Secure Provisioning of Cloud Services
based on SLA Management

SPECS Project - Deliverable 2.3.2

Reference Architecture for Cloud SLA Negotiation: Development and Tests – Final Prototype

Version no. 1.1
30 April 2016

The activities reported in this deliverable are partially supported
by the European Community's Seventh Framework Programme under grant agreement no. 610795.

Deliverable information

Deliverable no.:	D2.3.2
Deliverable title:	Reference Architecture for Cloud SLA Negotiation: Development and Tests – Final Prototype
Deliverable nature:	Prototype
Dissemination level:	Public
Contractual delivery:	30 April 2016
Actual delivery date:	30 April 2016
Author(s):	Madalina Erascu (IeAT)
Contributors:	Alessandra De Benedictis (CeRICT), Jolanda Modic (XLAB), Damjan Murn (XLAB), Massimiliano Rak (CeRICT), Adrian Spătaru (IeAT)
Reviewers:	Valentina Casola (CeRICT), Silvio La Porta (EMC)
Task contributing to the deliverable:	T2.3
Total number of pages:	38

Executive summary

The current deliverable is the second of the three deliverables (D2.3.1, D2.3.2, and D2.3.3) reporting the development activities that have been undertaken in the Negotiation module. The module is in charge of managing the negotiation and renegotiation phases of a Service Level Agreement (SLA) life-cycle. More precisely, it analyses users' security requirements, transforming them into a machine-readable language in order to be usable to the SPECS framework, as well as delivering the result of the negotiation/renegotiation back to the users.

As described in the previous deliverables, in order to cover the large number of requirements, the *Negotiation* module is composed by the three components: *SLO Manager, Supply Chain Manager*, and *Security Reasoner* and by a number of different artifacts, mainly conceptual models to cope with the security SLA structure.
The final description of the design of this module is reported in D2.3.3 that is released at M30, too. This deliverable is devoted to describe the final implementation details of the prototype; it reports (i) the final architecture of the Negotiation module (ii) the development activities related to T2.3 from M18 (release of D2.3.1) to M30 (release of this deliverable), (iii) references on how to install and use the prototype components developed in this task, and (iv) functional and performance test results, in accordance with the methodology respectively proposed in D4.5.2 and D1.5.2.

A preliminary prototype of the *SLO Manager* component was released at M18. The main functionality of the component, namely, translating the security requirements specified by the users into an SLA, did not change, but it was slightly adapted based on the other components' requirements.

The *Supply Chain Manager* component is first described in this deliverable. It orchestrates (with the support of the Enforcement module) the generation of supply chains to build proper SLA Offers for the users security requirements specified in a *SLA Template*.

The *Security Reasoner* component evaluates the security levels of different SLA Offers and compares them with the users security requirements. At M18, the component provided these functionalities, but as a standalone application. In this deliverable, we present it as a service, fully integrated in the Negotiation module, but offering specific APIs.

The components of the Negotiation module cover a very large number of requirements and are core components of the SPECS solution, although not transparent to the users. The components are available online.

Table of contents

Index of figures

Index of tables

1. Introduction

The Negotiation module prototype presented in this document is devoted to implement the SPECS *Negotiation* phase of an SLA life cycle. More precisely, we present the prototypes of the *SLO Manager, Supply Chain Manager* and *Security Reasoner* components of the *Negotiation* module.

The *SLO Manager* is the component that offers the negotiation API to the SPECS Application. It orchestrates the entire negotiation and renegotiation processes. It manages the creation of *SLA Templates*; it triggers generation of supply chains according to the End User's (EU) security requirements; and it invokes evaluation and ranking of the *SLA Offers* that are built according to the supply chains.

The *Supply Chain Manager* orchestrates (with the support of the Enforcement module) the generation of supply chains for the EU's security requirements specified in an *SLA Template*.

The *Security Reasoner* evaluates and ranks the *SLA Offers* created during the negotiation process. The evaluation is done by using security evaluation techniques that apply quantification algorithms to reason about the level of security provided by each of the SLA offers, with respect to the EU requirements.

The three prototypes are available on line and can be downloaded from the project repository: [1] (*SLO Manager*), [2] (*Supply Chain Manager*) and [3] (*Security Reasoner*).

In the reminder of this deliverable we discuss on the coverage of the negotiation requirements (Section 3.1) and for each component we will report about implementation, installation, usage and testing (both functional and performance evaluation), respectively in Section 3.2 (*SLO Manager*), Section 3.3 (*Supply Chain Manager*), and Section 3.4 (*Security Reasoner*). Regarding Security review and assessment, for the three components developed within Negotiation module, they have been reported in D1.5.2 (Appendix 7).

2. Relationship with other deliverables

The overview of the relationship of this deliverable with others from different WPs is presented in Figure 1. This deliverable is based on the existing work on architecture, requirements, use cases, etc., developed in the previous deliverables, but also constitutes an input for the deliverables finalizing the architecture, core modules and their integration, and use cases.

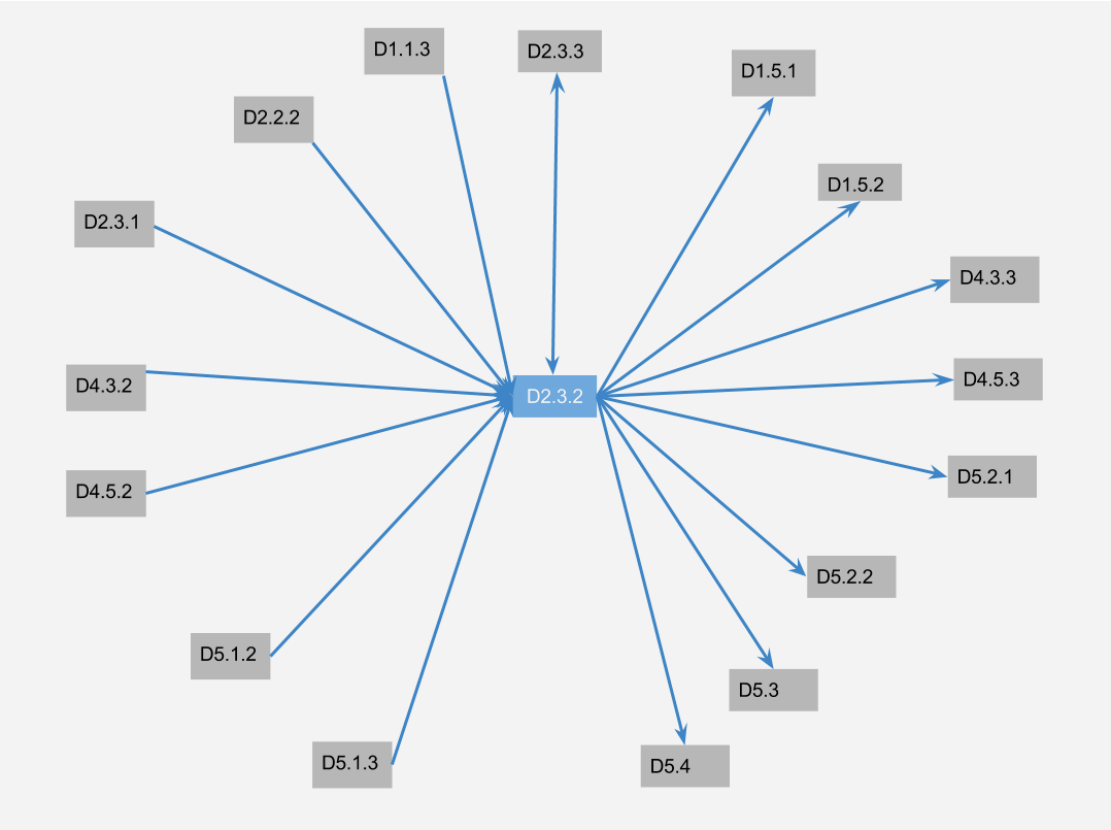

Figure 1. Relationship with other deliverables

The following deliverables are input for D2.3.2:

- D1.1.3: The final design of the SPECS solution provided input for the implementation of the Negotiation components.
- D2.2.2: The final report on the conceptual framework for Cloud SLA Negotiation is taken into consideration for the design and development in this deliverable.
- D2.3.1: The preliminary prototype of the Negotiation module is enhanced in the current deliverable.
- D4.3.3: During the SLA negotiation phase, the Enforcement module also supports the generation of valid supply chains according to EU security requirements, Cloud Service Providers (CSPs) capabilities, and possible SPECS security enhancements.
- D4.5.2: The verification and testing methodologies developed in this deliverable play an important role in the development of the prototype implementation of the Negotiation components.
- D5.1.2, D5.1.3: The use cases developed in these deliverables provided feedback for the implementation of the prototypes of the Negotiation module.

The following are the deliverables and WPs that take, as input, the results obtained in this deliverable:

- D1.5.1, D1.5.2: In the integration testing and examples, which will be presented in

these two deliverables, the Negotiation components are used.

- D5.2.1, D5.2.2, D5.3, D5.4: In the evaluation of the scenarios in these deliverables, the Negotiation components will be utilized.

Note that there is an input-output dependency between this deliverable and D2.3.3 where we present the report of the final architecture of the Negotiation module.

3. Service Level Agreement Negotiation Module

SPECS Negotiation implies an agreement on the security level of services requested by a Cloud Service Consumer (CSC), referred in the following as EU, and offered by the CSP, agreement finalized in a signed SLA. Furthermore, according to D1.1.1, D2.1.2 and D2.2.2, the negotiation module takes care of both *negotiation* and *renegotiation SLA phases*.

During the last year of the project, the negotiation and renegotiation process was finalized and its relationship with the Enforcement module and the application was clarified thanks to the definition of the supply chain generation method, offered by the Enforcement core components and the final set of APIs offered to the SPECS applications.

The final architecture of the Negotiation module is reported in Figure 3. Negotiation Module Architecture; we present components and the functionalities exposed to the other modules of the SPECS framework and the interactions among them.

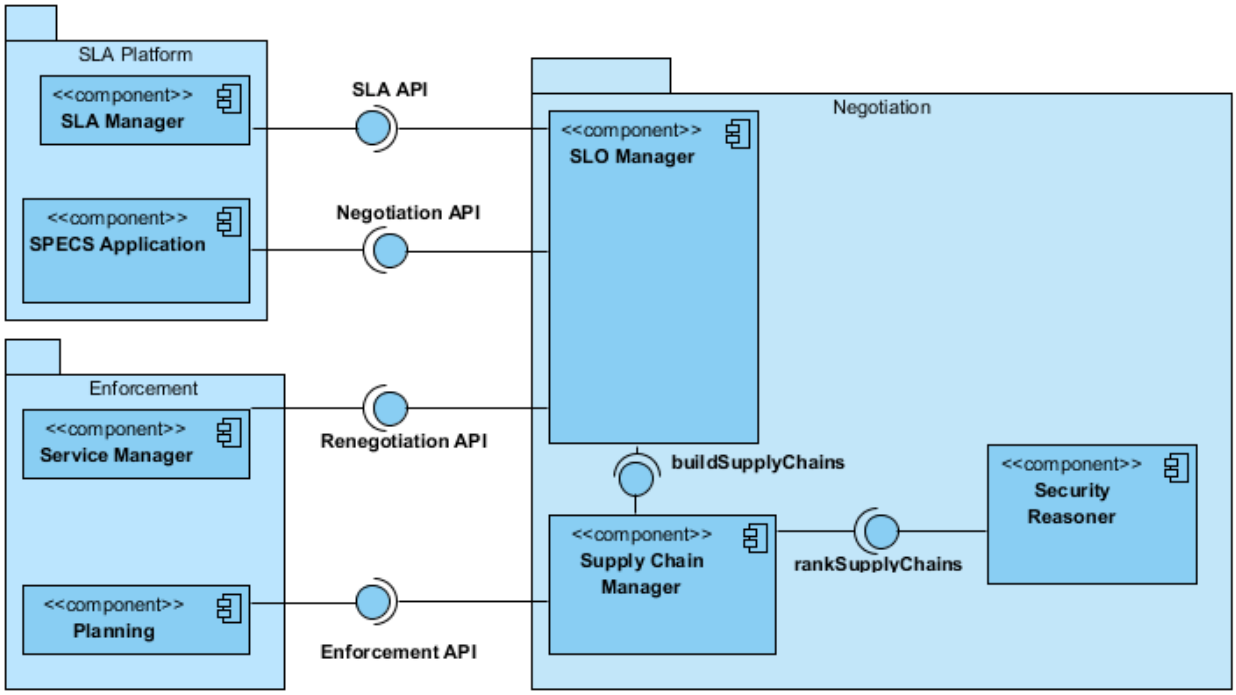

Figure 2. High-level Negotiation Module Architecture as of D2.2.2

In Figure 2 we report the preliminary architecture of the Negotiation module, as reported in D2.2.2, where (1) Negotiation API was an interface between the *SPECS Application* and *SLO Manager*, (2) Renegotiation API was an interface between *Service Manager* and *SLO Manager,* (3) *Service Manager* was a component of the Enforcement module.
As we mentioned above, the Negotiation API (see D2.3.3 Annex A) serves both negotiation and renegotiation activities, the only difference is in the flow in which methods are invoked.
In D2.3.3 (Section 3) we present a detailed flow of the negotiation and the two types of renegotiation in SPECS (EU triggered renegotiation process and CSP triggered renegotiation process). In order to better address these aspects, in the final Negotiation architecture (Figure 3) Negotiation and Renegotiation methods are offered by the same API (the Negotiation API) and the Service Manager component does not communicate with the Negotiation module anymore.

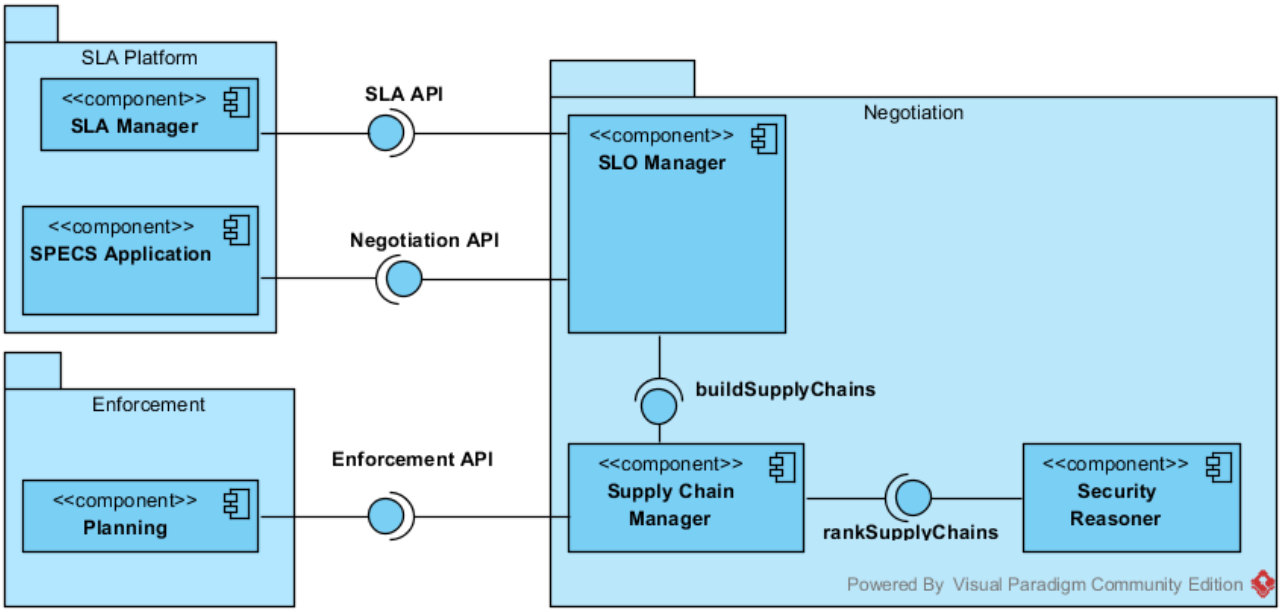

Figure 3. Negotiation Module Architecture

This deliverable focuses on the installation, usage and tests (Functional and Performance Tests) of the components developed in the framework.

As already said, Security review and assessment for the three components developed within Negotiation module are reported in D1.5.2 (Appendix 7).

3.1 Status of development activities

In Table 1 we schematically report the list of SPECS software components associated with the Negotiation module, together with the requirements (as stated in D2.2.2) that they, respectively, cover.

Negotiation Module	**SPECS Software Components**		
Requirements	***SLO Manager***	***Supply Chain Manager***	***Security Reasoner***
SLANEG_R1	X		
SLANEG_R2	X		
SLANEG_R3	X		
SLANEG_R4	X		
SLANEG_R5		X	
SLANEG_R6	X		
SLANEG_R7			X
SLANEG_R8			X
SLANEG_R9			X
SLANEG_R10	X	X	
SLANEG_R11			X
SLANEG_R12	X		
SLANEG_R13	X		
SLANEG_R16	X		
SLANEG_R17	X		
SLANEG_R18 – R29	Covered by the SLA models		

SLANEG_R30 – R31	Covered by the Enforcement module		
SLANEG_R32 – R33	Covered by the SLA Platform module		
SLAPL_R10		X	
SLAPL_R14			X
SLAPL_R21		X	
SLAPL_R33	X		
ENF_PLAN_R1		X	
ENF_PLAN_R12		X	

Table 1. SPECS Components related to the Negotiation module and related requirements

In the previous version of this deliverable we reported the coverage of 17 out of 33 requirements. As discussed in D2.3.3, in the current implementation, available artifacts and models cover all elicited requirements. In particular, requirements from SLANEG_R18 to SLANEG_R29 are covered by the SLA models, as reported in D2.3.3; SLANEG_R30 and SLANEG_R31 are covered by the Enforcement module (RDS component); SLANEG_R32 and SLANEG_R33 are covered by the SLA Platform module.
All components are available for download from the SPECS repository.

3.2 SLO Manager

SLO Manager is the main component of the Negotiation module, representing the interface for the negotiation and renegotiation of Security SLAs. It exposes a REST API for interaction (see D2.3.3 Annex A) with other components and modules of the SPECS solution (see D2.3.3 for more details). An EU *negotiates* her/his security requirements through an *SLA Template*. An *SLA Template* contains EU security requirements, the services and their security levels which are subject to negotiation. After the requirements have been set, a list of *SLA Offers* is generated, using the information provided by the Planning component of the Enforcement module, via the *Supply Chain Manager* component. An *SLA Offer* corresponds to a supply chain and each supply chain is composed of one CSP and a set of resources (e.g. virtual machines) enriched with security mechanisms enforcing and monitoring EU's chosen security features. Hence, one could see *SLA Offers* as instances of *SLA Templates.* The supply chains are created by SPECS according to the available security mechanisms (either offered by SPECS or provided by external CSPs) and security requirements provided by the EU. Before each *SLA Offer* is proposed to the EU, it is validated by the CSP in order to guarantee for example that the services and their security levels can be actually met (valid *SLA Offer*). Valid SLA Offers are then ranked according to the EU's requirements by applying the reasoning algorithms that perform comparisons and evaluations to determine what are the valid *SLA Offers* that better match EU requirements.
SLO Manager is available for download at [1].

3.2.1 Installation

In this section, we report the installation guide, which covers two scenarios:

- Installing by using precompiled binaries (SPECS recommended);
- Compiling and installing from source (for advanced users).

Install using precompiled binaries

The precompiled binaries are available under the SPECS Artifact Repository[1]

[1] http://ftp.specs-project.eu/public/artifacts/

Prerequisites:

- Oracle Java JDK 7;
- MongoDB;
- Java Servlet/Web Container (recommended: Apache Tomcat 7)
- Download the web application archive (war) from the artifact repository[2]

Installation steps:

- The application must be deployed to the Application Server (Tomcat) If Apache Tomcat 7.0.x is used, the war file needs to be copied into the "/webapps" folder inside the home directory (CATALINA_HOME) of Apache Tomcat 7.0.x.
- MongoDB must be up and running, listening on the default port of 27017.

Install from source

Prerequisites:

- Git client
- Apache Maven
- Maven dependencies (not available in the maven repo, have to be downloaded and installed):
  - specs-data-model[3]
  - specs-negotiation-supply-chain-manager[4]
- Apache Tomcat
- MongoDB
- SPECS components:
  - planning-api[5]
  - service-manager-api[6]
  - sla-manager-api[7]
  - security-reasoner-api [3]

Downloading the source code:

```
git clone https://bitbucket.org/specs-team/specs-core-negotiation-slomanager.git
```

Database configuration:

- During the application start up MongoDB must be up and running, accepting traffic on the default port 27017.

How to run tests:

- change directory in the root of the project
- run `mvn test`

Deployment instructions:

1. *Manual deployment to Tomcat:*

[2] http://ftp.specs-project.eu/public/artifacts/sla-negotiation/slo-manager/slomanager-api-0.1-SNAPSHOT.war
[3] https://bitbucket.org/specs-team/specs-utility-data-model.git
[4] https://bitbucket.org/specs-team/specs-core-negotiation-supply_chain_manager
[5] https://bitbucket.org/specs-team/specs-enforcement-planning
[6] https://bitbucket.org/specs-team/specs-sla_platform-service_manager-services_api
[7] https://bitbucket.org/specs-team/specs-sla_platform-sla_manager-sla-api

a. change directory to the root of the project
b. run `mvn package`
c. in directory `target` you will find `slomanager-api.war`. Deploy it to your Application Server by copying it into the "/webapps" folder inside the home directory for Apache Tomcat 7.0.x (the one pointed to by the `CATALINA_HOME` variable.

2. *Maven Tomcat deployer:*
   a. Create a user as script manager in `tomcat-users.xml`:
      `<user name = "deploy" password="s3cret" roles="manager-script">`
   b. change directory to the root of the project
   c. run `mvn tomcat7:deploy`
   d. the application will be deployed on the context path `/slomanager-api`

3.2.2 Usage

The *SLO Manager* component is accessible via a REST APIs interface. The development of RESTful Web Services is done in *Spring Boot Framework* [4], more specifically the `spring-boot-starter-web and spring-boot-data-mongo` artifacts for maintaining resources' persistence using a MongoDB [5] database.
All REST operations over resources are mapped under the path `/sla-negotiation/*`. They are represented by specific Java classes under the packages `eu.specs.datamodel.sla and eu.specs.datamodel.agreement`, inside the repository `specs-utility-datamodel`[3].

In Figure 4, we give an example of REST call to retrieve the *SLA Template* collection. There are two *SLA Templates* available, NIST [6], respectively CCM [7] compliant.

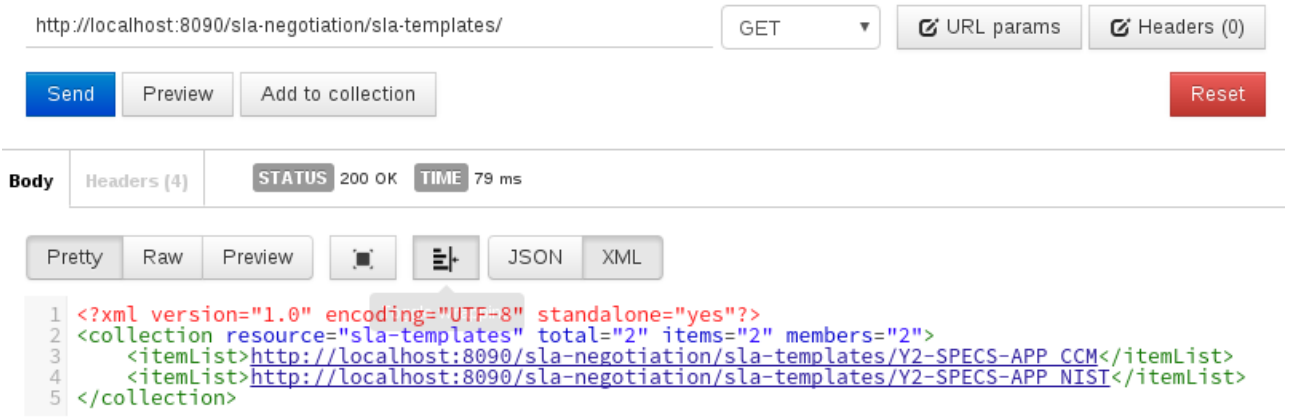

Figure 4. Retrieval of *SLA Templates* Collection

3.2.3 Functional Tests

In order to test the functionality of the *SLO Manager* REST API and the correctness of the underlying methods and procedures, JUnit [8] tests have been implemented (see Appendix 1. Negotiation Module – Functional Tests SLO Manager Tests). Since the component leverages other SPECS components and depends on the result of their actions in the negotiation process, mocking techniques (WireMock [9]) have been used to mimic their expected behaviour during unit and component tests.

SonarQube [2] has been used to assess quality of software, code coverage and conditions during tests. The results can be accessed online at [10]. Figure 5 gives an overview of the

code, such as number of lines of code, files and functions, and details about the complexity of the code. Figure 6 provides quality analysis information for the component, with regard to maintenance of the code, coding standards and the amount of code that has been tested. The *major* issue, shown in Figure 6, means that the main class has not been covered by tests; this is because it contains code that cannot be tested statically. This part of the code has been covered by integration tests in Section 4 of D1.5.2. Besides this issue, all written tests have successfully passed, obtaining a coverage of over 90% of the application. There is *no critical* issue meaning that no bug or security flow is present in the code. Since the Technical Debt Ratio is 0.1% (see http://docs.sonarqube.org/display/PLUG/SQALE+Plugin for an explanation how this number is computed), SonarQube decides that SQALE Rating is *A*, hence the *SLO Manager* component ready to be deployed and used by other SPECS components in its current state.

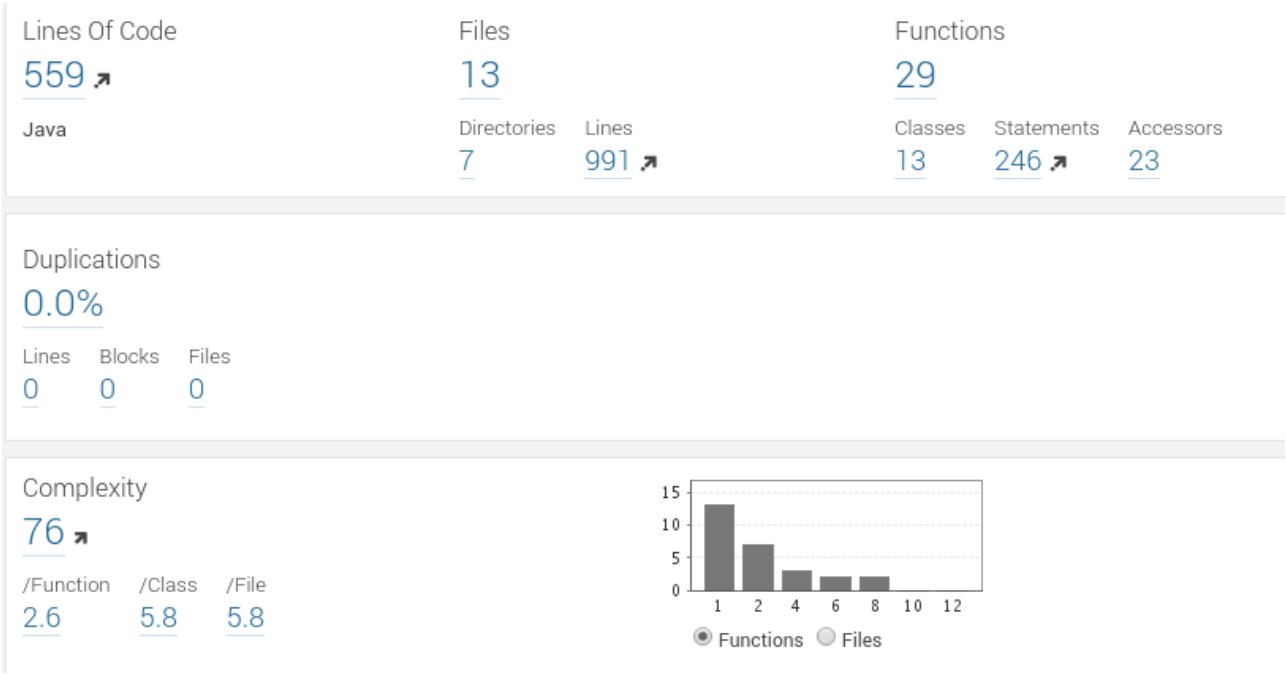

Figure 5. Code Properties for *SLO Manager* Component

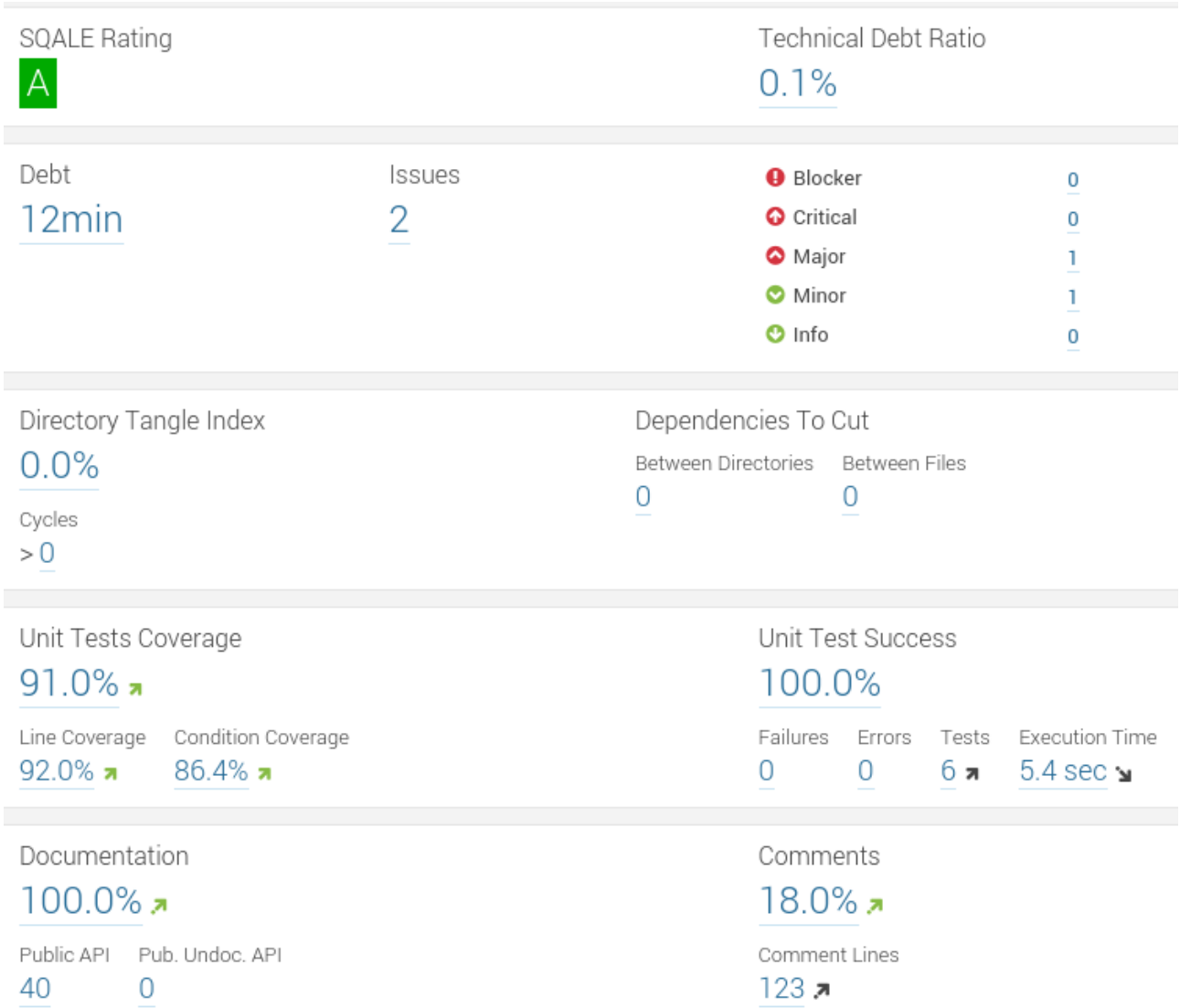

Figure 6. Quality Analysis for *SLO Manager* Component

3.2.4 Performance Tests

The performance and scalability of the *SLO Manager* was evaluated according to the methodology presented in D1.5.2 and adopted at project level. Two indicators are considered:

- *response time* (the time elapsed between the request of a service up to the production of the result), and
- *throughput* (the number of services executed per second).

These indices were evaluated based on a set of increasing workloads and the testing was conducted in the testing environment discussed in D1.5.2 and D1.6.2.
We prepared three *user profiles* for the SLO Manager component (see Table 2).

User profile	Description	Scripts
Create *SLATemplate*	Create a new *SLATemplate*	`SLATemplatePostAtOnce.scala`
Get *SLATemplate*	Retrieve the list of *SLATemplates* get one of them and store it	`SLATemplateGetAtOnce.scala`
Start Negotiation	Retrieve the list of *SLATemplates*, start negotiation for one of them	`StartNegotiationAtOnce.scala`

Table 2. *SLO Manager* User Profiles for Performance Tests

The component was stressed by issuing an increasing number of requests targeted to the exposed API calls endpoints involved in the user profiles described above.

Input Rate (users / s)	Throughput (req / s)	Response Time (ms)
0	0	0
1	2.44	6
100	108,25	196
200	142,44	394
500	228,53	1932
750	245,37	4048
1000	267,53	8235

Table 3. *SLO Manager* **Performance Tests. Get** *SLA Template* ***(a)***

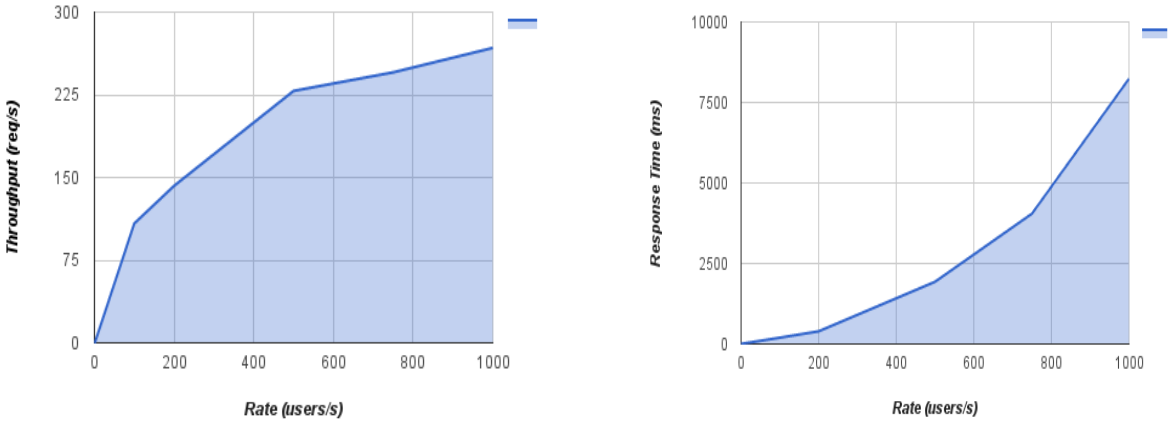

Figure 7. ***SLO Manager*** **Performance Tests. Get** ***SLA Template (b)***

Input Rate (users / s)	Throughput (req / s)	Response Time (ms)
0	0	0
1	1,217	50
100	97,523	203
200	116,986	96
500	149,022	312
750	206,157	433
1000	194,32	771

Table 4. ***SLO Manager*** **Performance Tests. Create** ***SLA Template*** **(a)**

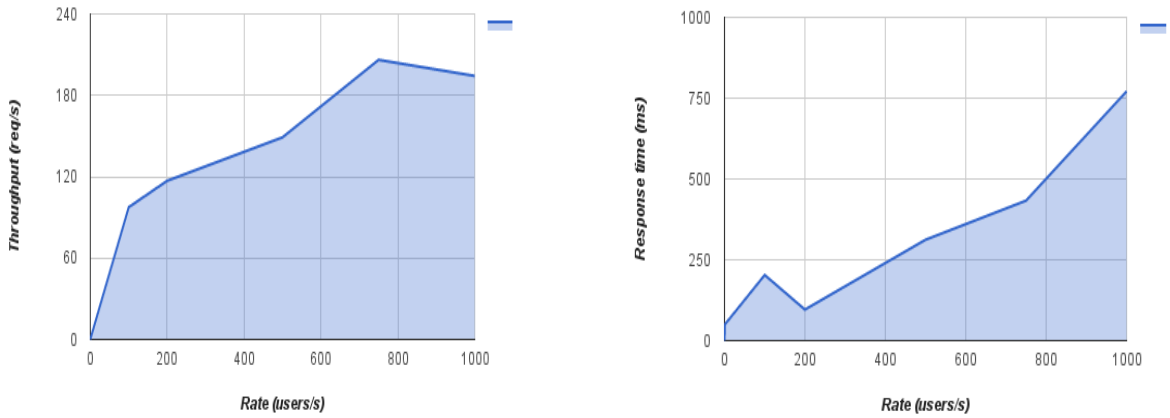

Figure 8. *SLO Manager* Performance Tests. Create *SLA Template* (b)

Input Rate (users / s)	Throughput (req / s)	Response Time (ms)	Failed (%)
0	0	0	0
1	4.53	29	0
10	1.38	68	0
100	48,888	877	0
150	52,845	364	0
200	68,347	505	0
300	76,6	346	0
500	111,682	408	0
750	111,757	2225	0
900	94,972	8922	2

Table 5. *SLO Manager* Performance Tests. *Start Negotiation* (a)

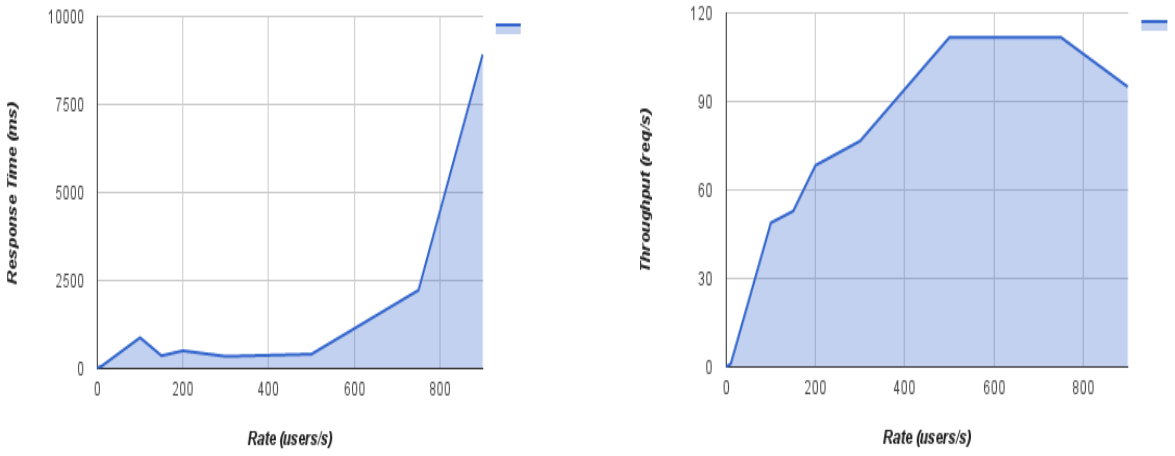

Figure 9. *SLO Manager* Performance Tests. *Start Negotiation (b)*

3.3 Supply Chain Manager

The *Supply Chain Manager* component orchestrates (with the support of the Enforcement module) the generation of supply chains for the EU's security requirements specified in an SLA Template. As discussed in deliverable D2.3.3, the *Supply Chain Manager* component (i) takes the *SLA Template* received from the *SLO Manager*, (ii) parses it to prepare the input for the Enforcement module, (iii) triggers the generation of supply chains, and returns a list of IDs of the generated supply chains.
The prototype of the component is available on Bitbucket [2]. In the following subsections, we report the installation and usage guideline, and tests preformed for the component.
Note that the *Supply Chain Manager* is not a web application (with a defined REST API), but just a Java library, directly and internally invoked by the SLO Manager. For this reason, we do not report in this deliverable Performance Tests for the *Supply Chain Manager* component since it is implemented as a Java library acting as an interface between the *SLO Manager* and the *Planning* componnent (responsible with the generation of supply chains) of the Enforcement module. We direct the reader to WP4 deliverables for Performance Tests for the Planning component.

3.3.1 Installation

Prerequisites:

- Java 7

The *Supply Chain Manager* is a Java library which can be included in the Maven-based project as a dependency:

```
<dependency>
   <groupId>eu.specs-project.core.negotiation</groupId>
   <artifactId>supply-chain-manager</artifactId>
   <version>0.1-SNAPSHOT</version>
</dependency>
```

In a non-Maven project, the *Supply Chain Manager* can be downloaded manually from the SPECS Maven repository and included in the project's classpath:

```
https://nexus.services.ieat.ro/nexus/content/repositories/specs-
snapshots/eu/specs-project/core/negotiation/supply-chain-manager/
```

To build the *Supply Chain Manager* component from the source code, the Apache Maven 3 tool is needed. First, clone the project from the Bitbucket repository using a Git client:

```
git clone https://bitbucket.org/specs-team/specs-core-negotiation-
supply_chain_manager
```

Then, go into the cloned directory and run:

```
mvn package
```

3.3.2 Usage

The *Supply Chain Manager* is implemented as a Java library and is packed as a Java archive (JAR) file, which has to be added to the Java classpath. The *Supply Chain Manager* provides the following Java API:

```
SupplyChainManager(String serviceManagerApiAddress, String planningApiAddress)
```

The *Supply Chain Manager* constructor accepts two parameters: address of the *Service Manager* and address of the Planning component.

```
List<SupplyChain> buildSupplyChains(AgreementOffer agreementOffer) throws SupplyChainManagerException
```

The method buildSupplyChains accepts one parameter: *SLA Template*, which is of type `AgreementOffer`. The method returns a list of built supply chains. In case anything goes wrong, the method throws `SupplyChainManagerException`.

3.3.3 Functional Tests

In order to verify the correctnes of the behaviour of the *Supply Chain Manager* component, a JUnit [8] test has been implemented. Dependencies on other SPECS components (the *Service Manager* component of the *SLA Platform* and the Planning component of the Enforcement module) have been mocked with the WireMock farmework [9]. For the code quality assessment, SonarQube [2] has been used.

The unit tests are described in Appendix 1, whilst the code quality report is depicted in Figure 10 and Figure 11. Note that there are *no critical nor major* errors which determines a low rate of *Technical Debt Ratio* (0.4%), hence *Supply Chain Manager* component, by a SQALE rating of A, is ready to be deployed and used by other SPECS components in its current state (see Figure 11).

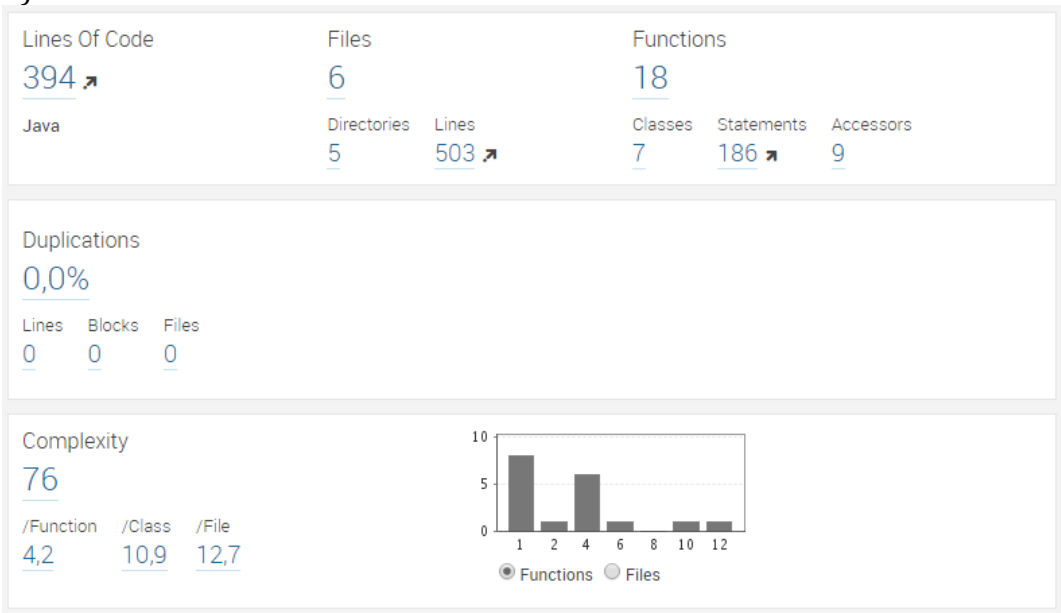

Figure 10. Code quality report - *Supply Chain Manager*, part 1

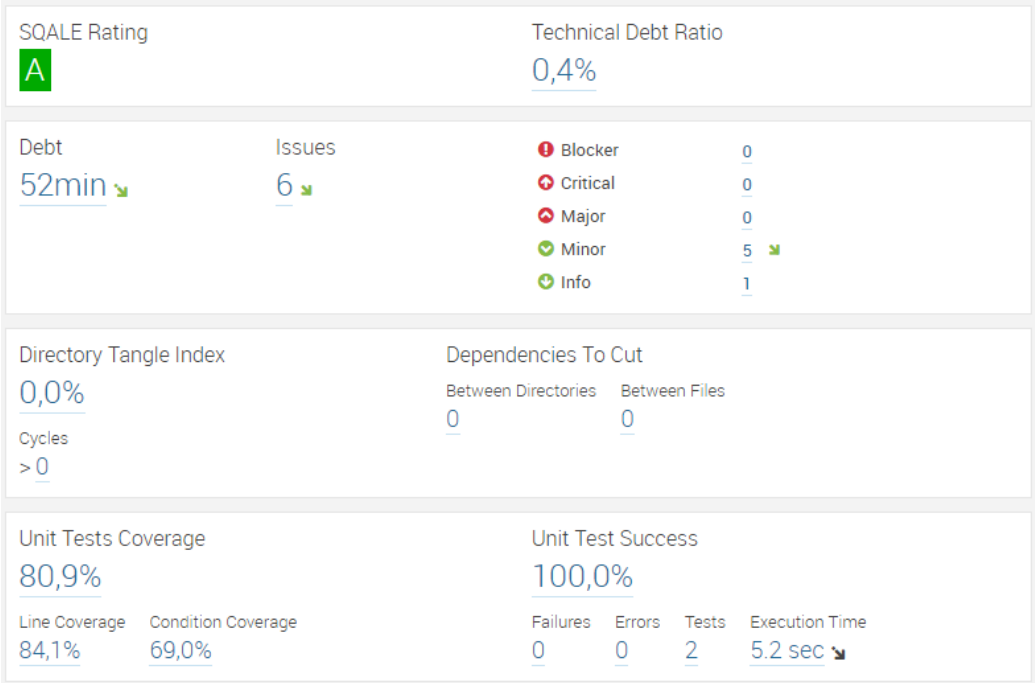

Figure 11. Code quality report - *Supply Chain Manager*, part 2

3.4 Security Reasoner

The *Security Reasoner* is invoked by *the Supply Chain Manager* to rank *SLA Offers*. Moreover, its functions may be invoked to help the SPECS Owner evaluate and compare available CSPs, independently of the SPECS flow.
The prototype of the component is available on Bitbucket at [3]. In the following subsections, we report the installation and usage guidelines, along with the results of the tests performed for the component.

3.4.1 Installation

In this section, we report the installation guide, which covers two scenarios:

- Installing by using precompiled binaries (SPECS recommended);
- Compiling and installing from source (for advanced users).

Installing by using precompiled binaries

The precompiled binaries are available under the SPECS Artifact Repository [3].
Requirements

- Oracle Java JDK 7;
- Java Servlet/Web Container (recommended: Apache Tomcat 7)

Installation steps

1. Download the web application archive (war) from the artifact repository [3];
2. Deploy the war in the java servlet/web container. If Apache Tomcat 7.0.x is used, the war file needs to be copied into the "/webapps" folder inside the home directory (CATALINA_HOME) of Apache Tomcat 7.0.x.

Compiling and installing from source

In order to compile and install the *Security Reasoner*, the following requirements must be satisfied:

Requirements

- a Git client
- Apache Maven 3.3.x
- Oracle Java JDK 7
- Apache Tomcat 7

Installation steps:

1. clone the Bitbucket repository with the following command:
   `git clone https://bitbucket.org/specs-team/specs-core-negotiation-securityreasoner.git`
2. change directory to the root of the project.
3. Under `specs-core-negotiation-securityreasoner` run:
   `mvn install`

The installation generates a web application archive (war) file, under the "/target" subfolder. In order to use the component, the war file has to be deployed in the java servlet/web container. If Apache Tomcat 7.0.x is used, the war file needs to be copied into the "/webapps" folder inside the home directory (CATALINA_HOME) of Apache Tomcat 7.0.x.

3.4.2 Usage

In deliverable D2.3.1, we illustrated how to use the *Security Reasoner standalone application* via its web interface. As discussed in D2.3.3, the Security Reasoner is now available as a component exposing a REST API (i.e., the *Evaluation API*). Therefore, the functionalities discussed in D2.3.1 (Single Evaluation, Comparison Evaluation, Upload the CAIQ template, Upload/Edit weights) are now available as REST API calls. A detailed description of API calls is reported in Annex A of D2.3.3; in the following, we discuss only how to use the *Security Reasoner* in the SPECS Negotiation flow.
With regards to the Negotiation flow, the *Security Reasoner* allows to:

- upload an *SLA Offer* and obtain an updated CAIQ for the involved provider, including all the security controls offered by SPECS.
  This is done by accessing the resource:
  /specs-app-SecurityReasoner/slacaiqs, with the POST HTTP method, by giving, as an input, an SLA Offer in the SPECS SLA machine readable format. The call returns the URI of the updated CAIQ.

- retrieve the score of an *SLA Offer*.
  This is done by accessing the resource:
  /specs-app-SecurityReasoner/slacaiqs/{id-slacaiq}/score, with the GET HTTP method. No request body is specified and the call returns the score associated with the root of the specified SLACaiq. If a *category* parameter is specified in the query string, the score of the requested category is returned.
  Note that this call must follow the previous one. The two calls together allow a set of ranked *SLA Offers* to be obtained, from which the EU can make a selection.

3.4.3 Functional Tests

In order to test the functionality of the *Security Reasoner* and the correctness of the underlying methods and procedures, JUnit tests have been implemented. The component does not depend directly on other SPECS components, therefore it was not necessary to use mock-ups.
SonarQube was used to assess the quality of software, code coverage and conditions during tests. The unit tests are described in Appendix 1, whilst the code quality report is available at [11] and summarized in Figure 12 and Figure 13.

The code quality report in Figure 13 states that there is one *critical* issue, related to a static variable not declared as final. Moreover, there are still some *major* issues. Since the new version of the Security Reasoner has been derived from the standalone application, which was developed as a proof-of-concept application, the presence of these issues is reasonable. However, the detailed report, available at [11], shows that most of these issues are related to duplicate code blocks and imperfect use of counters. Hence the SQALE Rating is A also for this component.

Figure 12. Code quality report – Security Reasoner, part 1

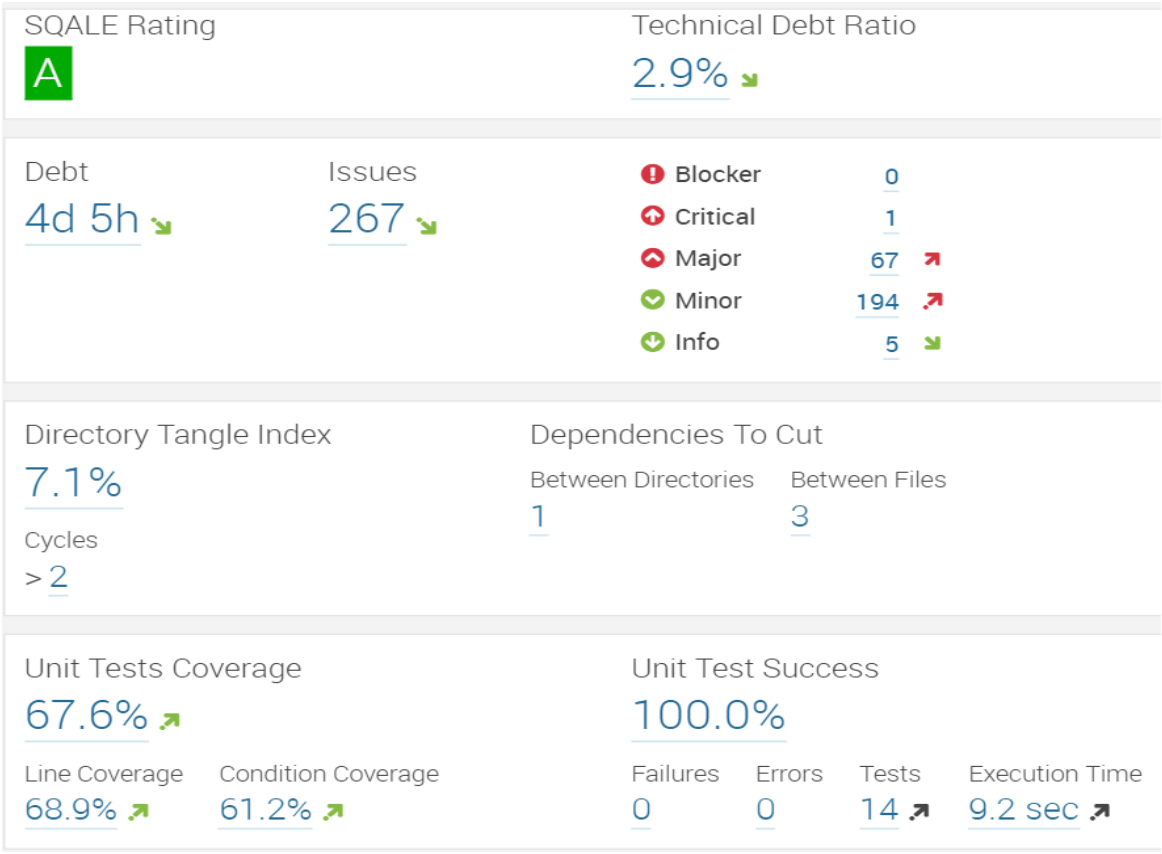

Figure 13. Code quality report – Security Reasoner, part 2

3.4.4 Performance Tests

In order to evaluate the performance and scalability of the *Security Reasoner* according to the approach defined in deliverable D1.5.2, we considered the two performance indices *response time* (the time elapsed between the request of a service up to the production of the result) and *throughput* (the number of services executed per second). These indices were evaluated based on a set of increasing workloads and the testing was conducted in the testing environment discussed in D1.5.2 and D1.6.2.

The component was stressed by issuing an increasing number of contemporary requests targeted to the exposed API calls' endpoints. In particular, we tested the following calls for a number of requests (i.e., contemporary users) up to 40.

- `/specs-app-SecurityReasoner/caiqs` (POST): used to create a new CAIQ resource;
- `/specs-app-SecurityReasoner/caiqs` (GET): used to retrieve all stored CAIQs;
- `/specs-app-SecurityReasoner/judgements` (POST): used to upload a new judgement resource;
- `/specs-app-SecurityReasoner/judgements/{judgement-id}` (GET): used to retrieve a judgement by its id;
- `/specs-app-SecurityReasoner/caiqs/{caiq-id}/evaluate` (GET): used to evaluate a CAIQ based on a judgement and create a tree-like structure.

Results are reported in the following tables and graphs.

Input Rate (users / s)	Throughput (req / s)	Response Time (ms)
0	0	0
1	0,742	36717
2	1,083	55414
3	1,619	46984
4	2,066	41742
5	2,683	46443

Table 6. Security Reasoner POST CAIQ call performance (1)

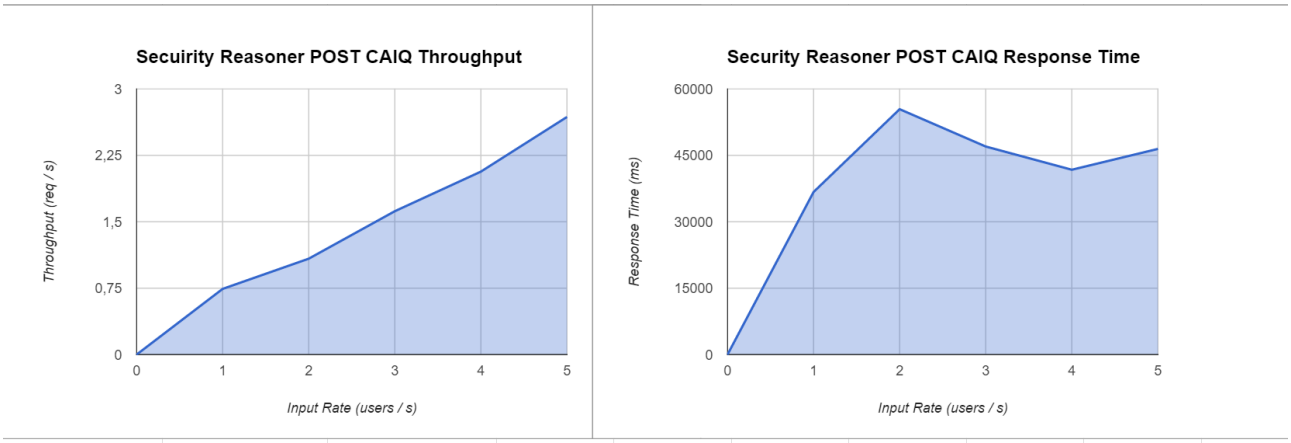

Figure 14. Security Reasoner POST CAIQ call performance (2)

Input Rate (users / s)	Throughput (req / s)	Response Time (ms)
0	0	0
5	4,99	369
20	19,951	338
50	33,842	6247
70	42,279	25744
100	66,508	30991

Table 7. Security Reasoner GET all CAIQs call performance (1)

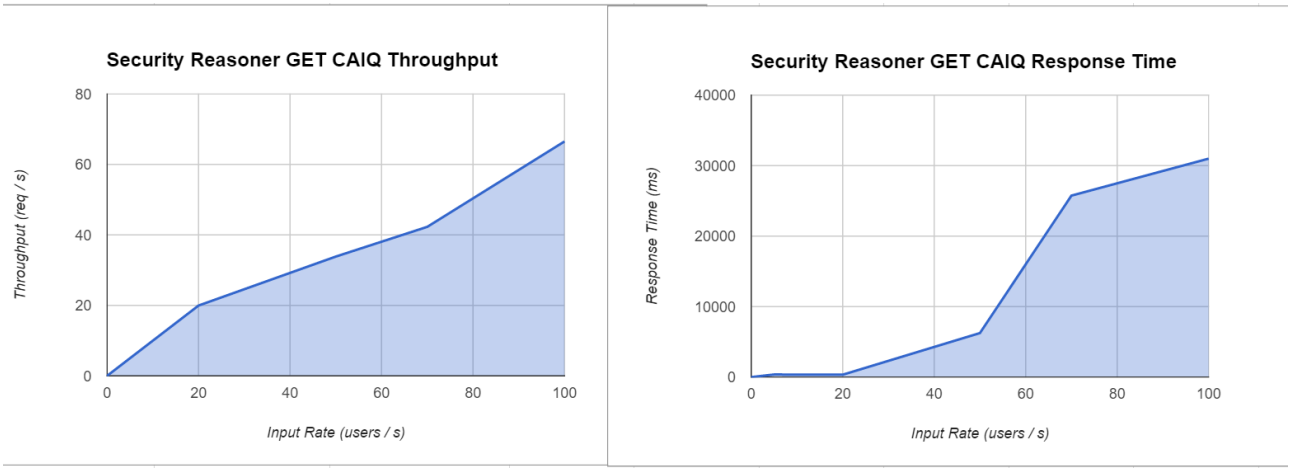

Figure 15. Security Reasoner GET all CAIQs call performance (2)

Input Rate (users / s)	Throughput (req / s)	Response Time (ms)
0	0	0
1	1,002	768
2	1,995	799
3	2,251	17934
4	2,223	44776
5	2,848	55944

Table 8. Security Reasoner POST Judgement performance (1)

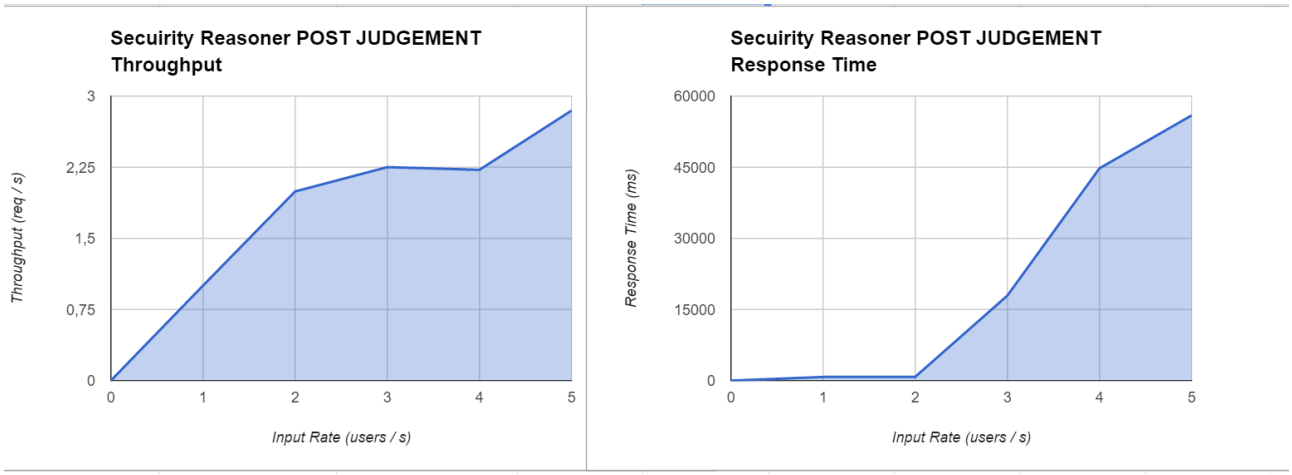

Figure 16. Security Reasoner POST Judgement performance (2)

Input Rate (users / s)	Throughput (req / s)	Response Time (ms)
0	0	0
5	8,744	482
10	15,274	570
20	27,749	589
30	25,706	26213
40	39,934	28583

Table 9. Security Reasoner GET judgement performance (1)

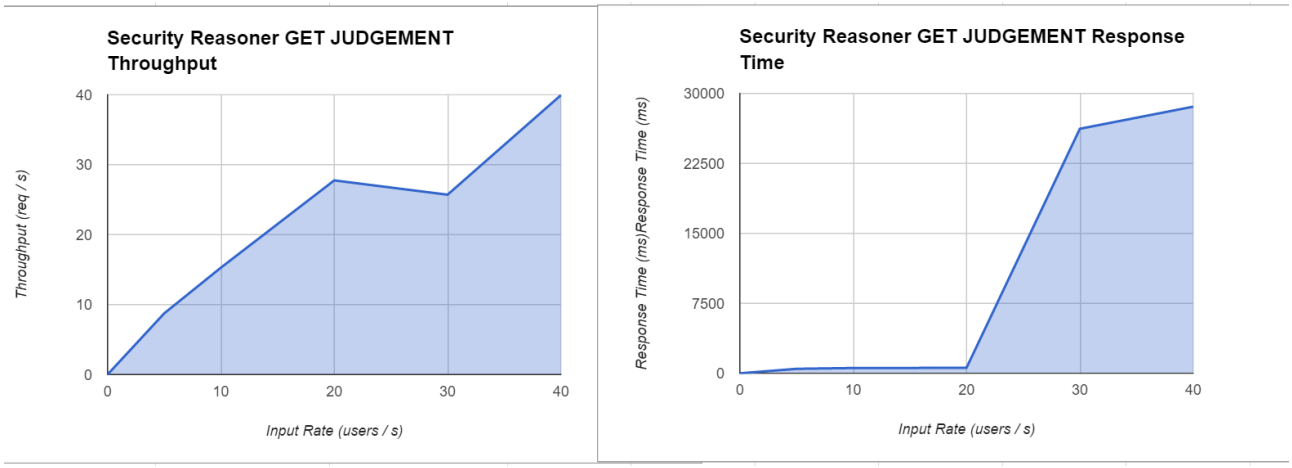

Figure 17. Security Reasoner GET judgement performance (2)

Input Rate (users / s)	Throughput (req / s)	Response Time (ms)
0	0	0
1	4	286
5	19,801	398
10	31,367	460
20	27,661	1747
40	48,126	5766

Table 10. Security Reasoner Evaluate performance (1)

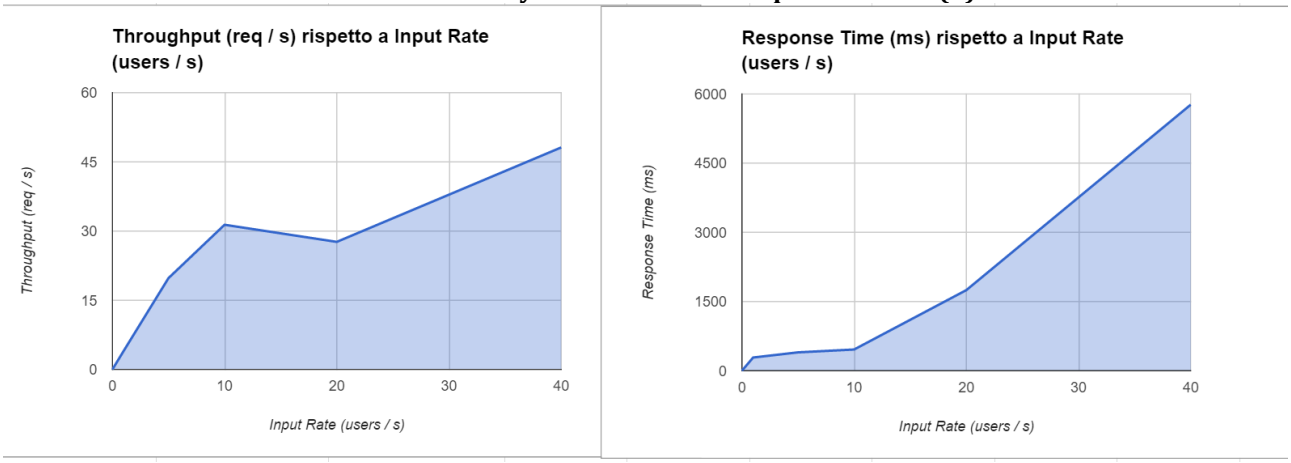

Figure 18. Security Reasoner Evaluate performance (2)

4. Conclusions

This document presents the final prototype of the *Negotiation* module, which is composed of three components (*SLO Manager*, *Supply Chain Manager* and *Security Reasoner*). All components are available online and their functionalities and performance has been successfully tested. We conclude that the final implementation of the *Negotiation* module implements all the corresponding requirements.

Performance analysis illustrates that the components of the Negotiation module are able to handle up to 700 users/second for the *SLO Manager* or up to 5 users/second for POST methods, respectively 40 users/second for GET methods for the *Security Reasoner*. We remind the reader of this deliverable that the Performance Tests of the *Supply Chain Manager* are not reported in this deliverable since it is implemented as a Java library acting as an interface between the *SLO Manager* and the *Planning*. We direct the reader to WP4 deliverables for Performance Tests for the Planning component.

In D2.3.3 *Reference architecture for Cloud SLA negotiation: development and tests - Final Report,* we will present how the Negotiation module advanced the state of the art of Security SLA Negotiation and how the second objective of the project (*Allow user-centric negotiation of Cloud SLA*) related to WP2 (*Negotiation*) has been successfully fulfilled. A detailed description of the negotiation and renegotiation flows will also be available.

5. Bibliography

[1] SPECS, "Negotiation Module. SLO Manager Component," 2015. [Online]. Available: https://bitbucket.org/specs-team/specs-core-negotiation-slomanager.

[2] "Negotiation Module. Supply Chain Manager." [Online]. Available: https://bitbucket.org/specs-team/specs-core-negotiation-supply_chain_manager.

[3] "Negotiation Module. Security Reasoner." [Online]. Available: https://bitbucket.org/specs-team/specs-core-negotiation-securityreasoner.

[4] "Spring Boot Framework." [Online]. Available: https://spring.io/.

[5] "MongoDB." [Online]. Available: https://www.mongodb.org/.

[6] "NIST SP-800-52: Guidelines for the Selection, Configuration, and Use of Transport Layer Security (TLS) Implementations," 2014.

[7] Cloud Security Alliance, "CSA CAIQ Questionnaire." [Online]. Available: https://cloudsecurityalliance.org/star/?r=4376#_registry.

[8] "JUnit." [Online]. Available: http://junit.org/.

[9] "WireMock." [Online]. Available: http://wiremock.org/.

[10] "Code quality report - SLO Manager." [Online]. Available: https://sonar.services.ieat.ro/dashboard/index/1480.

[11] "Code quality report – Security Reasoner." [Online]. Available: https://sonar.services.ieat.ro/dashboard/index/2116

Appendix 1. Negotiation Module – Functional Tests

This annex briefly summarizes the results of the tests performed on the Negotiation Module, namely on *SLO Manager*, *Supply Chain Manager* and *Security Reasoner* components. The SPECS Continuous Integration System (https://bamboo.services.ieat.ro), introduced in deliverable D4.5.2, continuously performs these tests, updating the results every time the code changes.

SLO Manager Tests

In order to test the functionality of the REST API and its underlying methods, we considered writing tests which cover application setup and REST operations over simple resources and over collections. These are mandatory tests that check the persistence in the database and the correctness of resource representation, via XML, such that the negotiation process can be tested.

The most complex test is the one which covers the negotiation process (Test ID negotiationFlow). Since errors have been covered by persistence and representation tests, as explained above, during the negotiation flow test, only the expected behaviour of a user, engaged in the negotiation process, is tested.

Test ID	**testSetUp**
Test objective	Checks if the configuration of leveraged components is done correctly.
Verified requirements	
Inputs	*SLO Manager* properties file
Expected results	Properties are mapped without exceptions
Outputs	N/A
Comments	Operation completed successfully

Test ID	**testCollection**
Test objective	Checks if Collection representation is serialized and deserialized correctly.
Verified requirements	*SLANEG_R10*
Inputs	none
Expected results	Collection information persists after deserialization.
Outputs	N/A
Comments	Operation completed successfully

Test ID	**offerErrors**
Test objective	Verifies functionality of REST API on *SLA Offers*. Checks return messages in case of errors. Retrieval operations for non-existent *SLA Templates* will result in **NOT FOUND** status, while PUT/POST operations on non-existent *SLA Templates* will result in **BAD REQUEST** status.
Verified requirements	*SLANEG_R7, SLANEG_R8, SLANEG_R9, SLANEG_R10*
Inputs	none
Expected results	1. Request creation of *SLA Offers* for non-existent *SLA Template:* **400 BAD REQUEST** 2. Create *SLA Offers* for non-existent template: **404 NOT FOUND** 3. PUT selected *SLA Offer* for non-existent *SLA Templat*e: **400 BAD REQUEST** 4. Retrieve current *SLA Offer* for non-existent *SLA Template*: **404 NOT FOUND**

Outputs	N/A
Comments	All operations completed successfully

Test ID	**retrieveTemplateNotFound**
Test objective	Checks status code in case of retrieval of non-existent *SLA Template.*
Verified requirements	*SLANEG_R7*
Inputs	none
Expected results	**404 NOT FOUND**
Outputs	N/A
Comments	Operation completed successfully

Test ID	**deleteAllTemplates**
Test objective	Checks Delete REST method and empty Collection representation. After the DELETE method returned **204 NO CONTENT**, a GET request is made to retrieve the collection of *SLA Templates*. This collection should contain no *SLA Template* resource URIs.
Verified requirements	*SLANEG_R7*
Inputs	none
Expected results	1. Delete method status: **404 NOT FOUND** 2. Retrieval of *SLA Templates* collection has status: **200 OK** 3. Returned resource Collection equal to `<collection resource="sla-templates" total="0" items="0" members="0"/>`
Outputs	N/A
Comments	All operations completed successfully

Test ID	**createAndDeleteTemplate**
Test objective	Verifies return status for *SLA Template* operations.
Verified requirements	*SLANEG_R7, SLANEG_R19, SLANEG_R20, SLANEG_R21*
Inputs	*SLA Template*
Expected results	*SLA Template* is: 1. created: **201 CREATED** 2. retrived: **200 OK** 3. updated: **200 OK** 4. deleted: **204 NO CONTENT** 5. not found: **404 NOT FOUND**
Outputs	N/A
Comments	All operations completed successfully

Test ID	**negotiationFlow**
Test objective	This test mimics a negotiation scenario in which a NIST *SLA Template* is used to generate two *SLA Offers*, where one is selected to be implemented. The *SLA Manager*, *Planning* and *Services Manager* components are mocked to return predefined behaviour. The mocked services verify REST calls and their content.
Verified requirements	*SLANEG_R1, SLANEG_R2, SLANEG_R3, SLANEG_R7, SLANEG_R8, SLANEG_R10, SLANEG_R11, SLANEG_R16 SLANEG_R19, SLANEG_R20, SLANEG_R21, SLANEG_R22, SLANEG_R23*
Inputs	*SLA Template*
Expected results	1. The negotiation process starts by making a POST to a previously selected template. *Side effects*: a new *SLA Template* is created in *SLA Manager* mock. This template is returned. *Expected status*: **200 OK.** 2. Request for *SLA Offers*, two *SLA Offers* are generated for a given

	SLA Template, based on two predefined supply chains offered by the *Planning* mock. *Side effects*: A *SLA Offer* is created in *SLA Manager* mock for each supply chain. The collection of *SLA Offers* is returned. *Expected status*: **201 CREATED**, *SLA Offers* are ordered by rank. 3. One *SLA Offer* is selected and the initial *SLA Template* is updated with this value. *Side effects*: All *SLA Offers* are deleted from *SLA Manager* mock. All supply chains that are not associated with the selected *SLA Offer* will be deleted. *Expected status*: **200 OK.** 4. Retrieval of current selected *SLA Offer*. *Expected status*: **200 OK.** 5. *SLA Offer* retrieved using the REST API at Step 4 coincides with the *SLA Offer* which was serialized at Step 3.
Outputs	N/A
Comments	All operations completed successfully

Supply Chain Manager Tests

In the following tables we present two unit tests executed for the verification of the behaviour of the *Supply Chain Manager*.

Test ID	**testSupplyChainManager**
Test objective	Verifies the whole process of building supply chains based on the test *SLA Template*.
Verified requirements	*ENF_PLAN_R3, ENF_PLAN_R14*
Inputs	Test *SLA Template*, Planning and Service Manager components are mocked to return expected behaviour.
Expected results	Supply chains should be created successfully and correspond to the test *SLA Template*.
Outputs	List of built supply chains.
Comments	All operations completed successfully.

Test ID	**testUnsupportedMetric**
Test objective	Verifies *Supply Chain Manager* behaviour in case the *SLA Template* contains metric for which no security mechanisms are available.
Verified requirements	*ENF_PLAN_R3, ENF_PLAN_R14*
Inputs	*Test SLA Template, Planning and Service Manager components are mocked to return expected behaviour.*
Expected results	The *Supply Chain Manager* should throw SupplyChainManagerException with corresponding description.
Outputs	No outputs.
Comments	All operations completed successfully.

Security Reasoner Tests

In the following tables, we present the unit tests executed for the verification of the behaviour of the *Security Reasoner.*

Test ID	**Create caiq**
Test objective	Tests if a caiq resource is created correctly when submitting a valid Caiq

	Template
Verified requirements	
Inputs	*A CaiqTemplate in a valid format*
Expected results	The resource should be created and a **201 CREATED** code should be returned
Outputs	N/A
Comments	Operation completed successfully

Test ID	**Fail Create caiq**
Test objective	Tests if an error is raised when trying to create a new caiq resource from an invalid Caiq Template
Verified requirements	
Inputs	*A CaiqTemplate in an invalid format*
Expected results	The resource should not be created and a **435 INVALID INPUT** code should be returned
Outputs	N/A
Comments	Operation completed successfully

Test ID	**Update caiq**
Test objective	Tests if a Caiq is updated correctly when a valid caiq template and a valid resource endpoint are provided
Verified requirements	
Inputs	*CaiqTemplate in a valid format*
Expected results	The resource should be correctly updated and a **200 OK** code should be returned
Outputs	N/A
Comments	Operation completed successfully

Test ID	**Update caiq bad resource**
Test objective	Tests if an error is raised when trying to update a non-existing caiq resource
Verified requirements	
Inputs	*CaiqTemplate in a valid format*
Expected results	A **404 NOT FOUND** error code should be returned
Outputs	N/A
Comments	Operation completed successfully

Test ID	**Update caiq fail**
Test objective	Tests if an error is raised when trying to update a caiq resource by submitting an invalid caiq
Verified requirements	
Inputs	*CaiqTemplate in an invalid format*
Expected results	The resource should not be updated and a **435 INVALID INPUT** code should be returned
Outputs	N/A
Comments	Operation completed successfully

Test ID	Delete caiq
Test objective	Tests if an existing caiq resource is deleted properly
Verified requirements	
Inputs	
Expected results	The resource should be deleted and a **204 DELETED** code should be returned
Outputs	N/A
Comments	Operation completed successfully

Test ID	Delete caiq fail
Test objective	Tests if an error is raised when trying to delete a non-existing caiq resource
Verified requirements	
Inputs	
Expected results	A **404 NOT FOUND** error code should be returned
Outputs	N/A
Comments	Operation completed successfully

Test ID	Retrieve caiq
Test objective	Tests if an existing caiq resource is retrieved properly
Verified requirements	
Inputs	
Expected results	The resource should be retrieved and a **200 OK** code should be returned
Outputs	N/A
Comments	Operation completed successfully

Test ID	Retrieve caiq fail
Test objective	Tests if an error is raised when trying to retrieve a no-existing caiq resource
Verified requirements	
Inputs	
Expected results	A **404 NOT FOUND** error code should be returned
Outputs	N/A
Comments	Operation completed successfully

Test ID	Retrieve All caiqs
Test objective	Tests if the *Caiq Collection* is retrieved properly.
Verified requirements	
Inputs	
Expected results	The collection should be retrieved and a **200 OK** code should be returned
Outputs	N/A
Comments	Operation completed successfully

Test ID	Create association caiq/judgement
Test objective	Tests if the association caiq/judgement is properly created
Verified requirements	
Inputs	*Judgement-id*
Expected results	The association should be created and a **201 CREATED** code should be returned
Outputs	N/A
Comments	Operation completed successfully

Test ID	Create association caiq/judgement bad caiq
Test objective	Tests if an error is raised when trying to create an association caiq/judgement with a non-existing caiq
Verified requirements	
Inputs	*Judgement-id*
Expected results	The **404 NOT FOUND** error code should be returned
Outputs	N/A
Comments	Operation completed successfully

Test ID	Create association caiq/judgement bad judgment
Test objective	Tests if an error is raised when trying to create an association caiq/judgement with a non-existing judgement
Verified requirements	
Inputs	*Judgement-id*
Expected results	The **422 UNPROCESSABLE ENTITY** error code should be returned
Outputs	N/A
Comments	Operation completed successfully

Test ID	Evaluate caiq
Test objective	Tests if a caiq is correctly evaluated based on an existing caiq-judgement association
Verified requirements	
Inputs	*Judgement-id or none*
Expected results	The Caiq should be correctly evaluated and a **201 CREATED** code should be returned. If no judgement parameter is specified in the query string, the evaluation is done with the default judgement.
Outputs	N/A
Comments	Operation completed successfully

Test ID	Evaluate caiq no association
Test objective	Tests if an error is raised when trying to evaluate a caiq based on a non-existing caiq-judgement association
Verified requirements	
Inputs	*Judgement-id or none*
Expected results	A **409 CONFLICT** error code should be returned
Outputs	N/A
Comments	Operation completed successfully

Test ID	Create judgement
Test objective	Tests if a judgement resource is properly created when submitting a judgment in a valid format
Verified requirements	
Inputs	*JudgementTemplate*
Expected results	The association should be created and a **201 CREATED** code should be returned
Outputs	N/A
Comments	Operation completed successfully

Test ID	Create judgement invalid format
Test objective	Tests if an error is raised when trying to create a new judgment from an invalid template

Verified requirements	
Inputs	*JudgementTemplate*
Expected results	The judgement should not be created and a **435 INVALID INPUT** code should be returned
Outputs	N/A
Comments	Operation completed successfully

Test ID	**Update judgement**
Test objective	Tests if a *Judgement* is properly updated when submitting a valid template
Verified requirements	
Inputs	*JudgementTemplate*
Expected results	The *Judgement* should be created and a **201 CREATED** code should be returned
Outputs	N/A
Comments	Operation completed successfully

Test ID	**Update judgement invalid format**
Test objective	Tests if an error is raised when trying to update a judgment with an invalid template
Verified requirements	
Inputs	*JudgementTemplate*
Expected results	The *Judgement* should not be updated and a **435 INVALID INPUT** code should be returned
Outputs	N/A
Comments	Operation completed successfully

Test ID	**Update judgement bad resource**
Test objective	Tests if an error is raised when trying to update a non-existing judgment resource
Verified requirements	
Inputs	*JudgementTemplate*
Expected results	A **404 NOT FOUND** code should be returned
Outputs	N/A
Comments	Operation completed successfully

Test ID	**Delete judgement**
Test objective	Tests if a judgement is deleted properly
Verified requirements	
Inputs	
Expected results	The *Judgement* should be deleted and a **204 DELETED** code should be returned
Outputs	N/A
Comments	Operation completed successfully

Test ID	**Delete judgement bad resource**
Test objective	Tests if an error is raised when trying to delete a non-existing judgment resource
Verified requirements	
Inputs	
Expected results	A **404 NOT FOUND** code should be returned

Outputs	N/A
Comments	Operation completed successfully

Test ID	**Retrieve judgement**
Test objective	Tests if an existing judgement is correctly retrieved
Verified requirements	
Inputs	
Expected results	The judgement should be retrieved and a **200 OK** code should be returned
Outputs	N/A
Comments	Operation completed successfully

Test ID	**Retrieve All judgements**
Test objective	Tests if the *Judgement Collection* is correctly retrieved
Verified requirements	
Inputs	
Expected results	The Collection should be retrieved and a **200 OK** code should be returned
Outputs	N/A
Comments	Operation completed successfully

Test ID	**Create Slacaiq**
Test objective	Tests is a *SlaCaiq* resource is created correctly
Verified requirements	*SLANEG_R7, SLANEG_R8, SLANEG_R9, SLANEG_R11, SLANEG_R14*
Inputs	*SlaCaiqTemplate*
Expected results	The SLACaiq resource should be created and a **201 CREATED** code should be returned
Outputs	N/A
Comments	Operation completed successfully

Test ID	**Create Slacaiq invalid template**
Test objective	Tests if an error is raised when trying to create a *SlaCaiq* resource from an invalid template
Verified requirements	*SLANEG_R7, SLANEG_R8, SLANEG_R9, SLANEG_R11, SLANEG_R14*
Inputs	*SlaCaiqTemplate*
Expected results	A **435 INVALID INPUT code** should be returned
Outputs	N/A
Comments	Operation completed successfully

Test ID	**Create Slacaiq judgement not existing**
Test objective	Tests if an error is raised when trying to create a *SlaCaiq* resource from an invalid template
Verified requirements	*SLANEG_R7, SLANEG_R8, SLANEG_R9, SLANEG_R11, SLANEG_R14*
Inputs	*SlaCaiqTemplate*
Expected results	A **409 SYSTEM NOT INITIALIZED code** should be returned
Outputs	N/A
Comments	Operation completed successfully

Test ID	**Retrieve All Slacaiqs**
Test objective	Tests if the *SlaCaiq Collection* is retrieved properly
Verified	

requirements	
Inputs	
Expected results	The Collection should be retrieved and a **200 OK** code should be returned
Outputs	N/A
Comments	Operation completed successfully

Test ID	**Retrieve Slacaiq score**
Test objective	Tests if the *SlaCaiq* score is properly retrieved
Verified requirements	*SLANEG_R7, SLANEG_R8, SLANEG_R9, SLANEG_R11, SLANEG_R14*
Inputs	*SlaCaiq-id OR SlaCaiq-id and QueryString*
Expected results	The SlaCaiq score should be retrieved and a **200 OK** code should be returned
Outputs	N/A
Comments	Operation completed successfully