SLAs Empowering a Dependable Service Economy

**Project no.** FP7- 216556  
**Instrument:** Integrated Project (IP)  
**Objective ICT-2007.1.2:** Service and Software Architectures, Infrastructures and Engineering

## Deliverable D.A1b

### Integrated Framework (full lifecycle)

**Keywords:**  
Service Level Agreement, Service-Oriented Infrastructure, Software

**Due date of deliverable:** 31st July 2010  
**Actual submission to EC date:** 10th September 2010

**Start date of project:** 1st June 2008  
**Duration:** 38 months

**Lead contractor for this deliverable:** FBK  
**Revision:** V.1.1 (6th September 2010)

---

<table>
<thead>
<tr>
<th>Dissemination level</th>
<th>PU</th>
<th>Public</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Project co-funded by the European Commission within the Seventh Framework Programme (2007-2013)
## Document Status

<table>
<thead>
<tr>
<th>Deliverable Lead</th>
<th>Gabriele Zacco, FBK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reviewer 1</td>
<td>Sam Guinea, PMI</td>
</tr>
<tr>
<td>Reviewer 2</td>
<td>Giovanni Falcone, FZI</td>
</tr>
<tr>
<td>PMT Reviewer</td>
<td>Wolfgang Theilmann</td>
</tr>
<tr>
<td>Complete version submitted to reviewers</td>
<td>July 7, 2010</td>
</tr>
<tr>
<td>Comments of reviewer 1 received</td>
<td>July 20, 2010</td>
</tr>
<tr>
<td>Comments of reviewer 2 received</td>
<td>July 14, 2010</td>
</tr>
<tr>
<td>Revised deliverable submitted to PMT</td>
<td>August 5, 2010</td>
</tr>
<tr>
<td>PMT Approval</td>
<td>August 10, 2010</td>
</tr>
</tbody>
</table>

## Contributors

<table>
<thead>
<tr>
<th>Partner</th>
<th>Contributors</th>
</tr>
</thead>
<tbody>
<tr>
<td>FBK</td>
<td>Gabriele Zacco, Nawaz Khurshid</td>
</tr>
<tr>
<td>UDO</td>
<td>Miguel Angel Rojas Gonzales</td>
</tr>
<tr>
<td>FZI</td>
<td>Martin Kuester</td>
</tr>
<tr>
<td>CITY</td>
<td>Miha Vuk</td>
</tr>
<tr>
<td>INTEL</td>
<td>John Kennedy, Andrew Edmonds, Victor Bayon</td>
</tr>
<tr>
<td>TID</td>
<td>Juan Lambea</td>
</tr>
<tr>
<td>SAP</td>
<td>Tariq Ellahi</td>
</tr>
</tbody>
</table>

## Notices

The information in this document is provided "as is", and no guarantee or warranty is given that the information is fit for any particular purpose. The above referenced consortium members shall have no liability for damages of any kind including without limitation direct, special, indirect, or consequential damages that may result from the use of these materials subject to any liability which is mandatory due to applicable law. Copyright 2009 by the SLA@SOI consortium.

* Other names and brands may be claimed as the property of others.

This work is licensed under a [Creative Commons Attribution 3.0 License](https://creativecommons.org/licenses/by/3.0/).
<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Author</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>10/06/10</td>
<td>Gabriele Zacco</td>
<td>Skeleton, introductions and other contents</td>
</tr>
<tr>
<td>0.2</td>
<td>16/06/10</td>
<td>Miguel Angel Rojas Gonzales</td>
<td>Generic SLA Manager adoption section</td>
</tr>
<tr>
<td>0.3</td>
<td>17/06/10</td>
<td>Gabriele Zacco</td>
<td>Technical environment section</td>
</tr>
<tr>
<td>0.4</td>
<td>18/06/10</td>
<td>Martin Kuester</td>
<td>Service Evaluation adoption section</td>
</tr>
<tr>
<td>0.5</td>
<td>22/06/10</td>
<td>Miha Vuk</td>
<td>Monitoring adoption section</td>
</tr>
<tr>
<td>0.6</td>
<td>23/06/10</td>
<td>John Kennedy, Andrew Edmonds, Victor Bayon</td>
<td>Infrastructure Service Manager adoption section</td>
</tr>
<tr>
<td>0.7</td>
<td>23/06/10</td>
<td>Juan Lambea</td>
<td>Section 5.4.2, Business adoption updated</td>
</tr>
<tr>
<td>0.8</td>
<td>25/06/10</td>
<td>Gabriele Zacco, Nawaz Khurshid</td>
<td>Refinement of several sections, Integration section</td>
</tr>
<tr>
<td>0.9</td>
<td>03/07/10</td>
<td>Tariq Ellahi</td>
<td>Section 5.4.3</td>
</tr>
<tr>
<td>0.10</td>
<td>03/08/10</td>
<td>Gabriele Zacco</td>
<td>Filled in framework usage from developer and user points of view; integrated reviewer’s feedback;</td>
</tr>
<tr>
<td>1.0</td>
<td>05/08/10</td>
<td>Gabriele Zacco</td>
<td>First version</td>
</tr>
<tr>
<td>1.1</td>
<td>06/09/10</td>
<td>Gabriele Zacco</td>
<td>Implemented PMT feedback; added integration test, main achievements and figures</td>
</tr>
</tbody>
</table>
Executive Summary

This document describes the implementation of the SLA@SOI framework that has been developed during the first two years of the European research project SLA@SOI. Differently from the formal deliverable D.A1a which describes the framework from an architectural point of view, this deliverable presents the actual realization of the architecture. Together with the deliverable D.A1a, this document serves for reporting the progress of work package A1 at project month 26.

This document presents the software-related artefacts, that is mainly code but not only, that have been developed within the project and all the relevant achieved results. The main facts about the development may be summed up as follows:

- realization of a real framework approach that can be flexibly integrated, configured, and extended in different environments – based on the industry standard OSGi;
- guidelines for an effective adoption of the framework;
- significant improvement of the achieved code quality through rigorous testing including integration tests (3 main scenarios covered) and unit tests (more than 250 JUnit tests);
- a complete documentation of in terms of inline comments and JavaDocs (the almost 90,000 Java source code lines are documented by about 25,000 JavaDocs lines and about 15,000 lines more of Java comments);

This deliverable also provides an overview of the technical environment that has produced such results. It also describes the development and integration approaches that have been followed for delivering the software. Finally it presents both a user’s and a developer’s perspective about the usage of the SLA@SOI framework.
Table of Contents

1 Introduction ........................................................................................................... 8

2 Technical Environment ......................................................................................... 8
  2.1 Tools .................................................................................................................. 9
  2.1.1 Wiki .............................................................................................................. 9
  2.1.2 Subversion .................................................................................................... 10
  2.1.3 Trac ............................................................................................................... 10
  2.1.4 Maven and Artifactory .................................................................................. 11
  2.1.5 Continuum ................................................................................................... 12
  2.2 Organizational Procedures ............................................................................... 13
  2.3 Quality Management ....................................................................................... 14
    2.3.1 Unit Testing ................................................................................................. 14
    2.3.2 Coding Practices ........................................................................................ 14
    2.3.3 Coding Style ................................................................................................ 15
    2.3.4 JavaDocs ..................................................................................................... 15
    2.3.5 Maven Generated Website ......................................................................... 16

3 Technical Integration ............................................................................................. 16
  3.1 Approach ........................................................................................................... 16
  3.2 Design-time ..................................................................................................... 17
  3.3 Run-time .......................................................................................................... 18
  3.4 Integration Testing ............................................................................................ 19

4 Framework Usage .................................................................................................. 22
  4.1 Requirements ................................................................................................... 22
  4.2 Download ......................................................................................................... 23
  4.3 Installation ....................................................................................................... 23
  4.4 Configuration ................................................................................................... 24
  4.5 Run .................................................................................................................... 24

5 Framework Development ...................................................................................... 25
  5.1 Structure of the Source Code .......................................................................... 25
  5.2 Platform Adoption ............................................................................................ 26
  5.2.1 Generic SLA Manager Adoption .................................................................. 26
  5.2.2 Business Components Adoption .................................................................. 27
  5.2.3 Software Service Manager Adoption ......................................................... 29
  5.2.4 Infrastructure Service Manager Adoption .................................................. 31
  5.2.5 Service Evaluation Adoption ....................................................................... 34
  5.2.6 Monitoring Adoption .................................................................................. 35
  5.2.7 Integration of a new Component into the Platform ...................................... 35
  5.3 Build Instructions ............................................................................................. 41
    5.3.1 Requirements .............................................................................................. 41
    5.3.2 Download ................................................................................................... 42
    5.3.3 Configuration ............................................................................................... 42
    5.3.4 Build ........................................................................................................... 42
    5.3.5 Run .............................................................................................................. 42

6 Conclusions ............................................................................................................ 44

7 References ............................................................................................................. 44

Appendix A: Glossary ............................................................................................... 47
Appendix B: Abbreviations ....................................................................................... 50
# Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>SLA@SOI platform source code hierarchy</td>
<td>10</td>
</tr>
<tr>
<td>Figure 2</td>
<td>SLA@SOI Developer TRAC</td>
<td>11</td>
</tr>
<tr>
<td>Figure 3</td>
<td>SLA@SOI Artifactory Web console</td>
<td>12</td>
</tr>
<tr>
<td>Figure 4</td>
<td>SLA@SOI Continuum Web console</td>
<td>13</td>
</tr>
<tr>
<td>Figure 5</td>
<td>OSGi stack</td>
<td>17</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Negotiation scenario</td>
<td>20</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Provisioning scenario</td>
<td>21</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Run-time scenario</td>
<td>22</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Binary package structure</td>
<td>23</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Code for creating a SLAM's context</td>
<td>27</td>
</tr>
<tr>
<td>Figure 11</td>
<td>Code for injecting POC/PAC into SLAM</td>
<td>27</td>
</tr>
<tr>
<td>Figure 12</td>
<td>Code for injecting Context into Discovery and PostSale</td>
<td>28</td>
</tr>
<tr>
<td>Figure 13</td>
<td>Code for injecting POC/PAC into B-SLAM</td>
<td>28</td>
</tr>
<tr>
<td>Figure 14</td>
<td>Code for injecting Business Authorization</td>
<td>28</td>
</tr>
<tr>
<td>Figure 15</td>
<td>Code for injecting Customer Relations and Product Discovery</td>
<td>28</td>
</tr>
<tr>
<td>Figure 16</td>
<td>Code for representing a VM</td>
<td>32</td>
</tr>
<tr>
<td>Figure 17</td>
<td>Code for ProvisionRequest and Compute operations</td>
<td>33</td>
</tr>
<tr>
<td>Figure 18</td>
<td>Code for provisioning a VM</td>
<td>33</td>
</tr>
<tr>
<td>Figure 19</td>
<td>Code for getting provisioning details</td>
<td>34</td>
</tr>
<tr>
<td>Figure 20</td>
<td>Code for stopping the VM</td>
<td>34</td>
</tr>
<tr>
<td>Figure 21</td>
<td>Sample Spring application context file</td>
<td>36</td>
</tr>
<tr>
<td>Figure 22</td>
<td>Sample OSGi application context file</td>
<td>37</td>
</tr>
<tr>
<td>Figure 23</td>
<td>OSGi configuration files hierarchy</td>
<td>37</td>
</tr>
<tr>
<td>Figure 24</td>
<td>OSGi configuration within Maven POM</td>
<td>37</td>
</tr>
<tr>
<td>Figure 25</td>
<td>OSGi dependency injection within Maven POM</td>
<td>38</td>
</tr>
<tr>
<td>Figure 26</td>
<td>Configuration of the BeanPostProcess for OSGi service injection</td>
<td>38</td>
</tr>
<tr>
<td>Figure 27</td>
<td>POM configuration when using Service Reference Annotation</td>
<td>39</td>
</tr>
<tr>
<td>Figure 28</td>
<td>OSGi service injection when using Service Reference Annotation</td>
<td>39</td>
</tr>
<tr>
<td>Figure 29</td>
<td>OSGi service injection when using Activator Bundle Context</td>
<td>40</td>
</tr>
<tr>
<td>Figure 30</td>
<td>POM configuration when using Activator Bundle Context</td>
<td>41</td>
</tr>
</tbody>
</table>
1 Introduction

The deliverable D.A1b is mainly a software deliverable that consists of the actual implementation of the SLA@SOI framework. It complements deliverable D.A1a [3], which describes the SLA@SOI platform from an architectural point of view. The two deliverables provide a report of the progress of work package A1 up until project month 26.

The goal of the present document is to accompany the delivery of the software artefacts produced during the second year of the project by describing such software artefacts and all other related artefacts. Furthermore, we also aim at providing a description of the technical background in which such artefacts have been produced. Finally, we intend to give here both a user’s and a developer’s perspective about the usage of the SLA@SOI platform, by describing how it can be used for running sample test case scenarios and how its components can be used and extended in order to apply it to other domains.

The main shift with respect to the previous version of the deliverable D.A1b, that was released at M12 [2], is the description of the platform from a more developer-oriented perspective in order to allow for its adoption and exploitation; hence, the presentation is not only restricted to a user’s point of view. Moreover, there are other novelties with respect to the previous version of the deliverable. On the one side, the actual software deliverable, in terms of source and binary code, represents a complete refactoring of what was submitted after the first year of the project. This is mainly due to the revision of the framework architecture described in [3]. On the other side, some portions of the current document extend what was described in the first versions both of the deliverable D.A1a and of the deliverable D.A1b, whilst some portions of it represent a totally new perspective on the integrated platform.

More specifically, section 2 provides an update of the technical environment that has been used to develop the SLA@SOI platform. Section 3 describes the integration approach used to assemble the SLA-management framework. Section 4 describes the platform from an end-user perspective showing a possible usage of the platform based on the execution of sample integration tests. Section 5 addresses the topic of the possible adoption of the platform that is oriented towards its usage in different, and totally generic, scenarios. Section 6 draws some conclusions and section 7 contains the all references mentioned in the document. The last sections are devoted to the standard Glossary and Abbreviations lists.

2 Technical Environment

A detailed description of the technical environment used for the first year of the project is available in the deliverable D.A1a at milestone M12 [2]. The technical environment adopted for year two consists basically of an extension and a refinement of the former one.

More precisely, having in mind the SLA@SOI project’s goal of developing prototyped solutions that pursue a high quality level both for the code and the documentation, the approach to the technical environment has been directed in the following three directions. First of all the usage of the Wiki, SVN, and Maven tools plus the usage coding practices and coding style has been maintained and improved. Then, the adoption of Trac and Continuum has reached a maturity level
unknown in the first year of the project. Finally the generation of unit testing and Java docs has been enhanced.

The following sections focus on these aspects providing in particular the details about the novel aspects introduced in this second year of the project. Section 2.1 describes the instruments that have been used in order to support the development process; section 2.2 provides some details about the establishment of a common development process; section 2.3 deals with quality related topics.

2.1 Tools

The technical environment consists of a set of instruments that have been devised with the goal of supporting the harmonization and integration of the different perspectives of the stakeholders involved in the project. The main driver of the technical environment set-up is to facilitate the management and in particular the tracking both of the architectural specification of the platform and of the day-by-day development and integration activities carried on by the distributed development teams.

2.1.1 Wiki

The usage of the wiki [4] during the second year has been confirmed and consolidated with respect to year one. During the re-design of the architecture it has been intensively used for organizing and coordinating all the contributions to the definition and documentation of the new platform architecture. During the implementation phases, the wiki has had the role of container for the specification of the implementation details about the modules and for the common guidelines about the implementation itself. Finally, for the integration activities it has been used for storing the description of the guidelines used for integrating the framework. Moreover, the wiki has been the repository for the description of the technical environment itself and as a repository for documenting the discussions and the minutes of the meetings.

More in general, the wiki during the second year of the project has consolidated its role about the establishment of a more commonly understood and coordinated approach to the development and integration of the framework.

2.1.2 Subversion

During the second year of the project, the Subversion repository [5] has been re-organized. In particular, the portion of the repository [6] dedicated to host the SLA@SOI platform that has been developed during the second year of the project, has been separated from the one used in the first year and has been totally re-organized. As a matter of fact, due to the new architectural specifications a brand new repository space where to store the code was needed, even though there has been an obvious amount of code reuse with respect to the year one development. This new organization of the source and related folders has guaranteed a cleaner environment in which to build the new platform.

The organization of the SVN has strictly followed the prescription of the architectural specifications. In particular the hierarchy of the folder in the SVN – visible in Figure 1 - has been designed to reflect the structure of the platform and the documentation of it that has been stored in the wiki and in the relevant documents. For all the top-level building blocks of the architecture, and for the second-level sub-components as well, we have a corresponding folder within the SVN folder hierarchy. The advantage of using this approach has been to increase the traceability of the code with respect to its specification and documentation.
2.1.3 Trac

The usage of the Trac bug-tracking and software management system [8] reached a mature status towards the end of the Y2 activities, where it was configured for managing the bugs and the issues related to the development and integration of the platform.

A second instance of the Trac system [9] has been setup and configured on top of the SVN repository – the first being the one used for tracking the requirements of the platform [8]. Milestones, related both to the development and integration of the platform and its components have been set on the project timeline in order to better monitor the status of the activities. Then tickets have been created and have been assigned to the different milestones thus guaranteeing the monitoring of the development of the platform.
2.1.4 Maven and Artifactory

Maven [11] and Artifactory [12] have played the same central role of building tools as they had in Y1 even though with some changes.

As a matter of fact, though the dependency management has been still based on Maven and Artifactory, the management of the versioning of the artifacts that compose the integrated platform has been slightly changed. In order to resolve the issues that showed up in year one, related to the difficulty of reproducing on different machines the same build environment in terms of dependencies, there were two possible choices. The first one was to enforce a more strict policy for the update of the version number of the artifacts and the second one was to completely avoid the usage of incremental numbering of the artifacts until a stable development had been reached. The second solution has been judged as the easiest to implement by the developers and has actually allowed us to avoid the above mentioned problems.

Apart for the mentioned changes, Maven has still been used exploiting all its capabilities, that is for compiling, managing dependencies, packaging, deploying, versioning, testing, documenting and reporting the software developed within the project. Furthermore, it has been used for checking the code style, for testing the frameworks in cooperation with JUnit [13], for performing test coverage analysis in cooperation with Cobertura [14], for generating the website, for applying continuous integration techniques as will be explained in the next section, using the Continuum tool [15], and finally, for generating the documentation of the source code.

For what we have said above about the versioning policy, the role of the Artifactory repository has been less important since the platform build policy made it preferable to generate the artifacts directly in the local environment without resorting to the repository. Nevertheless the Artifactory and its Web interface [16] have been operative and useful also during Y2. All libraries that are generated by the SLA@SOI platform modules have been deployed onto the Artifactory repository, installed onto the FZI server, using the Maven
configuration. Not only are the internally generated artefacts hosted there, but also 3rd party libraries that are utilised by SLA@SOI modules are maintained and made available in this common place, thus allowing the SLA@SOI Subversion repository to contain only source code, avoiding the burden of hosting libraries and other 3rd party binary files.

![Figure 3: SLA@SOI Artifactory Web console](image)

2.1.5 Continuum

As well as for Trac, also the Continuum tool has taken advantage of a more mature approach. The setup of the Continuum service [17] has been anticipated with respect to what had happened in Y1 with a neat advantage from the point of view of the build management and the bug’s discovery and fixing.

The notification mechanism, though simplified in order to avoid spamming, has delivered a daily report about the status of the platform with a link to the web console where possible failures in the build of the platform modules were reported in details along with also a historical overview of the builds. This has demonstrated to be very motivating for the developers in order to reduce the time-to-fix for bugs and issues and the overall development within the project has taken great advantage from this.
2.2 Organizational Procedures

From the point of view of the organization procedures there have been no radical changes in the second year.

The only major and noticeable improvement with respect to the first year has been the establishment of very clearly defined and detailed guidelines for the development of the platform modules.

In addition to the ones already established in the first year of the project, like for instance for SVN and Maven, which have been reinforced and improved, and in addition to the quality assurance techniques that will be described in the next section, a set of other prescriptions have been shared in the very early phases of the development to guarantee that the implementation was based on a commonly agreed set of principles and best practices so that the integration might take advantage of a better prepared background.

The main concern has been the establishment of an approach to the development of the components of the platform that took into account from the start the final OSGi [18] deployment - which will be described in the next section – in which the modules have to run. This has been described through a series of tutorials and guidelines about OSGi-oriented development, about the development of Web services within OSGi, and about the best practices for debugging in such a development environment. Then, we have tackled the management of the logging and of the properties configuration. Finally, a common approach to the management of persistency within the platform has been arranged so that the usage of databases throughout the platform could be consistent.

The result of this effort has been the avoidance of the appearance of a series of issues related to configuration and integration that had delayed the delivery of the platform in the first year.
2.3 Quality Management

As for the first year of the project the attention for ensuring that the developed code is of high quality has been maintained. As we have said in the previous section, guidelines have been established at the level of code production to guarantee the delivery of good quality code. The start of the activities related to the Open Source release of the platform has stimulated even more attention to the establishment and monitoring of unit test activities, coding style and best practices constraints.

The policies have not changed so much with respect to year one (see the M12 deliverable for more details), but after the initial phases of the development they have been monitored more heavily.

2.3.1 Unit Testing

The production and unit tests monitoring started earlier in this second year with respect to the first one. A wiki page has been setup for these purposes and this has had the effect of stimulating the developers to produce and maintain effective unit tests that have eased a lot the following integration test phases.

JUnit has been used as the unit testing framework, and Maven has been used to manage and automate the execution of the tests. The Cobertura Code Coverage report has been used to track code coverage as well as highlight individual lines of code that have not been exercised during unit tests.

The Maven Surefire [19] report is used to summarise the amount of unit tests that have been executed, their names and success rate. Currently the number of successful unit tests that are attached to the source code of the platform is more than 250.

The coverage of the tests with respect to the implemented source code is analyzed by the Cobertura tool that is integrated within the Maven configuration files and that can be performed automatically.

The testing reports have become very useful towards the end of the second year of the project in parallel with the kick-off of the activities related to the Open Source release of the platform. These reports have become an important means by which to check some of the quality parameters required for releasing some of the framework components as Open Source.

The unit testing report and the test coverage report are part of the overall SLA@SOI platform site that is automatically generated with Maven and that can be found at [20].

2.3.2 Coding Practices

Although neither mandatory nor necessarily traceable, many developers within SLA@SOI have adopted coding practices to maximise the quality of their code. Some developers have configured their editors to configure warnings as errors to remove all potential ambiguities or identified inefficiencies in their code.

Other reports, such as the CPD (Copy Paste Detector) report and PMD (Programming Mistake Detector) reports are configured within Maven to help developers identify opportunities for refactoring away duplicated code, locate inaccessible or unused code, and detect other potential problems.

Code has regularly been refactored as it matures, helping create more generic, maintainable, tested code.
The report about the coding practices used for the development of the SLA@SOI platform is part of the overall site generated with Maven that can be found at [20].

2.3.3 Coding Style

Most developers have developed their own coding style over time – how they arrange brackets, whitespace, variable naming conventions and where, if anywhere, they document their code. If not managed, a project with multiple developers can quickly create inconsistent code that is difficult to read and almost impossible to maintain.

To avoid this problem in SLA@SOI a standard coding style has been adopted. This is monitored by performing a CheckStyle analysis which has been customised by the development team to detect all relevant violations. Developers have run the analysis either from within their development environment, or using the build management tool. As the majority of developers on the project are using the Eclipse [21] integrated development environment with powerful auto-formatting support, a custom Eclipse formatter has also been created to automatically apply as much of the style as possible. With one click of the Format command an entire project can now have the SLA@SOI formatting applied. This can reduce style violations dramatically. Any remaining violations are tackled manually by developers.

Also the check of the quality of the delivered code is integrated within Maven build procedures and is then automatically performed. It is based on using a style-check template against which the code is matched. Such a template is derived from one of the standard ones that are available and that the Eclipse platform incorporates and uses as a default.

The report about the adherence of the coding style used for the development of the SLA@SOI platform to the defined template is part of the overall site generated with Maven that can be found at [20].

2.3.4 JavaDocs

Although appropriate method and variable naming conventions help self-document code, the project uses JavaDocs to summarise the individual packages, interfaces, classes, attributes and methods.

The CheckStyle report detects any aspects of the interface that are not documented, and Maven automatically builds the hyperlinked HTML JavaDocs by default.

The overall amount of documentation of the source code is captured by the following figures. More than 25,000 lines of JavaDocs comments are included in the source files of the platform that explain the behaviour of the modules and provide a complete documentation to the framework API. About 15,000 more lines of Java comments are spread throughout the code to document the internals of the implementation.

The JavaDocs API is part of the overall site generated with Maven that can be found at [20].

The API documentation has also been produced separately and is available for browsing at [22] or as a zip file at [31].
2.3.5 Maven Generated Website

One of Maven’s most powerful commands is site, which can build a complete HTML website describing the project and reporting on the state of the code. The various web-pages in the site are derived from the metadata in the projects POM file, and the Maven reports that are configured to run against the source code.

The SLA@SOI project has used this facility to build the site of the SLA@SOI platform including all the possible info about the source code. The SLA@SOI platform website is available for browsing at [20] or as a zip archive at [32].

3 Technical Integration

The integration approach used in the second year of the project for delivering the SLA@SOI platform has radically changed from the one used in the first year. The need of satisfying a wider range of requirements and to support a broader set of features, along with the overall revision of the framework architecture has determined a shift from the static Spring-based approach used in Y1 to a dynamic OSGi-based approach for Y2. This has guaranteed the possibility of delivering a flexible and adaptable software framework that can be integrated, configured, and extended in different environments. In the following sub-sections we will present the approach, leaving to section 5.2 the presentation of the guidelines the actual adoption of the framework.

In particular, section 3.1 is devoted to the description of the integration approach in general and in particular to the presentation of the OSGi framework that has been used as the basis for gluing the components of the framework. section 3.2 is dedicated to the explanation of the design-time choices about the integration of the platform, while section 3.3 addresses the run-time environment in which the SLA@SOI platform is meant to run.

3.1 Approach

The approach chosen for integrating the SLA@SOI platform in the second year of the project is based on the usage of OSGi technology [18].

OSGi can be defined as the dynamic module system for Java. It provides the standardized primitives that allow applications to be constructed from small, reusable and collaborative components. More technically, it is a middleware providing a service-oriented, component-based environment for developers that offers standardized ways to manage the software lifecycle. To minimize the coupling between components, as well as make these couplings managed, the OSGi technology provides a service-oriented architecture that enables these components to dynamically discover each other for collaboration.

OSGi has a layered model as shown in Figure 5 (taken from [18]) in which the elements can be described as follows:
Figure 5: OSGi stack

- **Bundles** - Bundles are the OSGi components made by the developers.
- **Services** - The services layer connects bundles in a dynamic way by offering 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.

Modularity is thus the basic concept at the core of the OSGi specifications and it is embodied in the *bundle* concept. In Java terms, a bundle is a plain old JAR file. However, where in standard Java everything in a JAR is completely visible to all other JARs, OSGi hides everything in that JAR unless explicitly exported. A bundle that wants to use another JAR must explicitly import the parts it needs. By default, there is no sharing. Though the code hiding and explicit sharing provides many benefits (for example, allowing multiple versions of the same library being used in a single VM), the code sharing was only there to support OSGi services model. An important aspect related to the modularity of OSGi is, then, the fact that its services’ model describes the rules by which bundles collaborate each other.

The usage of the OSGi model for integrating the components of the SLA@SOI platform addresses the need of a flexible and dynamic composition of the framework that can be composed and enriched using the basic building blocks of the architecture so to satisfy the requirements of the different domain in which the platform is exploited. As a matter of fact, the different Use Cases developed in the B-line work packages of the project have taken advantage of this approach by integrating only those part of the framework that were actually needed – the core of the platform – and then supplying those domain dependent *bundles* that were necessary for the specific features of the Use Case. Even more, the OSGi reaches such a high level of modularity that this addition or enabling of specific components can occur at run-time without having to restart the framework.

### 3.2 Design-time

At design time, OSGi has been injected into SLA@SOI platform by making usage of the Spring Dynamic Modules framework [23]. Spring DM is an extension of the standard Spring Framework [24] that was used instead for the integration of the platform in year one.
While the Spring Framework provides simply a lightweight container and a non-invasive programming model enabled by the use of dependency injection, AOP, and portable service abstractions that is useful for static applications, Spring Dynamic Modules makes it easy to write Spring applications that can be deployed in an OSGI execution environment, and that can take advantage of the services offered by the OSGI framework. Spring's OSGi support makes development of OSGI applications simpler and more productive by building on the ease-of-use and power of the Spring Framework.

All of the core components of Y2 SLA@SOI platform have been developed as bundles. From the point of view of the technical implementation this means that all the modules, that have been developed as Maven projects, have used the Maven plugin named Apache Felix Bundle [25] for generating OSGI Bundle rather than the simple Java compiler plugin for creating simple jar files.

The generation of such bundles artifacts from simple Java code through the Apache Felix Bundle plugin is the result of a particular configuration that has to be imposed onto the classical Java module. A couple of configuration files are sufficient to let the Maven plugin transform what is a simple Java component first of all into a Spring bean and then into an OSGi bundle; that is to say, a binary unit of code that is ready to be activated onto an OSGi run-time and is ready to make its services available to the other bundles running on the same Java virtual machine. The full details of such a development process are further described from a developer’s point of view in section 5.2.7.

3.3 Run-time

The choice of OSGi as the integration mechanism for the platform has not been the last step in building the technical environment for the Y2 platform. OSGi is simply a standard and consists of a set of specifications whose implementations have been realized by a number of different providers. The most well known of these are: Equinox [26], Felix [27], Knopflerfish [28] and Concierge [29]. The SLA@SOI project, rather than sticking to any of these has chosen a different approach that is to adopt a run-time environment that could cope with many of them. Such an environment is named Pax-runner.

Pax-runner [30] is a tool to provision OSGi bundles in all major open source OSGi framework implementations (Felix, Equinox, Knopflerfish and Concierge). Using Pax-runner is very simple. Once you have installed it, you can use the following simple command `pax-run` to start it. This will start up Pax-runner with default options and with no provisioning. Of course, it is then possible to specify several useful option, e.g., for choosing a specific OSGi framework to be started and its version and even more. Instead of providing options at the command line it is possible to supply them to Pax-runner using a text file. By default, Pax-runner will look for, and use if available, a file named `runner.args` from the current directory. Options in such a file must be provided one per line. Besides the target OSGi framework and its version, it is possible to set other options such as the log level, the system properties and the profiles to be used.

Profiles offer a way to identify a collection of bundles that we’d like Pax-runner to install and start for us without explicitly identifying them one by one. For example, mostly every application needs some form of logging. With Pax-runner’s `log` profile, we need not worry ourselves with figuring out which bundles to install because we just can tell Pax-runner that we want logging by starting it like this:

```
pax-run --p=e --profiles=log
```
Pax-runner has a built-in profile that facilitates easy startup of Spring DM enabled bundles. To achieve this you just have to use a adhoc option as for instance:

\[\texttt{pax-run --profiles=spring-dm}\]

which starts up all the spring dm related bundles.

The SLA@SOI platform has adopted the Pax-runner launcher by creating its own runner.args file which defines the provisioning of all the necessary components. The project has used the profile concept to identify and load the collection of bundles related to each of the platform component. For instance the Business SLA Manager, the Generic SLA Manager, the Infrastructure SLA Manager, and so on have their own profile which defines the sub-bundles they have and that will be launched at run-time.

### 3.4 Integration Testing

The testing of the integrated platform is based on a set of three main scenarios. By putting together some of the multiple features of the individual components of the framework, these three scenarios demonstrate the capability that the individual platform modules have to communicate and cooperate in order to achieve high level goals. These goals correspond to the main activities that the platform is expected to carry on, namely negotiating a SLA, provisioning a SLA and monitoring the SLA.

The first integration test is about the Negotiation of a SLA. The sequence of interactions that are necessary between the components of the platform in order to reach the agreement on the service level, is showed in Figure 8.
Figure 6: Negotiation scenario
The second integration test, that depends upon the previous one, is the Provisioning scenario in which the services and resources, that were agreed within the negotiated SLA, gets provisioned. This is represented by the sequence diagram in Figure 7.

![Sequence Diagram](image)

**Figure 7: Provisioning scenario**

Finally, the third scenario, that depends upon the two previous ones, is about the Run-time scenario. It deals with the monitoring of the services and resources that have been provisioned at the previous step. It also consider reactions to possible violations of the provisioned SLA. Its representation, in the form of a sequence diagram, is presented in Figure 8.
Of course, these three tests do not cover the entire spectrum of capabilities of the platform both from a high level perspective and also in terms of functionalities that the single components provide. Nevertheless they have played a central role in assessing the successful integration of the components of the platform into a unique framework for the overall management of SLAs.

4 Framework Usage

This chapter presents a description of how to use the SLA@SOI framework for running a set of sample scenarios. The sample scenarios are taken from the integration tests used for testing the platform and are based on the ORC which, as per the amendment of the DoW, has been used also during the second year of the project for building around it the integration of the SLA@SOI framework. The perspective used here is a user’s one, thus leaving for the next chapter a more developer oriented description of the platform. The following sections describe all the steps that are needed, from download to launch, in order to execute sample tests.

4.1 Requirements

The requirements for running the Integration Tests used for testing the platform are the following:

- Java Runtime Environment 1.5
- Pax-runner 1.3.0

The execution of the platform described in the following sections assumes that a number of services that are already configured within the platform are actually available at different locations. We refer in particular to a DBMS instance available on FBK’s servers, an Event Bus instance running on an Intel’s server, and others.
that are available at different locations. The following description, thus, concentrate only on the execution of the core platform.

The platform has been tested both on Windows and on Linux systems.

### 4.2 Download

The binary distribution of the SLA@SOI platform, packed in an zip archive, is available for download at [33]. The archive contains a pre-compiled version of the SLA@SOI platform source code that is ready to be executed as described in the following sections.

### 4.3 Installation

The installation of the binary version of the platform is very simple. You have simply to unzip the archive downloaded as per section 4.2 in a folder on your machine.

After unzipping the binary package you will see a folder structure like the one in Figure 9.

```
root/
|--- Readme.txt
|--- startup.bat
|--- DB/
|--- ORC-data/
|   |--- ...
|   |--- ...
|   `--- ...
|--- osgi-config/
|   |--- ...
|   |--- ...
|   `--- ...
|--- pax-run/
|   `--- runner/
|     |--- bundles/
|     |--- cache/
|     |   `--- equinox/
|     |     |--- ...
|     |     |--- ...
|     |     `--- ...
|   `--- profiles/
|       |--- ...
|       |--- ...
|       `--- ...
```

**Figure 9: Binary package structure**

The contents of the different folders is the following:

- **Readme.txt**: instructions about how to handle the binary distribution;
- **startup.bat**: the script that configure and start the SLA@SOI platform;
- **DB**: the scripts that are needed to create and populate the persistency component (MySQL DB) of the platform;
- **ORC-data**: the templates and other configuration files related to ORC. This folder is referenced from within the source code of SLA@SOI Platform in order to load the ORC related configuration data;
4.4 Configuration

The configuration of the platform requires a set of steps (mainly the set of a couple of environment variables) that, within the binary distribution, are already comprised in the execution batch script (startup.bat) that comes along with the distribution package. Nothing else is required to configure the platform.

4.5 Run

In order to run the platform, simply run the startup.bat script that is contained in the root folder of the distribution archive. It will start the OSGi run-time and after a few seconds will load all the bundles that are part of the SLA@SOI framework.

At this point the OSGi console is available for performing all the commands related to the OSGi bundles life cycle (activation, deactivation, start, stop, etc.).

In order to avoid too verbose logging of the execution, it is sufficient to run at this point the command:

```
start <n>
```

where `<n>` is the number of the bundle named `org.ops4j.pax.configmanager` (usually it is number 81).

Now you are ready to execute one of the test sample that are exposed by the IntegrationTest bundle. In order to do this you simply have to run, at the OSGi console, the command:

```
start <n>
```

where `<n>` is the number of the bundle named `IntegrationTestModule`.

By default, the platform is configured to execute the Negotiation Scenario of the ORC use case.

If you want to run other scenarios and interactions, you must:

- set the relevant scenario identificator (`scenarioId`) and operation code (`opCode`) within the `config.properties` file that is located in the folder `osgi-config/Integration` inside the binary distribution archive
- refresh the integration bundle by running the command:

```
refresh <n>
```

where `<n>` is the number of the bundle named `IntegrationTestModule`. 
5 Framework Development

This section describes the platform from a developer’s point of view thus enabling a possible re-use and adoption of the platform in other domains than the ones used for integration and for the demonstration purposes in the B-line work packages. In particular, section 5.1 describes the structure of the source code of the platform; section 5.2 and its sub-sections provide all those information that are necessary to re-use and/or extend the individual main components of the core of the SLA@SOI framework, so that they can be tailored to possible new applications scenarios; finally, section 5.3 supplies the complete instructions for building and executing the platform starting from the download of source code.

5.1 Structure of the Source Code

The source code of the SLA@SOI platform is maintained in a Subversion repository that is hosted at [6].

The repository structure follows the best practices for Subversion repositories. Its organization and the procedures for its management have been agreed by the consortium and are described both in [2] and in section 2.1.2 of the present deliverable.

The structure of the repository is organized in a set of modules that reflects the SLA@SOI architecture. For each functional component, a module exists in the repository trunk. Modules are then internally organised in sub-modules. Namely, the top-level components of the framework are:

- businessManager
- genericSLAManager
- businessSLAManager
- softwareSLAManager
- infrastructureSLAManager
- softwareServiceManager
- infrastructureServiceManager
- monitoringManager
- serviceEvaluation
- manageabilityAgent

The adoption and re-use of most of these modules constitute the subject of section 5.2. Besides them, other functional and non-functional modules are located in the root folder as well. The other functional modules like, for instance, ISMOCCIBundle and service-advertisement are complementary to the main components and will not be the subject of any further explanation in the following. The non-functional modules are used for achieving different goals. First of all, the pax-runner and profiles modules make it possible to assemble the launch of the platform; secondly, the common and models module provide shared facilities and, finally, the resources modules enable features related to the check of the code style.

The build of the source code is handled by Maven. The entire hierarchy of functional modules inside the repository is mavenized, which means that every module contains a Maven configuration file, named pom.xml. This file enables execution of the tasks related the lifecycle phases of that piece of code, e.g. clean, build, testing, library packaging, local install and remote deployment. Maven’s modular architecture allows the usage of plug-ins for performing a number of other tasks related to the development and maintenance of the source code as, for instance, the automated generation of documentation, the check of the coding style, the management of bugs and the generation of a Web site that
contains all the information about the module. These tasks are configured inside the pom.xml file, which contains all the other possible information about the corresponding module, e.g. names of the developers.

The repository exploits the inheritance and override mechanisms of Maven for multi-module projects. Inheritance allows within the configuration file of a father module the collection of all configurations that are common to a set of children modules, leaving up to the children to override them if necessary. For commonly configured tasks like documentation generation, style-checking and similar this has been implemented within the SLA@SOI source code.

For more details about the management of dependencies management please refer to section 2.1.4.

5.2 Platform Adoption

The following sections – one per each of the main platform components – provide details about the adoption of the components themselves explaining how to take advantage of them from a developer point of view thus providing hints at the source code level about their possible re-use and extension. The last subsection is devoted to explain how to integrate a newly developed component into the SLA@SOI platform.

5.2.1 Generic SLA Manager Adoption

The adoption of the Generic SLA Manager (G-SLAM) involves a set of standard steps for the instantiation of generic, available components, and the insertion of some use-case specific components. More specifically, the G-SLAM includes the following generic components:

- Authentication proxy
- Syntax Converter (reference implementation for SLA@SOI model’s XML rendering, and WS-Agreement)
- Protocol Engine
- Monitoring Manager
- Advertisements System (Publish/Subscribe bus)
- SLA Registry
- SLA Template Registry

First of all, for each of such components it is necessary to create and edit an associated Pax-runner profile. Such profiles directs Pax-runner in loading and initializing each component within the OSGi framework.

Then it is necessary to generate a SLA Manager Context using the G-SLAM’s createContext service. This service creates a new reference to a SLAM, and instantiates new instances of the generic components, which are then injected into the newly referenced SLAM. Figure 10 shows some the code snippet which is necessary to create a SLAM’s context.

```java
SwSLAMActivator.INSTANCE++;
SLAMConfiguration swConfig = gslamServices.loadConfigurationFrom("swslam.instance1.cfg");
swContext = gslamServices.createContext(swConfig.name + "+
SwSLAMActivator.INSTANCES, swConfig.epr, swConfig.group);
```
SwAuthorization auth = new SwAuthorization();
swContext.setAuthorization(auth);

**Figure 10: Code for creating a SLAM's context**

After these steps, the use-case-specific components, i.e..

- The Planning/Optimization Component, and
- The Provisioning/Adjustment Component.

need to be injected. These two components bear the knowledge that is specific to each use case. Basing on such knowledge, they can make complex decisions during the negotiation, provisioning and runtime phases of the SLA management. Having implemented the respective standardized interfaces, and built the components in the form of OSGi bundles, one has to implement code similar to the one shown in Figure 11.

```java
// inject the POC into the swContext
PlanningOptimization newPOC = swPOCBuilder.create();
swContext.setPlanningOptimization(newPOC);
injectIntoContext(newPOC, swContext);

// inject the PAC into the swContext
ProvisioningAdjustement newPAC = swPACBuilder.create();
swContext.setProvisioningAdjustement(newPAC);
injectIntoContext(newPAC, swContext);
```

**Figure 11: Code for injecting POC/PAC into SLAM**

### 5.2.2 Business Components Adoption

The Business Management adoption permits the exploitation of the platform’s functionalities that are related to:

- product creation and discovery
- product lifecycle management
- customer relationships and feedback
- prices management based on promotions, policies and customizing.

The Business components are divided in three main components:

- Business Manager
- Business SLA Manager
- Business Web Tools (GUI)

The Business Manager component is directly connected to the Business SLA Manager and in particular to two of its internal component that is to say the Planning and Optimization Component (POC), and the Planning and Adjustment Component (PAC). A graphical user interface allows to to manage all the business elements, and is connected with the other components.

The Business SLA Manager is a specialization of the Generic SLA Manager. Inside the B-SLAM, the POC and PAC components are connected with the Business Manager.

The code snippet in Figure 12 can be used for enabling the usage of the Business Manager inside the platform:

```java
// inject Context into productDiscovery
LOGGER.info("inject Context Into discovery");
injectIntoContext(discovery, bContext);
```
// inject Context into postSale
LOGGER.info("inject Context Into postSale");
injectIntoContext(customerRelations, bContext);

Figure 12: Code for injecting Context into Discovery and PostSale

The way for injecting the specific implementation of the POC and PAC interfaces of the Generic SLAM into the B-SLAM is shown in Figure 13:

```java
@ServiceReference(filter = "(proxy=business-poc)")
public void setPOCService(PlanningOptimization bPOCServices) {
    LOGGER.info(business-POC injected successfully into business-slam);
    this.bPOCSServices = bPOCServices;
}
@ServiceReference(filter = "(proxy=business-pac)")
public void setPACService(ProvisioningAdjustement bPOCServices) {
    LOGGER.info("business-PAC injected successfully into business-slam");
    this.bPACSServices = bPACServices;
}
```

Figure 13: Code for injecting POC/PAC into B-SLAM

The code snippet in Figure 14 allows to inject the business authorization instead of using the generic solution provided along with the G-SLAM:

```java
Authorization authorization = new SWAuthorization();
```

```java
@ServiceReference(filter = "(proxy=business-auth)")
public void setBAuthorization(Authorization bAuthorization) {
    LOGGER.info("business-Auth injected successfully into business-slam");
    this.bAuthorization = bAuthorization;
}
```

Figure 14: Code for injecting Business Authorization

Finally, in order to insert customer relations and product discovery injection, the code to be used is the one in Figure 15:

```java
@ServiceReference
public void setICustomerRelations(ICustomerRelations customerRelations) {
    LOGGER.info("ICustomerRelations injected successfully into business-slam");
    this.customerRelations = customerRelations;
}
@ServiceReference
public void setProductDiscovery(ProductDiscovery discovery) {
    LOGGER.info("ProductDiscovery injected successfully into business-slam");
    this.discovery = discovery;
}
```

Figure 15: Code for injecting Customer Relations and Product Discovery
5.2.3 Software Service Manager Adoption

The Software Service Manager components delivers extensibility mechanisms which can be used to introduce domain specific customizations to fit the needs of each particular use case. This section describes how domain specific customizations can be achieved through these extensibility mechanisms. The domain specific customization is enabled through two interfaces which use cases can implement to plug in domain specific service management logic. These interfaces are IProvisioning and IMakeabilityAgentFacade. These interfaces are described in the following paragraphs.

IProvisioning

The interface IProvisioning encapsulates actions for deploying, configuring and startup/shutdown related functionalities of software services. The functionality defined in IProvisioning is predominantly domain and software service specific. Therefore, the Software Service Manager component adopts a plug-in type mechanism whereby domain specific plug-in (implementing the IProvisioning interface) can be deployed so that the Software Service Manager component can use these plug-ins to perform software service configuration, startup and shutdown operations. IProvisioning is domain-specific interface that is to be implemented by all use cases employing the Software Service Manager component. This interface encapsulates the basic functionality for starting and stopping service instances. Furthermore, it allows the Software Service Manager module to access (or create) a domain-specific Manageability Agents for an existing service instance. Finally, all implementations of this interface encapsulate the knowledge necessary to determine the endpoints for their services. The IProvisioning interface contains the following methods:

5.2.3.1.1 startServiceInstance

```java
void startServiceInstance(ServiceBuilder builder,
                          Settings connectionSettings,
                          String notificationChannel)
                              throws ServiceStartupException,
                              MessagingException;
```

This operation starts new service instance according to the configuration specified in the builder object. Once the service is available a notification is sent over the channel specified.

The parameter of this operation are:

- **builder** - The builder object used to set up the service instance. The builder must be fully specified, i.e., all dependencies to external services have to be resolved.
- **connectionSettings** - Connection settings for the message-oriented middleware hosting the notification channel.
- **notificationChannel** - The channel used to notify the responsible parties once the service's provisioning has been completed.

The exception thrown by this operation are:

- **ServiceStartupException** - Thrown if an error occurs during the initialization of the provisioning process.
- **MessagingException** - Thrown if the messaging system cannot be reached.
5.2.3.1.2 stopServiceInstance

```java
void stopServiceInstance(ServiceBuilder builder)
```

This operation is used to Stop a running service instance.
The parameter of this operation is:
- `builder` - The builder object used to create the service instance.

5.2.3.1.3 getEndpoints

```java
List<Endpoint> getEndpoints(ServiceBuilder builder)
```

The `getEndpoints` operation determines the endpoints of a service instance created by the given builder object.
The parameter of this operation is:
- `builder` - The builder object used to create the service instance.
The return type of this operation is:
- the list of Endpoints of the service.

5.2.3.1.4 getManageabilityAgent

```java
IManageabilityAgentFacade getManageabilityAgent(ServiceBuilder builder)
```

If available, this methods returns a facade to the manageability system of the given, domain-specific service.
The parameter of this operation is:
- `builder` - The builder object used to create the service instance.
The return type of this operation is:
- the ManageabilityAgent of the service instance.

**IManageabilityAgentFacade**

This interface provides a generic access point for the Software Service Manager component to use case specific ManageabilityAgents. To correctly implement this interface, you need to consider the following:
- there exists exactly one `IManageabilityAgentFacade` per service instance;
- if multiple service instances are managed by a single agent, the requests are dispatched by the Façade;
- the methods of the Façade are kept on a general level;
- basically, all more specific actions of a domain-specific `ManageabilityAgent` have to be realized through the `executeAction` method.

5.2.3.1.5 getInstance

```java
ServiceInstance getInstance()
```
The return type of this operation is:
- the ServiceInstance that is controlled by the ManageabilityAgent.

### 5.2.3.1.6 getSensorSubscriptionData

```java
List<SensorSubscriptionData> getSensorSubscriptionData()
```

The return type of this operation is:
- the list of available sensors and where to find their data.

### 5.2.3.1.7 configureMonitoringSystem

```java
void configureMonitoringSystem(IMonitoringSystemConfiguration configuration)
```

Applies the given configuration to the monitoring system of the managed Service Instance. The given configuration replaces the previous one (if any).

The parameter of this operation is:
- configuration – Configuration to be applied.

### 5.2.3.1.8 deconfigureMonitoring

```java
void deconfigureMonitoring()
```

Disable the currently applied monitoring configuration.

### 5.2.3.1.9 executeAction

```java
IEffectorResult executeAction(IEffectorAction action)
```

Execute domain-specific action for the adjustment of a Service Instance.

The parameter of this operation is:
- action - Domain-specific implementation of the IEffectAction interface that contains information about what action to execute and its parameters.

The return type of this operation is:
- the result of the action in domain specific format.

### 5.2.4 Infrastructure Service Manager Adoption

The Infrastructure Service Manager (Infr-SM) is the component responsible for the creation, lifecycle management, optimisation and manipulation of infrastructure resources. Infr-SMs are manipulated by both Infrastructure PACs and Infrastructure POCs, and so, for convenience a single Infr-SM Proxy can be used to provide a single convenient access point to the Infr-SM.

An OCCI-compatible Infrastructure Service Manager has been implemented by the A4 work package in the second year of the project. It exposes an OCCI interface on top of the Apache TASHI provisioning system. It can provision virtual
machines of required sizes hosted by either the KVM and Xen hypervisors. An Infrastructure Service Manager Proxy has been implemented that simplifies manipulations of the Infrastructure Service Manager by the Infrastructure PAC and Infrastructure POC. The implementations of the Infrastructure Service Manager and its proxy are described in detail in Deliverable D.A4a SLA Aware Infrastructure Management.

The recommended approach for developers of PACs and POCs that need to manipulate the OCCI Infr-SM is to access it via the Infr-SM Proxy.

Once the Infr-SM Proxy library is imported, the developer can create a Compute resource to represent the particular VM(s) required. Code to represent one particular VM is reproduced in Figure 16.

```java
// We create a Compute Resource first
Category computeCategory = new Category();
computeCategory.setTerm(Terms.COMPUTE);
computeCategory.setScheme(Schemas.OCCI);
computeCategory.setLabel(Labels.COMPUTE_RESOURCE);

// Appliance Category (What image type we should use)
Category applianceCategory = imageregistry.getCategoryByID("Ubuntu 10.4 x86_64");

// SLA Under what SLA Type we should provision this resource (Domain Specific)
Category slaTypeCategory = slatyperegistry.getCategoryByID("GOLD");

// Location
Category locationCategory = locationregistry.getCategoryByID("Ireland");

// We create the actual Compute configuration
Compute computeConfiguration = new Compute();
computeConfiguration.setCpu_cores(Cores.FOUR);
computeConfiguration.setCpu_speed(CpuSpeed.Medium);
computeConfiguration.setMemory_size(Memory.Large);

// We set the desired hostname (Domain specific)
computeConfiguration.setHostname(hostName);

// We add the Categories to a Set
Set<Category> categories = new HashSet<Category>();
categories.add(applianceCategory);
categories.add(computeCategory);
categories.add(slaTypeCategory);
categories.add(locationCategory);

// We associate the categories to the compute configuration
computeConfiguration.setCategories(categories);

// We create a set of compute configurations and add
// our Compute Configuration
Set<Compute> computeConfigurations = new HashSet<Compute>();
computeConfigurations.add(computeConfiguration);
```

**Figure 16: Code for representing a VM**

A ProvisionRequestType is then created, and the Compute representation added to it, as in Figure 17.
We create a ProvisionRequestType and add the Compute representation:

```java
ProvisionRequestType provisionRequestType = new ProvisionRequestType();
provisionRequestType.setKinds(computeConfigurations);
```

// Start now, run forever:
```
Date startTime = new Date(System.currentTimeMillis());
Date stopTime = startTime;
provisionRequestType.setProvStartTime(startTime);
provisionRequestType.setProvStopTime(stopTime);
```

// Notification URI for Infra. Provisioning and Adjustment Component:
```
provisionRequestType.setNotificationUri(iPAC.notificationURI);
```

// Monitoring configuration:
```
provisionRequestType.setNotificationUrl(MonitoringConfiguration.DEFAULT);```

Figure 17: Code for ProvisionRequest and Compute operations

Finally, the `provision` method is called to perform the actual provisioning, as in Figure 18.

```java
// We call the ISM and try to provision
ProvisionResponseType provisionResponseType = null;
String infrastructureID = null;
try {
    provisionResponseType = infrastructure.provision(provisionRequestType);
    infrastructureID = provisionResponseType.getInfrastructureID();
} catch (DescriptorException e) {
    e.printStackTrace();
}
```

Figure 18: Code for provisioning a VM

To manage the provisioned VM(s), the `getDetails` method can be used as illustrated by the code in Figure 19.

```java
// We try to getDetails of our provisioning
try {
    provisionResponseType = infrastructure.getDetails(infrastructureID);
} catch (UnknownIdException e) {
    e.printStackTrace();
}
```

```
List<EndPoint> endPoints = provisionResponseType.getEndPoints();
for (Iterator<EndPoint> iterator = endPoints.iterator();
    iterator.hasNext());
```

```java
   EndPoint endPoint = iterator.next();
   // We get the url of this resource
   URL url = endPoint.getResourceUrl();
```
Finally, to stop the VM the `stop` method can be invoked as in Figure 20.

```java
// We stop our provisioning
try {
    infrastructure.stop(infrastructureID);
} catch (UnknownIdException e) {
    e.printStackTrace();
}
```

### Figure 20: Code for stopping the VM

#### 5.2.5 Service Evaluation Adoption

For the development of the Service Evaluation component a main goal was to clearly separate three parts that make up the complete functionality of design-time prediction of services. This design is sought to improve reuse and easy use-case adoption of the single units.

First, a sub-component is dedicated to the interaction with the Planning and Optimization Component (POC). This interaction includes bridging the world of SLA agreements – for arrival rates, data volumes as well as infrastructure etc. – and the world of design-time service models. In absence of explicit knowledge of what kind of service is actually predicted, the connection of SLA terms and elements of the design-time service models has to be established. Since this is a generic task needed for each adoption of the framework, it is separately bundled and always part of A6 and thus the framework. The provided interface is generic in the sense that it doesn’t make assumptions about the kind of SLA Template being passed. The required interface contains methods for repository access as discussed in the following.

Second, a model repository sub-component is included in the service evaluation component. The rationale is to allow for access to both models that are deployed as part of the adoption of the framework and models created outside the framework after adoption. Querying of system models is therefore not restricted to a given set of models that are shipped with the adopted framework. By that, we seek to leverage the opportunities of design-time prediction (with respect to accuracy and quality) and the options for reworking the models once the framework is launched. Hence, the flexible model repository component decouples the logic operating on service models. In fact, for use-case adoption there is no need to rewrite code within this component. Use-case specific adoptions are realized by modeling the type of services (as stubs to be filled by terms coming from SLAs) that should be part of the repository. Adopting the repository is accomplished by adopting the model stubs shipped with the adopted framework. This enables to detach the tasks of adopting the different framework components, which is a more technical task, and of designing use-case specific models for the services that will be part of the negotiation and finally service realization.

Third, all code that actually runs simulations or analyses on the basis of models remains separated and is kept outside the framework. This is due to the following reasons. The Palladio Component Model (PCM) and the associated tools (editors, etc.) for performance and reliability modeling of services provide a large toolkit that can be used by service providers to develop the system models in a standalone, Eclipse-based manner. Since the toolkit is already available as an Eclipse feature it is not part of the framework or one of its adoptions. It is considered rather supplementary software. However, the core PCM components that predict QoS attributes of the fully parameterized services are used as
external libraries. To support a distributed setting we decided to establish a web service-based interaction between the prediction client (being part of the framework) and the simulation server (running the analyses). As a considerable improvement to year one, the current solution doesn’t make any assumptions about (physical) distribution of model data, simulation platforms etc. For use case adoptions this means that the bundles providing the simulation service don’t have to be modified at all. For the SLA@SOI B-line use cases the simulation server that is employed for integration testing can be used without modification.

In summary, the service evaluation component eases, by design, adoption by use cases other than the ORC. Flexibility highly results from the dynamic model repository mechanism and the web-service based invocation of the simulation service.

### 5.2.6 Monitoring Adoption

The monitoring module of the platform comprises a set of general reusable components that provides a flexible solution for all possible domains.

On the instrumentation level proper data has to be measured and captured regularly. Ganglia connector can be used to benefit from that data that Ganglia already captured. For measuring business processes, ActiveBPEL instrumentation-based sensors can be employed. For other custom data, custom sensors has to be developed with a help of provided examples. All new metrics has to be defined (registered) and aligned with all sensors and connectors.

Service related QoS terms can be implemented by a special sensor, but more preferably by derivation rules as a part of the service definition. Such rules define how new derived metrics (QoS term) are calculated from basic measured metrics.

SLA is monitored by Reasoning Engines that are instructed by monitoring plan provided by the Monitoring Manager component. If the default Reasoning Engine is not fully capable of reasoning on all SLOs (constraint), a new Reasoning Engine plus a Reasoning Component Gateway might be introduced.

Monitoring information is essential for all the parties involved in the SLA management. Therefore its presentation within a graphical user interface is a key feature for all the use cases. The data layer – which make use of the Low Level Monitoring System remote API - is common to all the use cases, while the presentation itself (GUI) is typically domain specific. Finally, all parties can subscribe to receive the notifications, violations and warnings they are interested in.

The essential prerequisite for the monitoring system is the presence a database for the storage of the historical info and for the landscape database(s).

### 5.2.7 Integration of a new Component into the Platform

In this section, we will explain how to integrate a generic component into the SLA@SOI platform. By generic component we mean a mavenized module endowed with both a Java interface and a Java implementation that either exposes some services to the other component of SLA@SOI component, or is meant to be a client of other components’ services, or even both.

The procedure to achieve the integration of a new component into the platform is made of the following steps:

1. Register and Expose the interface as an OSGI Service
   a. Spring Application Context
2. Inject the OSGi Service
   a. Service Reference Annotation
   b. Activator Bundle Context

3. Setup launch configuration

The first step (1.a) is to deploy a Spring Application Context that can be somehow recognized by an OSGi container. You have to register your module interface as an OSGi service to make it available to other bundles within the OSGi container. The Spring DM framework provides a suitable infrastructure to do all that.

In order to do this you have to create a folder named `resources\META-INF\spring` under the `src\main` folder of your mavenized module.

Inside the newly generated folder, you have to create a standard Application Context definition file (named, e.g., fooservice.xml). The content of such a file must reflect the one shown below. The important thing you have to take care of is the bean name, which is the name you choose for your interface and the implementation class.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   <bean name="fooService" class="org.springdm.impl.FooServiceImpl" />
</beans>
```

Figure 21: Sample Spring application context file

Step 1.b consist of creating another Application Context definition file. The name of such a file must follow the convention of being formed by appending the suffix `-osgi` to the name of the Spring Application Context file created at step 1.a, e.g. fooservice-osgi.xml. This file has to be created in the same folder as the precious one (resources/META-INF/spring). An important detail to pay attention to is the OSGI namespace which will allow Spring DM (the OSGIServiceFactoryBean in particular) to use Spring infrastructure to export and register your interface as an OSGi service within the SLA@SOI OSGi container. In this way it will be made available to other platform bundles that might not be Spring enabled. The `<service>` element is part of Spring-DM's configuration schema and simply tells Spring-DM to publish the bean identified by the `ref` attribute into the OSGi service registry under the interface specified by the `interface` attribute. We could have place both definitions into a single file, but separating them makes testing of POJO service functionality simpler, since it doesn't require any OSGi specific dependency and Spring DM testing infrastructure. Once our Spring-powered OSGi bundle is deployed into an OSGi container and brought up to the ACTIVE state, Spring DM Extender Bundle will recognize it and in the similar fashion as ContextLoaderListener loads Web Application Context, it will load Application Context defined in your bundle. A sample *-osgi.xml context file is shown in Figure 22.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:osgi="http://www.springframework.org/schema/osgi"
```
To summarize, at the end of this steps the resulting project hierarchy is the one shown in Figure 23.

Step 1.c consists of adding to your module’s pom file those declarations that will form the Manifest.mf of your OSGi bundle and will become the rule descriptor for your bundle at run time. Besides the simple <Bundle-Name> and <Bundle-SymbolicName> elements, the <Export-Package> instruction specifies what packages will be exported by your bundle, e.g. org.springdm.service package. The <Private-Package> instruction is definitely similar to the <Export-Package> instruction, except for the fact that these packages will not be exported by the bundle. In principle, the package that contain the interface definitions are exported and the package that contains the implementation classes are made private, e.g. in this case org.springdm.impl is declared as a private package.

The <Import-Package> instruction can be used to import other libraries that are available inside the runtime environment. For instance log4j libraries are the one that are mostly required by all the modules. Such a library can be imported as shown above. Notice that you also require to resolve this dependency at build time, and for this reason you must add the log4j dependency to the pom file as shown in Figure 25.
Another important aspect to be clarified is the difference between exported packages and export OSGi Services. Exported Packages are the third party libraries that are exported by the third party bundles. OSGI Service are the interfaces that are defined as bean by the OSGI bundle using Application and OSGI context file. At start-up they get registered inside the OSGI Registry which is based on publish/find/bind concept typical of SOA architectures. Thus, if a bundle is exposing a package, you can use this package by defining it inside your <Import-Package> instruction. On the other side, if a bundle is exposing an OSGI Service then, not only you have to import the package; you also need to inject the service. This is the exactly the subject topic of the following step.

As a matter of fact, step 2 is about the injection of an OSGI service from other SLA@SOI Platform component. For achieving this there are two possible alternative approaches, respectively Service Reference Annotation (previously mentioned as step 2.a) and Activator Bundle Context (previously mentioned as step 2.b).

Service Reference Annotation approach, corresponding to step 2.a, is based on the fact that Spring DM's annotation extension allows to pull in an OSGi service reference by annotating the setter of a given property. In addition to this, what the developer has to do is to configure a BeanPostProcessor which tells Spring how to handle your methods annotated with @ServiceReference. For instance, if you know there exist an OSGI Service by the name of FooService, in order to inject and use this service, you must first configure the BeanPostProcess by putting the definition that is visible in Figure 26 inside your application context file.

```xml
<bean
    class = "org.springframework.osgi.extensions.annotation.
            ServiceReferenceInjectionBeanPostProcessor"
/>  
```

**Figure 26: Configuration of the BeanPostProcess for OSGi service injection**

Then you have to add the dependency to the service and spring osgi annotation package. Moreover, you have to import the service interface package and the annotation package inside <Import-Package> instruction. As an example, if the interface is defined inside the org.springdm.service package of the SpringDMTestBundle, your pom file should contain the code snippet in Figure 27.

```xml
<dependency>
    <groupId>org.springframework.osgi</groupId>
    <artifactId>spring-osgi-annotation</artifactId>
    <version>1.2.0</version>
    <type>jar</type>
    <scope>provided</scope>
</dependency>
```

```xml
<dependency>
    <groupId>SpringDMTestBundle</groupId>
</dependency>
```

**Figure 27: Dependency for OSGi service injection**
Finally, you have to use the `@ServiceReference` annotation to inject the service inside your implementation. In Figure 28 the `FooService` is injected and then used inside the `start()` method.

```java
class SpringDMClient {
    protected FooService service;
    @ServiceReference  // obtained automatically by Spring-DM and // OSGi Container.
    public void setfooServices( FooService services ) {
        System.out.println("Registered OSGI FooService Found");
        this.service = services;
    }
    public void start() {
        service.foo();
    }
    public void stop() {} }
An alternative approach for the injection of OSGi service, corresponding to step 2.b., is applicable when you have defined an activator for your bundle. A bundle activator is a Java class that implements the `org.osgi.framework.BundleActivator` interface. The activator gets instantiated when the bundle is started. The activator class is the bundle's hook to the OSGi lifecycle layer for management. By implementing the `BundleActivator` interface, the OSGi framework invokes the `start()` and `stop()` methods to initialize or shutdown the bundle functionalities. The OSGi container calls the implemented `start()` and `stop()` methods when the bundle is started and stopped. The `start()` method registers the bundle component in the registry so that another bundle can use it. You also get a `BundleContext`, which allows the API to communicate with the container. The access to the OSGi framework occurs through the bundle context object supplied to the bundle activator. Bundles, then, register services or start processes using the lifecycle methods, `start(BundleContext)` and `stop(BundleContext)`. You can use the bundle context to register or look up services from the service registry. Thus, for instance, if you know that there exist an OSGi Service by the name of `FooService`, you can inject and use this service using bundle context as shown below.

```java
tracker = new ServiceTracker(context, FooService.class.getName(), null);
tracker.open();
FooService fooService = (FooService) tracker.waitForService(2000);
if (fooService!=null){
    System.out.println("Registered OSGi FooService Found ");
}
```

**Figure 29: OSGi service injection when using Activator Bundle Context**

For all this to work, it is required to add the spring context dependencies and the `<Import-Package>` instruction for corresponding packages inside the pom file. A sample pom file is shown below.

```xml
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>2.5.6</version>
</dependency>

<dependency>
    <groupId>org.apache.felix</groupId>
    <artifactId>org.osgi.core</artifactId>
    <version>1.2.0</version>
</dependency>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.felix</groupId>
            <artifactId>maven-bundle-plugin</artifactId>
            <version>${felix-version}</version>
            <extensions>true</extensions>
            <configuration>
                <instructions>
                    <Bundle-Name>YourBundleName</Bundle-Name>
                    <Bundle-SymbolicName>YourBundleName</Bundle-SymbolicName>
                    <Bundle-Activator>org.slasoi.test.TestActivator</Bundle-Activator>
                    <Import-Package>org.apache.log4j;resolution:=optional,
```
Step 3 of the procedure for integrating a generic component into the SLA@SOI platform, address the preparation of the new bundle for launching it inside the integrated SLA@SOI Platform.

First of all you must define a profile of your bundle. With this profile you will define the collection of bundles, if any, besides your own bundle that you would like Pax-runner to install and start when the platform starts. Each platform component must define its profile. Such a profile is referenced from within the main launch configuration file (namely, runner.args) using the --profile option. Profiles consists of a composite files in which all the required bundles are listed. The composite scanner process at start up will read the composite files and provisions in the target framework all those bundles that are declared. As a reference consider the composite file for the integration module whose location inside the source package is the following:

/profiles/integration/integration/1.0.0/integration-1.0.0.composite

After creating the composite for your bundle, put the entry for its profile inside the /pax-runner/runner.args. This will enable the launch of the new component inside the OSGi runtime that is started when giving the pax-run command.

5.3 Build Instructions

The goal of this section is to enable the user to build the SLA@SOI platform from scratch, that is starting from the download of the source code. Following this step-by-step procedure it will be possible also to clarify better the role of the different technologies used for building and running the SLA@SOI platform.

5.3.1 Requirements

The requirements for performing the following steps about building and executing the framework are the following:

- Java Runtime Environment 1.5
- Maven 2.2.1
- Pax-runner 1.3.0

The configuration of Maven constitutes a point of attention. The important sections to be changed in the Maven configuration file (settings.xml) are:

- <localRepository>: must point to the local maven repo;
- <server>: must contain the list of servers used for downloading artefacts;
- <proxy>: only if you are behind proxy, you must write the proxy configuration in this section.

The environment where the platform is to be built must be configured by setting a number of variables:
5.3.2 **Download**

The source code of the platform can be obtained either from the SVN repository 44[6] or from the FTP site where a source package is available [34]. The download of the platform comprises the ORC-data folder that hosts the templates and other configuration files related to ORC. This folder is referenced from within the source code of SLA@SOI Platform in order to load the ORC related configuration data.

5.3.3 **Configuration**

The configuration of the environment for building the SLA@SOI platform consist of setting the following environment variables:

- **SLASOI_HOME**: must point to the folder /common/osgi-config. The purpose of SLASOI_HOME variable is to keep all the configuration files required by different platform components in a common place. This allows for a complete portability of the platform code, in particular with respect to its integration within the SLA@SOI platform.
- **SLASOI_ORC_HOME**: must point to the folder ORC-data that was mentioned before. This variable is created with the goal of referencing ORC related data from the source code.

5.3.4 **Build**

The following command must be run from the command line at the root of the source code distribution. It will install all the platform components to the local Maven repository.

```bash
mvn clean install -fae -Dmaven.test.skip=true
```

The next necessary step in order to build all the components that are needed for running the tests is to compile the Integration test module. This can be done by positioning at location /IntegrationTestModule and running the command:

```bash
mvn clean install -fae -Dmaven.test.skip=true
```

5.3.5 **Run**

Before launching the platform some configuration can be made for deciding which test to run.

The file to be modified for such configuration purposes is /common/osgi-config/Integration/config.properties. By default this file is configured to run the platform against the Negotiation scenario of the ORC use case. If you want to run other scenarios and interactions you must set the relevant scenarioId and OpCode in the configuration file.

The launch can now take place and it is made of two executions. Firstly, the SyntaxConverter, which currently is available as a service that is external with respect to the platform must be run by changing the current directory to

- **M2_HOME**: must point to <mavenDist>/apache-maven-2.2.1
- **M2_REPO**: must point to <mavenDist>/apache-maven-2.2.1/repo
- **MAVEN_OPTS**: must be set to -Xms128m -Xmax1500m
/generic-slamanager/syntax-converter and running the Syntax Converter broker application with the SycBroker.bat script. Then the platform can be executed by changing directory to /pax-runner, renaming the file runner.args.integration into runner.args, and, finally, launching the SLA@SOI Y2 platform by simply typing the command:

pax-run

At this point, it will take some time before the entire SLA@SOI platform will be launched. The platform consist of more than 150 bundles that will get loaded after this command. In order to verify the completion of the launch of platform, after sometime you can type at the OSGi console the command ss. By this you can verify that all the bundles are in the ACTIVE state except for fragment bundles (which are not meant for ACTIVE state), org.ops4j.pax.configmanager (the logging control bundle) and the IntegrationTestModule bundle (the integration bundle).

In order to avoid detailed log prints during the execution of the tests, it is possible to optionally start the bundle org.ops4j.pax.configmanager by typing at the OSGi console the command:

start <n> (where <n> is the number corresponding to the org.ops4j.pax.configmanager bundle that appears in the ss console)

After some time the SLA@SOI platform is up and running waiting for requests to be fulfilled.

In order to start the execution of the Integration Test Scenario you have to start the bundle named IntegrationTestModule_1.0.0. Simply type

start <n> (where <n> is the number corresponding to the IntegrationTestModule bundle that appears in the ss console)

If you want to execute other tests you have to edit configuration file by selecting the scenario id and operation code that you would like to execute. By default the Negotiation Scenario (scenarioId=1) and the entire ORC Run (opCode=1600) are selected. After editing the configuration file, you have to save the configuration file with the scenarioId and opCode of your choice and then refresh the bundle name IntegrationTestModule_1.0.0 by typing at the OSGi console:

refresh <n> (where <n> is the number corresponding to the IntegrationTestModule bundle that appears in the ss console).
6 Conclusions

In this deliverable we have presented the integrated framework, developed during the second year of the SLA@SOI project, by describing the technological premises that have made it possible in terms of tools, the approach for the development and integration of the framework itself, and both a user’s perspective and a developer’s perspective on it.

In particular, we have described how the platform has been realized using the industry standard OSGi with the goal of pursuing the requirements of flexibility, configurability, and extensibility. We have showed how the main result of this effort is a platform that, with the help of the provided guidelines can be effectively adopted in different contexts both as a whole and as in terms of individual components. This deliverable also has reported the significant improvements that have been achieved during the second year of the project in terms of code quality, testing activities and documentation.

Along with this also other aspects related to the management of the delivery of the software, and related artefacts, have been showed as a report of the vast amount of activities performed, within the A1 workpackage until M26, around the assembly of the SLA@SOI framework.

7 References

[1] SLA@SOI project: SLA@SOI Glossary. Annex to deliverable D.A1a Framework architecture (full lifecycle). July 2010


[9] SLA@SOI project, “SLA@SOI Developer Trac”, https://trac-dev.sla_at_soi.fzi.de

[10] SLA@SOI project, “SLA@SOI Trac”, https://trac.sla_at_soi.fzi.de

To obtain the credentials for accessing the references available onto the http://slasoi.xlab.si server, please contact us.
Appendix A: Glossary

The following list shows the most important entries of the SLA@SOI glossary. Note that terms that are specific for the current document and not part of the overall project wide glossary are marked with an asterix *.

**Agreement Initiator**
An agreement initiator is a party to a service level agreement. The initiator creates and manages an agreement on the availability of a service on behalf of either the service customer or service provider, depending on the domain-specific signalling requirements.

**Agreement Offer**
An offer is the description of the agreement relationship that is sent from agreement initiator to agreement responder during agreement creation, indicating the relationship which the initiator would like to form.

**Agreement Responder**
The agreement responder is a party to a service level agreement. The responder implements and exposes an agreement on behalf of either the service provider or service customer, depending on the domain-specific signalling requirements.

**Agreement Template**
An agreement template is an XML document used by the agreement responder to advertise the types of offers it is willing to accept.

**Agreement Term**
Agreement terms define the content of a service level agreement.

**Business Service**
A business service is exposed/invoked via at least some non IT elements.

**Business Manager**
A specialization of service provider: person that defines the SLATs of products and joins available services in a product.

**External Service**
External services are exposed across the boundaries of an organization, i.e. across at least two administrative domains.

**Framework Administrator**
A specialization of service provider: person that configures/adapts the SLA@SOI framework for a specific application.

**Guarantee Term**
Guarantee terms define the assurance on service quality associated with the service described by the service definition terms. They refer to the service description that is the subject of the agreement and define service level objectives, qualifying conditions and business value expressing the importance of the service level objectives.

**Hybrid Service**
A hybrid service is a set or bundle of other services where all these services are exposed to the customer but have different service interface types (e.g. an IT service and a business service).
Infrastructure Manager
A specialization of infrastructure provider: person/system that is interested to measure and control infrastructure properties.

Infrastructure Provider
A specific kind of service provider that focuses on the provisioning of infrastructure services.

Infrastructure Service
An infrastructure service is a specific IT service which exposes resource/hardware-centric capabilities.

Internal Service
Internal services are exposed within the boundaries of an organization, i.e. within one administrative domain.

IT Service
An IT service is exposed/invoked by means of information technology. Specific classes of IT services may be software services, infrastructure services or media services.

Offered Service
An abstract service (more precisely: service type) which is offered by a specific Service Provider to its Service Customers.

Operation Level Agreements
A specification of the conditions under which an internal service or a component is to be used by its “customer”.

Service
A means of delivering value to customers by facilitating outcomes customers want to achieve without the ownership of specific costs and risks. See also service interface type, service concreteness, service exposure

Service Concreteness
The stage a service reaches over time from a fully abstract type to actually instantiated. See also service type, offered service, service implementation, service instance

Service Consumer
Person(s) who actually consume/use the provided services. Typically they belong to the service customer.

Service Customer
Someone (person or group) who orders/buys services and defines and agrees the service level targets.

Service Description Term
Service Description Terms describe the functionality that will be delivered under the service level agreement. The agreement description may include also other non-functional items referring to the service description terms.

Service Exposure
Services can be exposed either internally (within the same administrative domain) or externally. See also internal service, external service

Service Implementation
A service implementation is a possible concrete realization of a given service type.

Service Instance
A concrete realization of an offered service which is ready for consumption by service users. It relies on the instantiations of all the resources required for a given service implementation.
<table>
<thead>
<tr>
<th>Service Interface Type</th>
<th>Describes the nature of an actually exposed service, i.e. about the nature of his invocation interface. See also business service, IT service, hybrid service</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service Level Consequence</td>
<td>An action that takes place in the event that a service level objective is not met.</td>
</tr>
<tr>
<td>Service Level Agreement</td>
<td>An agreement defines a dynamically-established and dynamically managed relationship between parties. The object of this relationship is the delivery of a service by one of the parties within the context of the agreement. The management of this delivery is achieved by agreeing on the respective roles, rights and obligations of the parties. The agreement may specify not only functional properties for identification or creation of the service, but also non-functional properties of the service such as performance or availability. Entities can dynamically establish and manage agreements via Web service interfaces.</td>
</tr>
<tr>
<td>Service Level Objective</td>
<td>Service Level Objective represents the quality of service aspect of the agreement. Syntactically, it is an assertion over the agreement terms of the agreement as well as such qualities as date and time.</td>
</tr>
<tr>
<td>Service Provider</td>
<td>An organization supplying services to one or more internal customers or external customers.</td>
</tr>
<tr>
<td>SLA Manager</td>
<td>A specialization of service provider: person/system that is responsible for managing SLATs and SLA relationships.</td>
</tr>
<tr>
<td>Software Designer</td>
<td>A specialization of software provider: person that designs/develops the architecture and components of a specific SLA based application.</td>
</tr>
<tr>
<td>Software Manager</td>
<td>A specialization of service provider: person that defines software-based services, takes care of their management and supports the SLA manager in creating appropriate SLA templates.</td>
</tr>
<tr>
<td>Software Provider</td>
<td>An organization producing software components which might be used by a service provider to assemble actual services.</td>
</tr>
<tr>
<td>Software Service</td>
<td>A software service is a specific IT service which is exposed/invoked by means of software entities such as Web services, user interfaces, or software-based business processes.</td>
</tr>
<tr>
<td>Software Component</td>
<td>Software components are the entities produced at design-time by a software provider.</td>
</tr>
<tr>
<td>Service Type</td>
<td>A service type (or abstract service) specifies the external interface of a service possibly including non-functional aspects. It does not specify any means (components, resources) which are needed for the actual provisioning of that service.</td>
</tr>
</tbody>
</table>
## Appendix B: Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AOP</td>
<td>Aspect Oriented Programming</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>BM</td>
<td>Business Manager</td>
</tr>
<tr>
<td>B-SLAM</td>
<td>Business SLA Manager</td>
</tr>
<tr>
<td>CPD</td>
<td>Copy Paste Detector</td>
</tr>
<tr>
<td>EMF</td>
<td>Eclipse Modelling Framework</td>
</tr>
<tr>
<td>ERP</td>
<td>Enterprise Resource Planning</td>
</tr>
<tr>
<td>IE</td>
<td>Interaction Event</td>
</tr>
<tr>
<td>FCR</td>
<td>Finite capacity regions</td>
</tr>
<tr>
<td>FTP</td>
<td>File Transfer Protocol</td>
</tr>
<tr>
<td>G-SLAM</td>
<td>Generic SLA Manager</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>Infr-SLAM</td>
<td>Infrastructure SLA Manager</td>
</tr>
<tr>
<td>Infr-SM</td>
<td>Infrastructure Service Manager</td>
</tr>
<tr>
<td>IoC</td>
<td>Inversion of Control</td>
</tr>
<tr>
<td>JAR</td>
<td>Java ARchive</td>
</tr>
<tr>
<td>KPI</td>
<td>Key Performance Indicator</td>
</tr>
<tr>
<td>KVM</td>
<td>Kernel-based Virtual Machine</td>
</tr>
<tr>
<td>LLMS</td>
<td>Low Level Monitoring System</td>
</tr>
<tr>
<td>LQN</td>
<td>Layered Queueing Networks</td>
</tr>
<tr>
<td>MA</td>
<td>Manageability Agent</td>
</tr>
<tr>
<td>MRE</td>
<td>Monitoring Result Event</td>
</tr>
<tr>
<td>MVC</td>
<td>Model View Controller</td>
</tr>
<tr>
<td>NFP</td>
<td>Non-functional property</td>
</tr>
<tr>
<td>OCCI</td>
<td>Open Cloud Computing Interface</td>
</tr>
<tr>
<td>ORC</td>
<td>Open Reference Case</td>
</tr>
<tr>
<td>OSGi</td>
<td>Open Source Gateway initiative</td>
</tr>
<tr>
<td>OVF</td>
<td>Open Virtualization Format</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
</tr>
<tr>
<td>QPN</td>
<td>Queueing Petri Nets</td>
</tr>
<tr>
<td>PAC</td>
<td>Provisioning and Adjustment Component</td>
</tr>
<tr>
<td>PCM</td>
<td>Palladio Component Model</td>
</tr>
<tr>
<td>PMD</td>
<td>Programming Mistake Detector</td>
</tr>
<tr>
<td>POC</td>
<td>Planning and Optimization Component</td>
</tr>
<tr>
<td>POJO</td>
<td>Plain Old Java Objects</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
</tr>
</tbody>
</table>
SaaS  Software as a Service
SE   Service Evaluation
SLA  Service Level Agreement
SLAM SLA Manager
SLAT Service Level Agreement Template
SLO  Service Level Objective
SM   Service Manager
SME  Small and Medium-sized Enterprise
SOA  Service Oriented Architecture
SVN  Subversion
SW-SLAM Software SLA Manager
SW-SM Software Service Manager
TCO  Total Cost of Ownership
TOGAF The Open Group Architecture Framework
VM   Virtual Machine