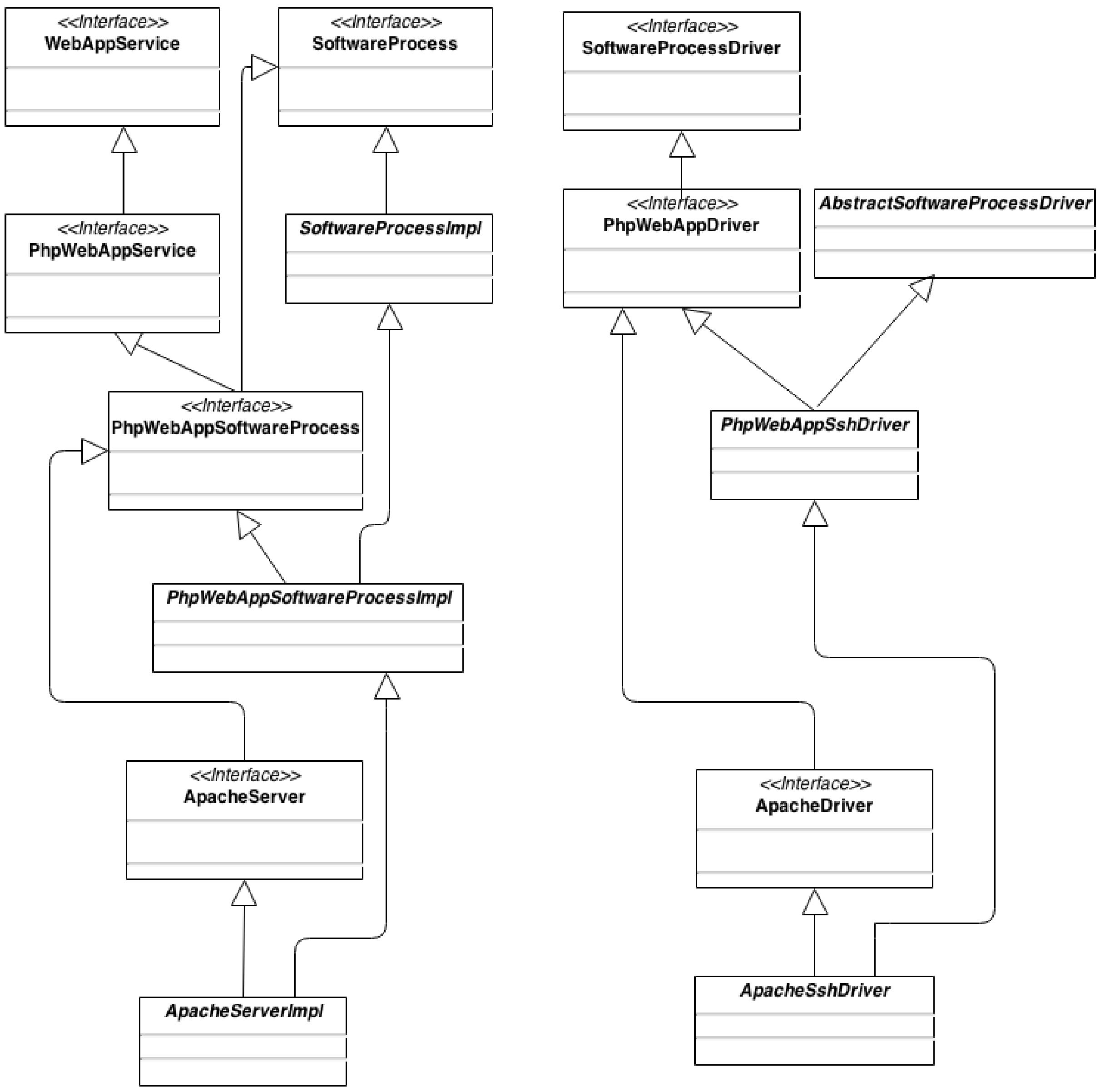


Figure 3. Overview of the PHP integration class diagram

3.2.4 Generic PHP classes (abstraction level)

In this section we present the aforementioned abstraction level class diagram (Figure 4). We have already stated that the abstraction level contains the generic logic necessary for describing PHP web applications using several classes of Brooklyn to integrate the new features in the Brooklyn's behavior, the latest are shown with a green background.

These classes are only mentioned in brief and will not be explained in detail here (the interested reader can see the official documentation to obtain more information about this classes [5]). Therefore, here we are going to review just the most important class of the diagram.

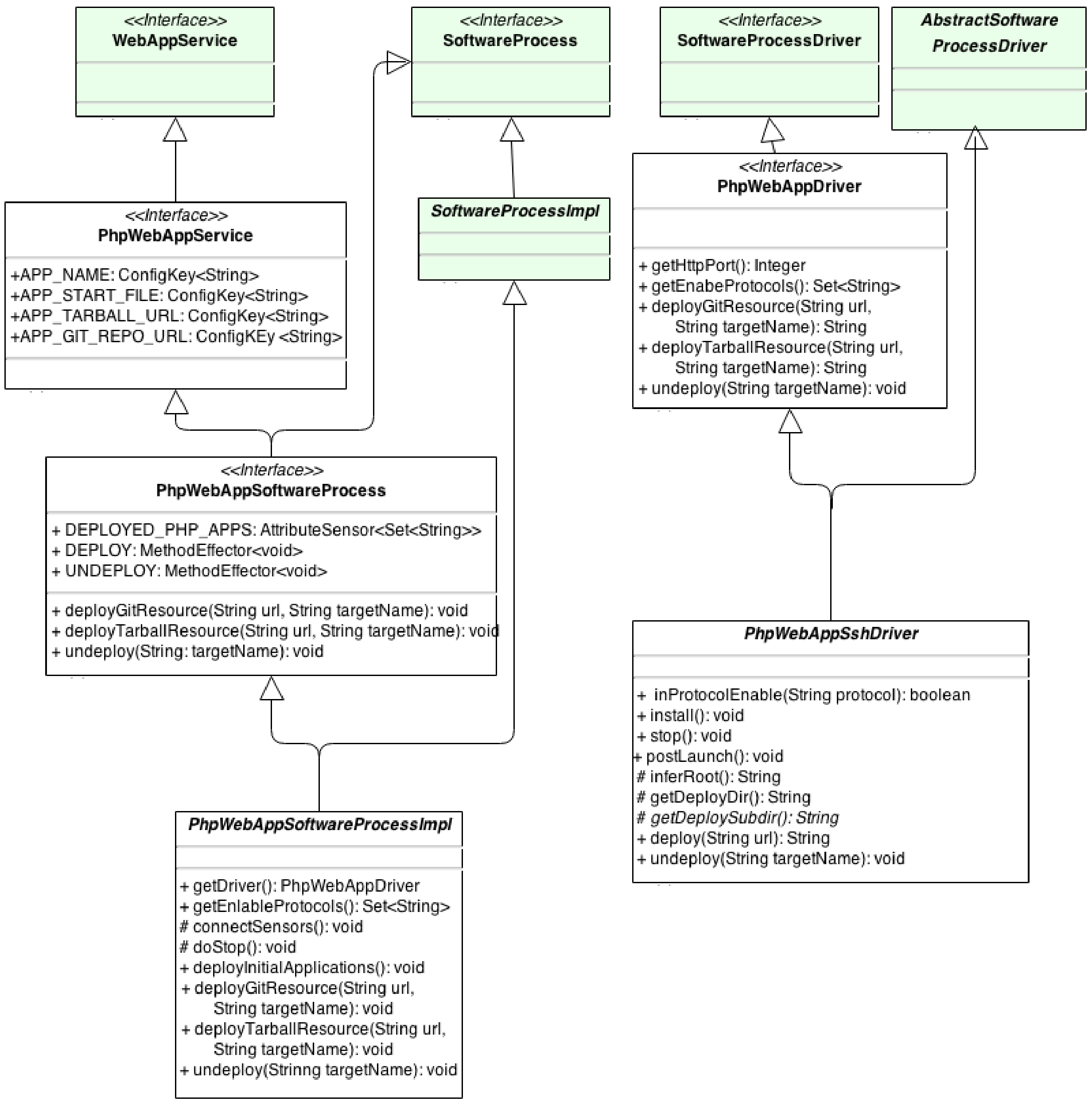

<<Interface>>
WebAppService
<<Interface>>
SoftwareProcess
<<Interface>>
SoftwareProcessDriver
AbstractSoftware ProcessDriver
SoftwareProcessImpl
<<Interface>>
PhpWebAppDriver
+ getHttpPort(): Integer
+ getEnabeProtocols(): Set<String>
+ deployGitResource(String url, String targetName): String
+ deployTarballResource(String url, String targetName): String
+ undeploy(String targetName): void
<<Interface>>
PhpWebAppService
+APP_NAME: ConfigKey<String>
+APP_START_FILE: ConfigKey<String>
+APP_TARBALL_URL: ConfigKey<String>
+APP_GIT_REPO_URL: ConfigKEy <String>
<<Interface>>
PhpWebAppSoftwareProcess
+ DEPLOYED_PHP_APPS: AttributeSensor<Set<String>>
+ DEPLOY: MethodEffector<void>
+ UNDEPLOY: MethodEffector<void>
+ deployGitResource(String url, String targetName): void
+ deployTarballResource(String url, String targetName): void
+ undeploy(String: targetName): void
PhpWebAppSshDriver
+ inProtocolEnable(String protocol): boolean
+ install(): void
+ stop(): void
+ postLaunch(): void
# inferRoot(): String
# getDeployDir(): String
# getDeploySubdir(): String
+ deploy(String url): String
+ undeploy(String targetName): void
PhpWebAppSoftwareProcessImpl
+ getDriver(): PhpWebAppDriver
+ getEnlableProtocols(): Set<String>
# connectSensors(): void
# doStop(): void
+ deployInitialApplications(): void
+ deployGitResource(String url, String targetName): void
+ deployTarballResource(String url, String targetName): void
+ undeploy(Strinng targetName): void

Figure 4. Abstraction level class diagram

3.2.4.1 PHPWebAppService

This interface is the most abstract level of a PHP entity and only contains the ConfigKeys necessary to describe the application. They are used in the application declaration in the YAML BluePrint to specify, for example, where the application deployment artifact is stored from different services (git, etc…), the start file application, etc. Moreover, this interface extends WebAppInterface which indicates that an entity is an Application Web. In what follows, we specify the ConfigKey:

- **APP_NAME**: Name of the application.
- **APP_START_FILE**: The PHP application file start.
- **APP_GIT_REPO_URL**: The git repository where the application is hosted.

- **APP_TARBALL_URL**: The source where the tarball resource is hosted.

Regarding database connection, a generic way to determine the connection in a PHP application is using a file which defines the params necessary to establish the connection with the database target and the application logic is charge of establish the connection. To add database connection support our initial proposal defines the next config keys for allowing the connection configuration.

ConfigKey	Description
DB_CONNECTION_FILE_CONFIG	File which contains the database connection params.
DB_CONNECTION_CONFIG_PARAMS	Map to define the params of the database connection and their values.

Table 2. Database connection datbase

3.2.4.2 PHPWebAppSoftwareProcess

This interface extends the PHPWebAppService and SoftwareProcess indicating that it is a PHP entity. SoftwareProcess defines several generic ConfigKeys and Effectors for managing a process software generic such as [6]:

ConfigKey	Description
START_TIMEOUT	timeout allowed to init the software.
SUGGESTED_VERSION	suggested stable version of the software
DOWNLOAD_URL	URL from where to download the software.
INSTALL_DIR	folder where to install the software modeled.
SUGGESTED_INSTALL_DIR	alternative installation folder.
RUN_DIR	folder to run the software (which could be installed or not).etc

Table 3. PhpWebAppSoftwareProcess ConfigKeys

In addition, PhpWebAppSoftwareProcess describes the effectors to deploy and undeploy applications.

- **DEPLOY_GIT_RESOURCE** and **DEPLOY_TARBALL_RESOURCE**: it allows an entity to be deployed using the url where it is stored (a git repo or an tarball url) . The effector returns the id of the deployed application.
- **UNDEPLOY**: it allows undeploying an entity which has already been deployed using the id application.
- **DEPLOYED_PHP_APPS**: it contains the deployed application's ids to allow them to be managed. It is updated each time the deploy/undeploy effectors are executed.

3.2.4.3 PHPWebAppSoftwareProcessImpl

PhpWebAppSoftwareProcessImpl is an abstract class that implements PhpWebAppSoftware and extends SeoftwareProcessImpl.

This class implements the generic behavior of the entity, defines the stop and initialize methods, it allows the entity enable port, etc, to be known and defines new management methods (as connectSensors()). Please note, that the getter and setter methods are deleted from the UML class and the naming convention is a suffix "Impl" for the entity class. Below, we present the definition methods:

- **getEnableProtocols()**:it allows the enable protocols to be known, e.g. http (by default) and https.
- **connectSensors()**: it connects the generic sensor with the data source, so we can say that it is a generic method because almost all sensors used feed off the final PHP support system, e.g. Apache status server page, etc. Thus, this method has to be used @Override (and used) by the final class which specifies this abstract class.
- **doStop()**: it allows the system to be stopped. Again, it is a generic method and it needs to be extended.
- **getDriver()**: it returns the driver used to manage this entity.
- **deploy(String url)/undeploy(String targetName)**: we have mentioned that the management mechanisms are contained in the driver, so these methods send the deploy/undeploy request to the entity driver, which is described by the getDriver() method. Moreover, these methods manage (update) the deployed application sensor when an application is added or removed.

3.2.4.4 PhpWebAppDriver

The PhpWebAppDriver interface extends the generic Brooklyn Driver classes, in fact it extends the SoftwareProcessDriver and the ProcessDriver.

Only. It defines several methods that have to be implemented in a specific class. In this case, the most important methods could be deploy and undeploy which are used by the entity to manage these tasks.

3.2.4.5 PhpWebAppSshDriver

The PhpWebAppSshDriver implements the aforementioned interface. It determines several methods that allow the management of the real PHP support system:

- **isProtocolEnalbe()**: discovers the protocols which could be used over the real system as http and https.
- **install()**: it allows the software needed to run the application to be installed and, in this case, this method has to be @Override to install the deployment environment. However, it contains the installation of the generic components necessary during the lifecycle, in this case PHP and git clients.
- **stop()**: obviously, it stops the execution environment. Again, it has to be @Override

in any given class.

- **postLaunch()**: once the PHP environment has been installed and configured this method carries out the management task needed by Brooklyn, like, for instance, connect the ROOT_URL sensor that contains the root path of a server (which are generated by **inferRoot()** method).
- **getDeployDir()** and **getDeploySubdir()**: return the folders which install and run the applications.
- **deploy_git_resource**, **deploy_tarball_resource** and **undeploy()**: they contain logic to deploy and undeploy an application in the final deployment environment. They implement a generic methodology to carry out these tasks, but they could be @Override in the specific class to define a new deployment methodology.

3.2.4.6 Apache Httpd Server

Apache httpd is the deployment environment selected to deploy and run a PHP application. As we have already mentioned, the classes used to model this component are based on the abstraction level explained in the previous section. Below, the class diagram used to add Apache support for Brooklyn is shown in Figure 5. Please note, that the names of the classes may be modified.

3.2.4.7 ApacheServer

ApacheServer interface extends the PhpSoftwareProcess to indicate that it is a PHP entity and that wraps the final deployment environment to deploy and run PHP applications. It determines the ConfigKey and the sensors of the Apache installation.

- **SUGGESTED_VERSION**: suggested stable version of the ApacheServer.
- **DEPLOYMENT_TIMEOUT**: represents the allowed timeout to deploy an application.
- **INSTALL_DIR**: folder where Apache is installed.
- **CONFIGURATION_DIR**: folder where the apache configuration files are stored.
- **AVAILABLE_SITES_CONFIGURATION_FOLDER**: folder that contains the configuration and application deployment and run folders where the applications are deployed.
- **DEPLOY_RUN_DIR**: deployment and execution application folder. It is given from AVAILABLE_SITES_CONFIGURATION_FOLDER.
- **DEFAULT_GROUP**: default by Apache, it is needed to run the applications deployed in DEPLOY_RUN_DIR.
- **HTTP_PORT**: Http port where Apache is listening
- **SERVER_STATUS**: URL where the status of the server can be consulted

In order to allow the management of the PHP applications, we have added several sensor:

Sensor	Description
TOTAL_ACCESSES	Number of accesses.

TOTAL_KBYTES	Total traffic in KBytes.
CPU_LOAD	Percentage CPU load.
UP_TIME	Total up time (in seconds).
REQUEST_PER_SEC	Requests per second.
BYTES_PER_SEC	Bytes processed per second.
BYTES_PER_REQ	Bytes per request.
BUSY_WORKERS	Number of busy workers.

Table 4. Apache sensors availables

3.2.4.8 ApacheServerImpl

ApacheServerImpl class implements the ApacheServer to indicate its PHP entity nature and extends PhpWebAppSoftwareImpl (see Figure 5). It represents the Apache installation used in the deployment environment.

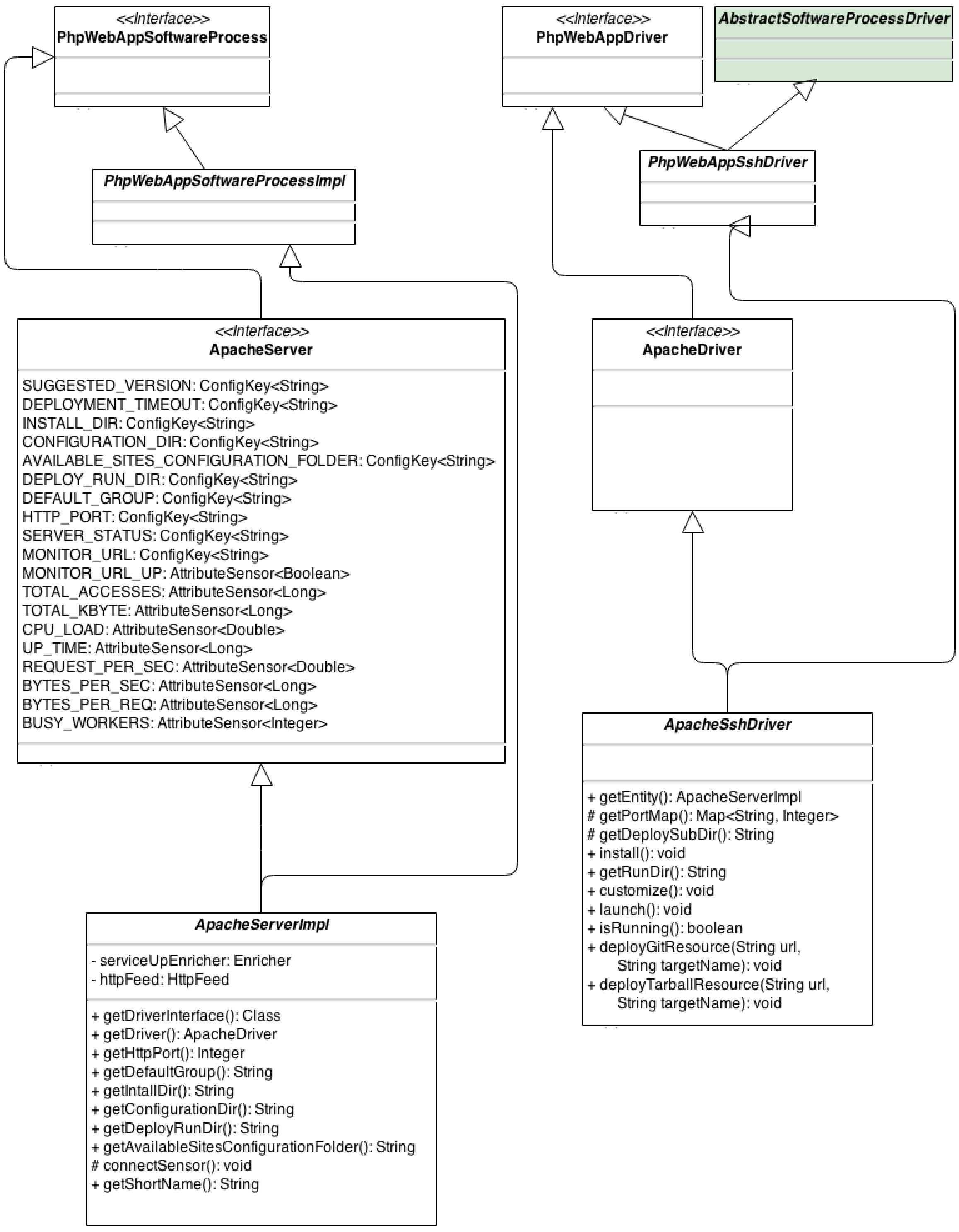
It defines an enricher:

- **serviceUpEnricher**: to wrap the start up of the server. It provides the data needed for the sensor isEnable.

In addition, it defines an HttpFeed to carry out the sensor pulling. This object will be used by the methods in charge of the sensor initialization.

This class defines several getter and setter methods which are used by the generic classes (PhpWebAppSoftwareProcessImpl and PhpWebAppSshDriver) to manage the entity.

- **connectSensor()**: it is responsible for initializing the sensors. It uses the httpFeed to pull the data source. Please note, that this method uses an @Override the super method implemented in PhpWebAppSoftwareProcessImpl.



<<Interface>> PhpWebAppSoftwareProcess
PhpWebAppSoftwareProcessImpl
<<Interface>> PhpWebAppDriver
AbstractSoftwareProcessDriver
PhpWebAppSshDriver
<<Interface>> ApacheServer
SUGGESTED_VERSION: ConfigKey<String>
DEPLOYMENT_TIMEOUT: ConfigKey<String>
INSTALL_DIR: ConfigKey<String>
CONFIGURATION_DIR: ConfigKey<String>
AVAILABLE_SITES_CONFIGURATION_FOLDER: ConfigKey<String>
DEPLOY_RUN_DIR: ConfigKey<String>
DEFAULT_GROUP: ConfigKey<String>
HTTP_PORT: ConfigKey<String>
SERVER_STATUS: ConfigKey<String>
MONITOR_URL: ConfigKey<String>
MONITOR_URL_UP: AttributeSensor<Boolean>
TOTAL_ACCESSES: AttributeSensor<Long>
TOTAL_KBYTE: AttributeSensor<Long>
CPU_LOAD: AttributeSensor<Double>
UP_TIME: AttributeSensor<Long>
REQUEST_PER_SEC: AttributeSensor<Double>
BYTES_PER_SEC: AttributeSensor<Long>
BYTES_PER_REQ: AttributeSensor<Long>
BUSY_WORKERS: AttributeSensor<Integer>
<<Interface>> ApacheDriver
ApacheSshDriver
+ getEntity(): ApacheServerImpl
# getPortMap(): Map<String, Integer>
# getDeploySubDir(): String
+ install(): void
+ getRunDir(): String
+ customize(): void
+ launch(): void
+ isRunning(): boolean
+ deployGitResource(String url, String targetName): void
+ deployTarballResource(String url, String targetName): void
ApacheServerImpl
- serviceUpEnricher: Enricher
- httpFeed: HttpFeed
+ getDriverInterface(): Class
+ getDriver(): ApacheDriver
+ getHttpPort(): Integer
+ getDefaultGroup(): String
+ getIntallDir(): String
+ getConfigurationDir(): String
+ getDeployRunDir(): String
+ getAvailableSitesConfigurationFolder(): String
# connectSensor(): void
+ getShortName(): String

Figure 5. Apache Web Server class diagram

3.2.4.9 ApacheDriver

ApacheDriver extends the ApacheDriver and the PhpWebAppSshDriver but it is an empty interface, so it does not define any ConfigKey, sensor or method.

3.2.4.10 ApacheSshDriver

The ApacheSshDriver class contains the management based on Ssh of the Apache Server installed. Please note, that currently the management is supported for a linux based system (Debian), in successive versions the operations will be deployed to support all deployment environments. This class defines several methods to implement at @Override their supers.

- **getEntity()**: returns entity that represents the Apache Web Server, in this case ApacheServe.
- **getPortMap()**: returns enable protocols and their listened ports.
- **getDeploySubDir()**: returns the folder path where the PHP application can be deployed.
- **getRunDir()**: determines the folder where the server is installed.
- **install()**: installs the server in the host. This method uses the super method which installs the common software such as PHP and git.
- **customize()**: this method configures the Apache server to allow Brooklyn to be managed.
- **isRunning()**: returns true when Apache is running and false otherwise.
- **deploy_git_resource()** and **deploy_tarball_git()**: deploy an application from a git repo or a tarball URL (these method have been aforementioned before).

Lightweight PHP support: Brooklyn supports a simpler and lightweight approach to support entity that are not supported by Brooklyn catalog, yet: VanillaSoftwareProcess + YAML.

3.3 Monitor mechanisms

Here we describe the monitoring mechanisms available in Brooklyn that allows SeaClouds to define new data retrieving mechanisms as well as policies that will act over Brooklyn sensors triggering some minor reconfiguration on the managed systems.

3.3.1 Data feeds

The usual strategy when reading information and value changes on sensors is to perform a periodic query or request over certain service or method. Different kinds of data feeds are provided by Brooklyn and they implement the core functionality that reads the sensors from the deployed modules, as shown in Figure 6.

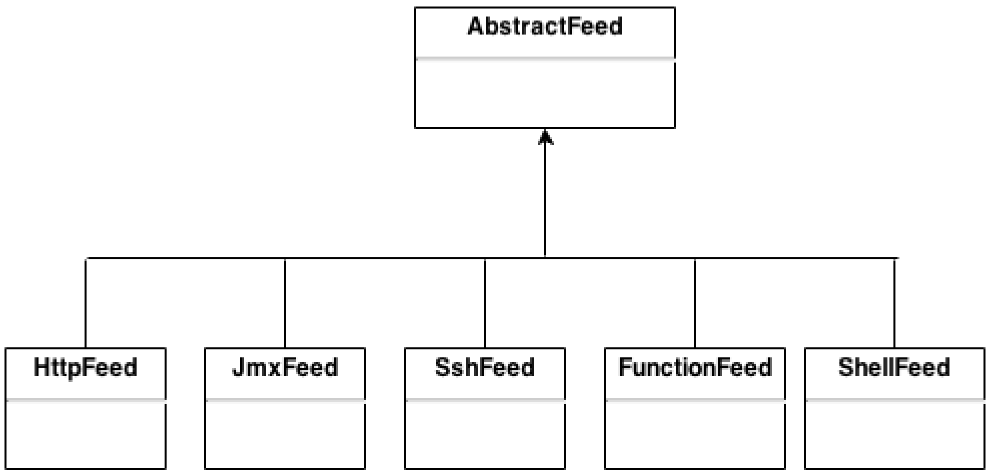


Figure 6. Data feeds hierarchy

The existing feeds are structured according to the Figure 6:

- Abstract Feed: It captures common fields and processes for sensor feeds. These generally poll or subscribe to get sensor values for an entity. They make it easy to poll over http, jmx, etc, and they provide the basic lifecycle methods for starting/stopping feeds.
- HttpFeed: Provides a feed of attribute values, by polling over http.
- JmxFeed: Provides a feed of attribute values, by polling or subscribing over jmx.
- SshFeed: Provides a feed of attribute values, by polling over ssh.
- FunctionFeed: Provides a feed of attribute values, by periodically invoking functions.
- ShellFeed: Provides a feed of attribute values, by executing shell commands.

3.3.1.1 *User-defined sensors*

Application-level metrics can be defined in some part of the process of deploying an application, creating a connection to the deployed sensor and retrieving the desired metrics. We have developed these monitoring mechanisms for a deployer unit such as Brooklyn, in which sensors can be defined on a YAML document.

We made use of EntityInitializer interface to implement a dynamic way of defining sensors that will retrieve information from a remote URI, typically from a monitoring service that the user provided.

3.3.1.2 *EntityInitializer*

Instances of EntityInitializer supply logic which can be used to initialize entities. These can be added to an EntitySpec programmatically, or declared as part of YAML recipes in a brooklyn.initializers section. In the case of the latter, implementing classes should define a no-arg constructor or a Map constructor so that YAML parameters can be supplied.

It simply contains the definition for the method “apply” which will be executed over an EntityLocal and perform the needed adjustments.

3.3.1.3 AddSensor

This class extends EntityInitializer (see Figure 7) and sets the main common attributes and methods on adding a new sensor to a given entity, such as:

- name: Name which will identify the sensor to be created.
- period: Periodic time of the poll requests
- targetType: This attribute is necessary to convert the monitoring data into a certain sensor type. Type needs to be specified with a full Java classname but many of the most used types are simplified to be more convenient (String, Integer, Double, or Float, …).

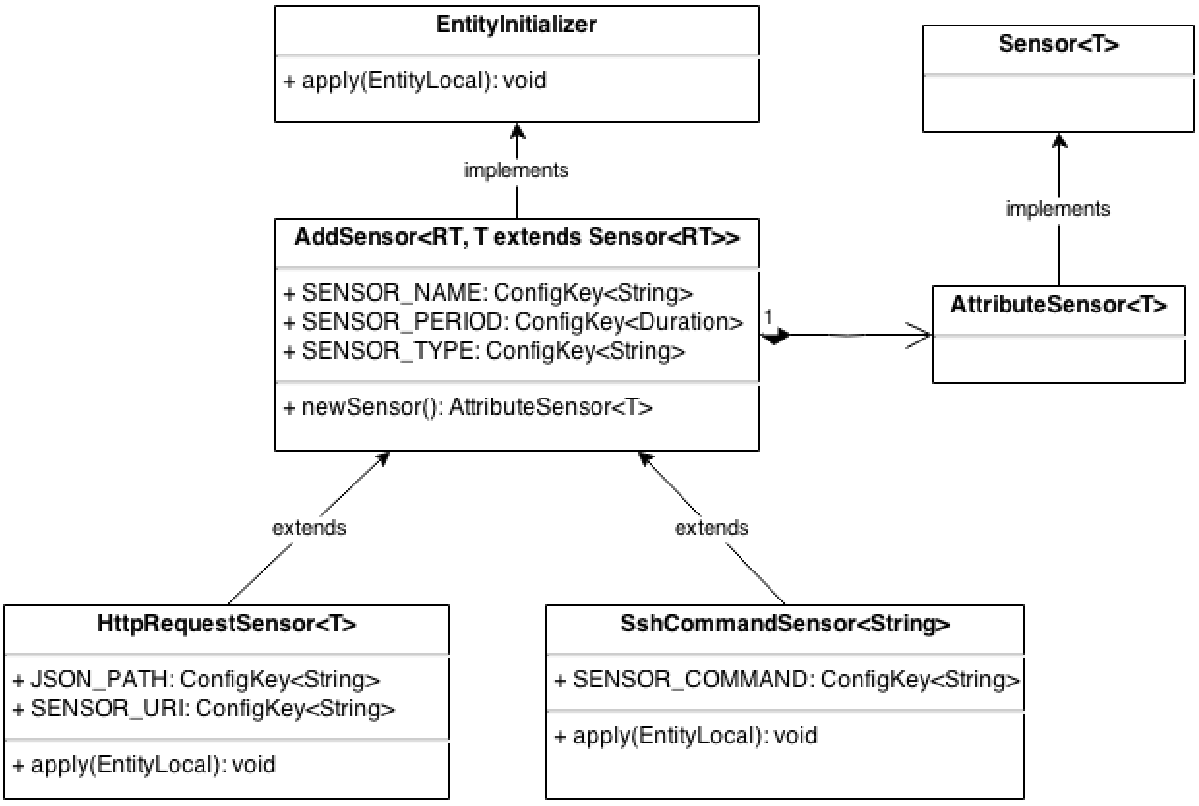

EntityInitializer
+ apply(EntityLocal): void
Sensor<T>
implements
implements
AddSensor<RT, T extends Sensor<RT>>
+ SENSOR_NAME: ConfigKey<String>
+ SENSOR_PERIOD: ConfigKey<Duration>
+ SENSOR_TYPE: ConfigKey<String>
+ newSensor(): AttributeSensor<T>
1
AttributeSensor<T>
extends
extends
HttpRequestSensor<T>
+ JSON_PATH: ConfigKey<String>
+ SENSOR_URI: ConfigKey<String>
+ apply(EntityLocal): void
SshCommandSensor<String>
+ SENSOR_COMMAND: ConfigKey<String>
+ apply(EntityLocal): void

Figure 7. AddSensor class diagram

3.3.1.4 HttpRequestSensor

This class creates a HttpFeed that will periodically generate an HttpRequest to a given URI, and dumping this information into a new sensor.

One of the requirements of using this sensor is that the HttpResponse from the monitoring URI needs to be a JSON document. JSON format makes it simpler to read more than one sensors on each request. In this way, we make use of json-path [7], a library that gives full path access to every attribute defined on a JSON document at any level.

To use this sensor initializer the following configuration keys need to be specified:

- uri: monitoring service URI that will generate the JSON document with sensors data.
- jsonPath: full path to the sensor value. (Examples on how to use this path notation

can be found at http://goessner.net/articles/JsonPath/).

3.3.2 Polices

Policies perform the active management enabled by Brooklyn. They can subscribe to entity sensors and be triggered by them or they can run periodically. Policies can add subscriptions to sensors on any entity. Normally a policy will subscribe to its related entity, to the child entities, and/or those entities which are members.

When a policy runs it can:

- perform calculations,
- look up other values,
- invoke effectors (management policies) or,
- cause the entity associated with the policy to emit sensor values (enricher policies).

Entities can have zero or more Policy instances attached to them. Policies are highly reusable as their inputs, thresholds and targets are customizable.

3.3.2.1 Mangement Policies

- Resizer Policy
- Increases or decreases the size of a Resizable entity based on an aggregate sensor value, the current size of the entity, and customized high/low watermarks.
- A Resizer policy can take any sensor as a metric, have its watermarks tuned live, and target any resizable entity - be it an application server managing how many instances it handles, or a tier managing global capacity.
- e.g. if the average request per second across a cluster of Tomcat servers goes over the high watermark, it will resize the cluster to bring the average back to within the watermarks.

3.3.2.2 EnricherPolicies

- **Delta:** converts absolute sensor values into a delta.
- **Time-weighted Delta**: converts absolute sensor values into a delta/second.
- **Rolling Mean:** converts the last N sensor values into a mean.
- **Rolling Time-window Mean:** converts the last N seconds of sensor values into a weighted mean.
- **Custom Aggregating**: aggregates multiple sensor values (usually across a tier, esp. a cluster) and performs a supplied aggregation method to them to return an aggregate figure, e.g. sum, mean, median, etc.

3.4 Dashboard code description

In this release we have presented a basic frontend using HTML5 and JavaScript libraries. In addition, in this version we have not implementing the SeaClouds unified API yet (it is under implementation), instead, to ensure the correct functionality of the Deployer connected with the Monitor, we base our initial design on existing API's like Brooklyn to gather the needed information.

3.5 SLA Service code description

The SLA Service is a WS-Agreement compliant module. The Core component of the SLA Service is a RESTful web service developed in Java. It uses mysql as database. The code is not yet publicly available at the time of the writing of the deliverable.

3.5.1 Repository

The repository entities are JPA entities that mimic the WS-Agreements entities, i.e. templates and agreements. The repository also provides Data Access Objects (DAOs) to isolate the use of the entities from the underlying technology.

A first version of the class diagram is shown in Figure 8.

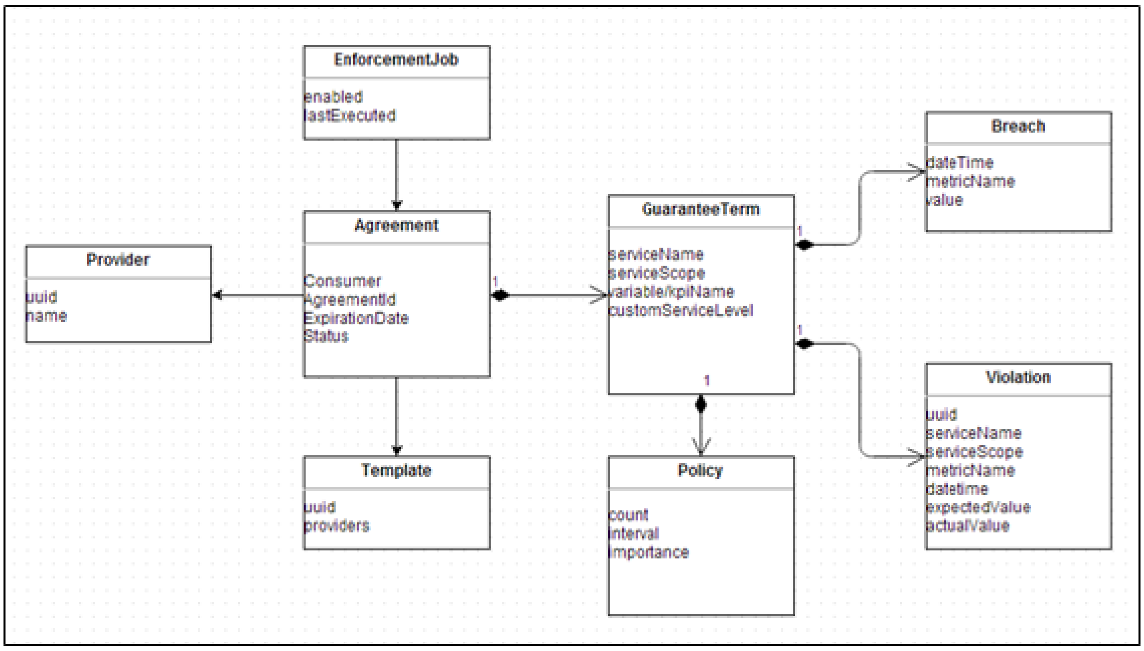


Figure 8. SLA Repository Architecture

The diagram shows the main entities in the specification and the way they are related:

- A Provider offers a service: a software service, hardware resources, etc.
- The service is described by ServiceDescriptionTerms with a Domain Specific Language. The ServiceDescriptionTerms are intended to define a service that has to be provisioned. This sla module needs external provision.
- The provided service is represented by a Template, and the Template can be used to generate an Agreement.
- An agreement is a "document" that associates a Service and a Consumer. When the relation is in negotiation-phase, it's called an AgreementOffer. Once the agreement is accepted, it's called a Agreement.
- A Template and an Agreement can describe some restrictions to be fulfilled by the

Consumer or by the Provider. The restrictions must be defined in a Domain-Specific Language

- A violation of any restriction generates a Violation.

3.5.2 Assessment

The assessment of existing agreements is a process periodically executed in a thread. This process walks over the agreements and checks if the agreements needs to be enforced. If so, a task is created is pushed to a Thread Pool Executor, to be executed at a later time. The tasks are pulled from the pool, and then executed. The needed metrics are retrieved from the source (in this draft implementation, from Brooklyn sensors), and the constraints are evaluated. This QoB evaluation generates penalties that will be notified to the Application Administrator. The sequence diagram in Figure 9 summarizes the process.

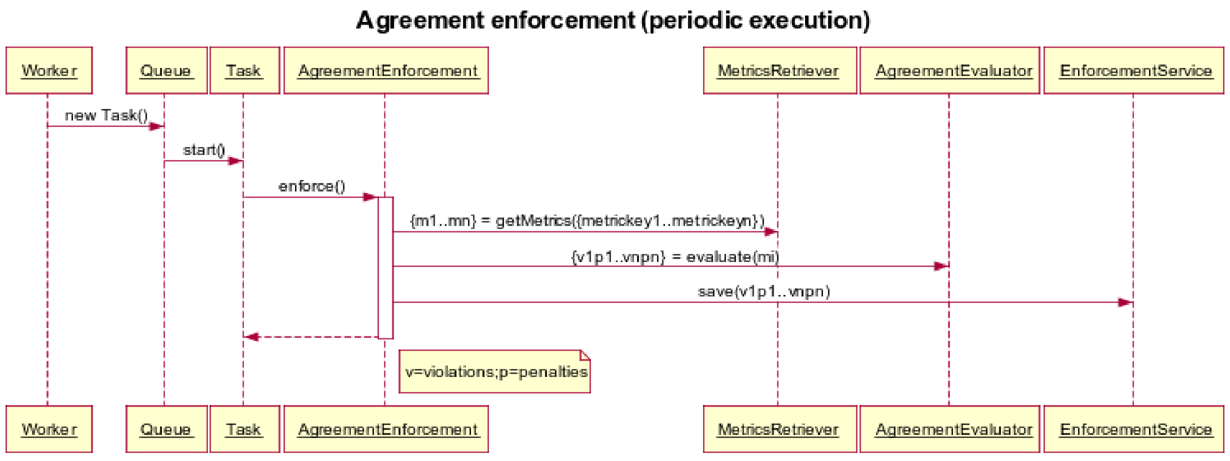


Figure 9. Assessment Behaviour

3.5.3 SLA Manager

The SLA Manager is the module acting as a RESTful web service. The REST endpoints are JAX-RS annotated classes, being jersey the JAX-RS implementation.

There is a resource endpoint per each interesting entity:

- Providers
- Templates
- Agreements
- Enforcement jobs
- Violations

Additionally, another endpoint is exposed to receive metrics in case of having Monitoring Managers that push metrics to interested observers.

4. Installation and Configuration

In this section, we present the pre-requisites and the how to instal and configure the corresponding pieces to execute the first prototype of the SeaClouds software platform.

4.1 Deployer and Monitor (and Dashboard)

Here we detail the steps required to install and configure the deployment and monitoring of a cloud application using the first version of the SeaClouds platform.

4.1.1 Brooklyn Installation

Currently, the Brooklyn version that contains the features described in this document is stored in the following repository (in the branch phpAppDBSupport):

https://github.com/kiuby88/incubator-brooklyn

The repo can be cloned in any repository using git and the necessary url:

https://github.com/kiuby88/incubator-brooklyn.git

The following command can be used:

```
$ git clone https://github.com/kiuby88/incubator-brooklyn.git
```

brooklyn$ will be the Brooklyn repository.

Before that Brooklyn will be compiled, it is necessary select the branch. So, you can use the command:

```
brooklyn$ git branch phpAppDBSupport
```

Next, brooklyn could be compiled using mvn command:

```
brooklyn$ mvn clean install
```

If you find a test error or Rat error, you are able to use the next command to sky it:

```
brooklyn$ mvn clean install -DskipTest=true -Drat.ignoreErrors=true
```

To launch Brooklyn, let's setup some paths for easy commands.

```
$ cd brooklyn$
$ BROOKLYN_DIR="$(pwd)"
$ export PATH=$PATH:$BROOKLYN_DIR/bin/
```

A quick test drive by launching Brooklyn can be done using:

```
$ brooklyn launch
```

Brooklyn will output the address of the management interface:

- INFO Starting brooklyn web-console on loopback interface because no security config is set.
- INFO Started Brooklyn console at http://127.0.0.1:8081/
- But before we really use Brooklyn, we need to setup some Locations.

- Stop Brooklyn with ctrl-c.

4.1.2 Pre-requisites

This section lists the general software prerequisites to install and run the components belonging to the SeaClouds platform.

- Java (JDK 1.6)
- Apache Maven: (version Maven 3.x)
- GIT

Now it is time to tell to the system, that where it can find Maven, Ant and Tomcat. To do this, the environment variables (and PATHs) will be set.

4.1.3 Brooklyn Configuration

It is necessary perform a configuration of Brooklyn. Brooklyn deploys applications to Locations. Locations can be clouds, machines with fixed IPs or localhost (for testing).

Brooklyn loads Location configuration from `~/.brooklyn/brooklyn.properties`.

First, create a `.brooklyn` folder in your home directory and download the template brooklyn.properties to that folder:

$ mkdir ~/.brooklyn

$ cd ~/.brooklyn

$ wget /use/guide/quickstart/brooklyn.properties

Next, open brooklyn.properties in a text editor and add your cloud credentials. If you would rather test Brooklyn on localhost, follow these instructions to ensure that your Brooklyn can access your machine.

Then, restart Brooklyn:

$ brooklyn launch

Next, you can download the SeaClouds Dashboard proposal, available in the SeaClouds GitHub:

$ git clone https://github.com/SeaCloudsEU/demo-dashboard.git

After downloading it, you only need to change the endpoint where the dashboard will search for Brooklyn. This setup is inside the file “config.js” in “js”/ folder.

Please notice that if you run the dashboard from local filesystem (by double clicking index.html) you will need to disable “Cross-Origin Resource Sharing” security protection from your browser. We encourage executing the Dashboard in a web server.

4.2 SLA Service Installation

In this section, we describe the procedure to be performed to install the SLA service of the SeaClouds platform.

4.2.1 Requirements

The requirements to install a working copy of the Core component of the SLA Service are:

- Oracle JDK >=1.6
- Database to install the database schema for the service: Mysql>=5.0
- Maven >= 3.0

4.2.2 Creating the mysql database

From mysql command tool, create a database (with a user with sufficient privileges, as root):

```
$ mysql -p -u root
```

```
mysql> CREATE DATABASE sla;
```

Create a user:

```
mysql> CREATE USER slauser@localhost IDENTIFIED BY '_sla_';
```

```
mysql> GRANT ALL PRIVILEGES ON sla.* TO slauser@localhost; -- optional WITH GRANT OPTION;
```

From command prompt, create needed tables:

```
$ mvn test exec:java -f sla-repository/pom.xml
```

Another option to create the database is execute a sql file from the project root directory:

```
$ bin/restoreDatabase.sh
```

The names used here are the default values of the sla core. See "Configuration" and "Running" sections to know how to change these values.

4.2.3 Configuration

A *configuration.properties.sample* that is placed in the parent directory has to be copied to *configuration.properties*.

Several parameters can be configured through this file.

1. db.* allows to configure the database username, password and name in case it has been changed from the proposed one in the section "Creating the mysql database". It can be selected if queries from hibernate must be shown or not. These parameters can be overriden at deployment time through the use of environment variables (see section Running),
2. log.* allows to configure the log files to be generated and the level of information,
3. enforcement.* several parameters from the enforcement can be customized,
4. service.basicsecurity.* basic security is enabled These parameters can be used to set the user name and password to access to the rest services.

4.2.4 Compilation

To compile the package:

```
$ mvn install
```

If you want to skip tests:

```
$ mvn install -Dmaven.test.skip=true
```

The result of the command is a war package in *sla-service/target* directory.

4.2.5 Running

Now, you can deploy the war to an application server. If you are using Tomcat, you must copy the war to $TOMCAT/webapps.

Alternatively, you can run an embedded tomcat:

```
$ bin/runserver.sh
```

that is just a shortcut for:

```
$ mvn tomcat:run -f sla-service/pom.xml
```

Some configuration parameters can be overridden using environment variables or jdk variables. The list of overridable parameters is:

DB_DRIVER; default value is com.mysql.jdbc.Driver

DB_URL; default value is jdbc:mysql://${db.host}:${db.port}/${db.name}

DB_USERNAME; default value is ${db.username}

DB_PASSWORD; default value is ${db.password}

DB_SHOWSQL; default value is ${db.showSQL}

For example, to use a different database configuration:

```
$ export DB_URL=http://localhost:8080/sla
$ export DB_USERNAME=sla
$ export DB_PASSWORD=<secret>
$ bin/runserver.sh
```

4.2.6 Testing

Check that everything is working:

```
$ curl http://localhost:8080/sla-service/providers
```

The actual address depends on the application server configuration. The embedded tomcat uses http://localhost:8080/sla-service/ as service root url.

4.3 Demo Execution

Next, we show the CAMP YAML blue print, which we have generated and used to deploy the

Nuro early application:

```
name: PHP NuroCaseStudy
services:
- serviceType: brooklyn.entity.webapp.apache.ApacheServer
 name: Apache Server
 id: apache
 location: localhost
 brooklyn.config:
   http_port: 80
   app_git_repo_url: <NURO_CASE_STUDY_GIT_REPO>
   db_connection_file_config: config/config.php
   db_connection_config_params:
     g_DatabaseHost: $brooklyn:formatString("%s", component("db").attributeWhenReady("
     host.address"))
     g_DatabasePort: $brooklyn:component("db").attributeWhenReady("mysql.port")
     g_DatabasePassword: $brooklyn:component("db").attributeWhenReady("mysql.password")
     g_DatabaseUser: root
 brooklyn.initializers:
 - type: brooklyn.entity.software.http.HttpRequestSensor
   brooklyn.config:
     name: nuro.analytics_time
     #uri: $brooklyn:formatString("%s/sensor.php",component("apache")
     .attributeWhenReady("host.address"))
     uri: http://<%brooklyn-machine-IP%>/nurocasestudyphp5-5/sensor.php
     jsonPath: $.database1.analytics_time
     targetType: double

- serviceType: brooklyn.entity.database.mysql.MySqlNode
 id: db
 name: MySQL Node
 location: localhost
 brooklyn.config:
   datastore.creation.script.url: <NURO_CASE_STUDY_SQL>
```

- Notes:

Currently, the HTTP sensor needs to define the IP where the application containing the sensor will be deployed. For example, if the application has to be deployed in localhost, <%brooklyn-machine-IP%> will be substituted by `localhost` or 127.0.0.1.

Since the code of the Nuro Case is confidential, currently, the values of <NURO_CASE_STUDY_GIT_REPO> and <NURO_CASE_STUDY_SQL>, related to the Git repo and the Database respectively, can be found in the Deliverable 6.3.1 [8].

In order to illustrate how the deployment and monitoring of the Nuro early Case Study, a video have been recorded:

https://drive.google.com/file/d/0Bw9KJPN8k2glaXlCVGxZSmw0dUk/edit?usp=sharing

5. Conclusion

This deliverable accompanies the first prototype of the SeaClouds platform. The software prototype can be downloaded from https://github.com/SeaCloudsEU/SeaCloudsPlatform.

Along the document, we have described the main services and functionalities currently implemented in the initial version of the software platform, containing mainly the deployer, monitoring, and SLA components, related to the run-time environment of SeaClouds.

As regards the design-time part, some research efforts are being performed proposing the algorithms needed to the discovery process and the planner component, including the matchmaking and the optimizer processes. In particular, currently, a simple matchmaking mechanism have been implemented, although at the moment of writing this document some efforts are required to detail the implementation of this mechanism, so we decided to detail it in the next deliverable related to the integration of components in the software platform.

6. References

[1] Deliverable D3.1 - Discovery design and orchestration functionalities: first specification.

[2] Deliverable D4.1 - Definition of the multi-deployment and monitoring strategies.

[3] Deliverable D4.2 - Cloud Application Programming Interface.

[4] jClouds - Project page: https://jclouds.apache.org/

[5] Brooklyn Official page:

https://brooklyn.incubator.apache.org/v/0.7.0-M1/use/guide/quickstart/index.html

[6] SoftwareProcess Interface reference: https://brooklyn.incubator.apache.org/v/0.7.0-M1/use/api/index.html

[7] Json-path - Java JsonPath implementation - Google Project Hosting:
https://code.google.com/p/json-path/

[8] Deliverable D6.3.1 - Case Studies Preliminary implementation.