SciMantic

Deliverable D2.2

Architecture of a Secure and Scalable Semantic Content Infrastructure

The SciMantic Consortium

University of Zurich
Trialox AG

For more information on this document or the SciMantic project, please contact:

Prof. Dr. Burkhard Stiller
University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14
CH-8050 Zürich
Switzerland

Phone: +41 44 635 67 10
Fax: +41 44 635 68 09
E-mail: stiller@ifi.uzh.ch

Document Control

Title:	Architecture of a Secure and Scalable Semantic Content Infrastructure
Type:	R&D
Editor:	Hasan
E-mail:	hasan@ifi.uzh.ch
Author:	Hasan
Contributors:	Manuel Innerhofer
Delivery Date:	11. 09. 2010

Executive Summary

This deliverable presents the design of the architecture of a secure and scalable Semantic Content Infrastructure (SCI). The SCI is a service-based infrastructure which provides a set of functionality for management and sharing of semantically linked data (content). SCI is a network of co-operating and independent nodes, where each node may join and leave the network at any time. Each SCI node implements an instance of a Semantic Content Sharing System (SCSS). The SCSS is an extensible service platform based on OSGi (Open Services Gateway initiative). It provides a framework to expose OSGi services as RESTful Web services and a number of interfaces to allow for secured access and manipulation of semantic contents. Furthermore, it enables content annotations with keywords and searching of shared content based on keywords. The Apache Project Clerezza initiated by trialox AG is developing a platform which serves as a basis for SCSS. In this deliverable, architectural components of SCSS and their interfaces are described in detail.

Table of Contents

1 Introduction 1
2 Architecture Overview 1
3 Smart Content Binding (SCB) 3
3.1 Triple Collections 3
3.2 GraphNode 4
3.3 Triple Collection Providers 5
3.4 Parser and Serializer 5
3.5 SPARQL Support 6
3.6 Triple Collection Manager 6
4 RESTful Web Service Framework 6
4.1 JAX-RS Resources 6
4.2 JAX-RS Providers 6
4.3 Web Request Processing 8
4.4 Type Handling 8
4.5 Type Rendering 9
5 Security 10
5.1 Authentication 10
5.2 Authorization 11
6 User Management 12
7 Concepts 13
8 Content Sharing 14
9 Summary 15

1 Introduction

The Semantic Content Infrastructure (SCI) comprises a network of interacting nodes which may join and leave the network at any time. Each SCI node implements an instance of a Semantic Content Sharing System (SCSS) and provides Web services which are accessible in a *secured* way: users are *authenticated* and their actions must pass an *authorization* process. Those Web services provide functionality to share contents either to the public or within a closed community. Shared contents in a community can be replicated and stored in a distributed fashion to increase availability. To allow searching of distributed contents in a *scalable* way, each SCI node implements distributed indexing within the SCSS. Since the SCSS provides for functionality to annotate contents with keywords (also called concepts), searching distributed contents based on concepts is one of the key functions to be supported. Key requirements of SCI are defined in D2.1 [3]. In this deliverable, the architecture of SCI is designed to meet those requirements and to enable implementation of the scenario described in D2.1.

The remaining sections in this deliverable are organized as follows: in Section 2 the overview of the architecture is presented. Section 3 describes the Smart Content Binding, which provides an access layer to the underlying data stores, and Section 4 presents the RESTful Web service framework in detail. While Section 5 illustrates the security mechanisms designed, the subsequent sections 6, 7, and 8 describe the user management, content sharing, and concept management respectively. Finally, a summary of the architecture design is given in Section 9.

2 Architecture Overview

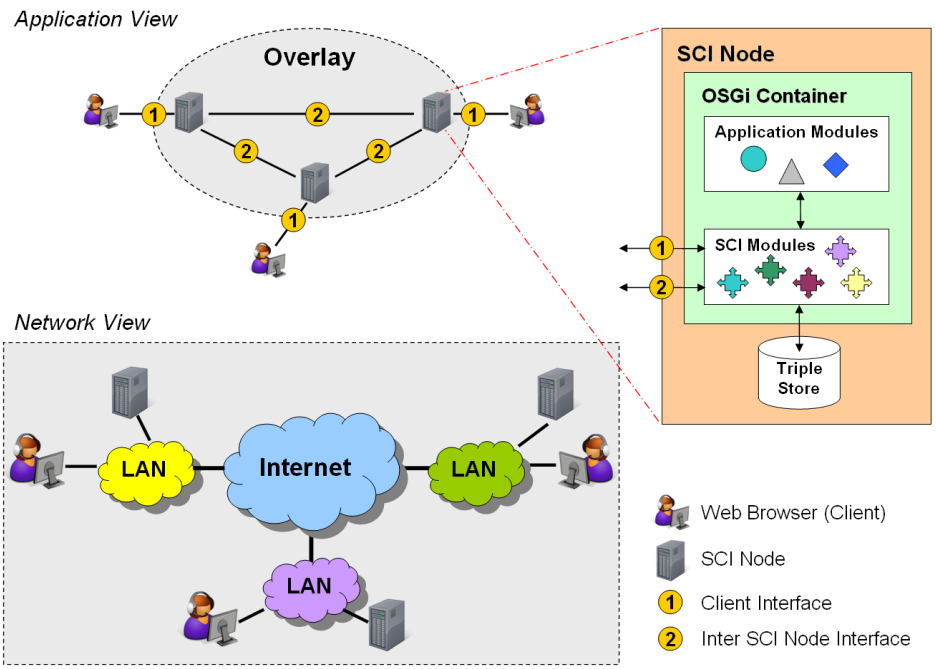

Figure 1: An overview of the SCI architecture.

An overview of the SCI architecture is depicted in Figure 1. Users use Web browsers (clients) to access contents (resources) managed by SCI nodes running an instance of SCSS. Trialox has initiated the development of the Apache Project Clerezza which provides a foundation for SCSS.

The SCI nodes build an overlay network which semantically links contents managed by each node. An inter-node interface defines the communication among SCI nodes. This interface comprises protocols required to implement various signaling and data exchange. This includes protocols for distributed indexing, contents querying, searching, and transfer.

The OSGi (Open Services Gateway initiative) [7], [8] architecture is chosen as the underlying software architecture for the SCSS to achieve a service oriented and modular design of software components. The OSGi architecture is described in D2.1, and is preferred compared to Service Component Architecture (SCA) [5] due to its maturity in implementation and support.

Figure 2 presents the architectural components of the SCSS. Interactions of those components are described in detail in subsequent sections. To perform a certain function, a component may use OSGi services provided by other components. OSGi services provided by a component can also be exposed as Web services, to be made accessible through the Web service framework.

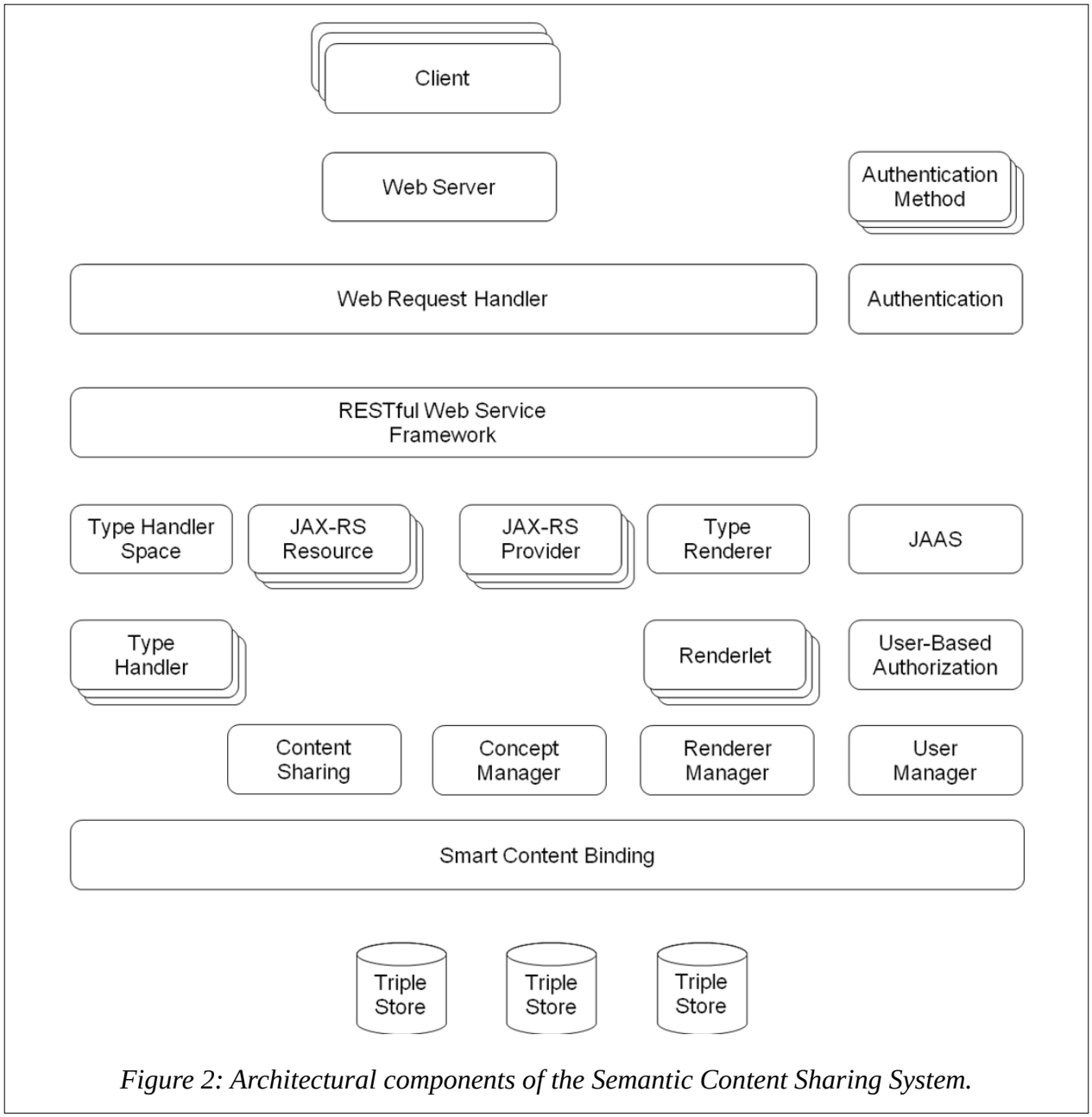

Figure 2: Architectural components of the Semantic Content Sharing System.

3 Smart Content Binding (SCB)

SCB defines a technology-agnostic layer to access and modify triple stores. It provides a java implementation of the graph data model specified by W3C RDF [9] and functionalities to operate on that data model. SCB offers a service interface to access multiple named graphs and it can use various weighted providers to manage RDF graphs in a technology specific manner, *e.g.*, using Jena [4] or Sesame [1]. It also provides for adaptors that allow an application to use Jena or Sesame APIs to process RDF graphs. Furthermore, SCB offers a serialization and a parsing service to convert a graph into a certain representation (format) and vice versa. The architecture of SCB is depicted in Figure 3 and described in details in the following subsections.

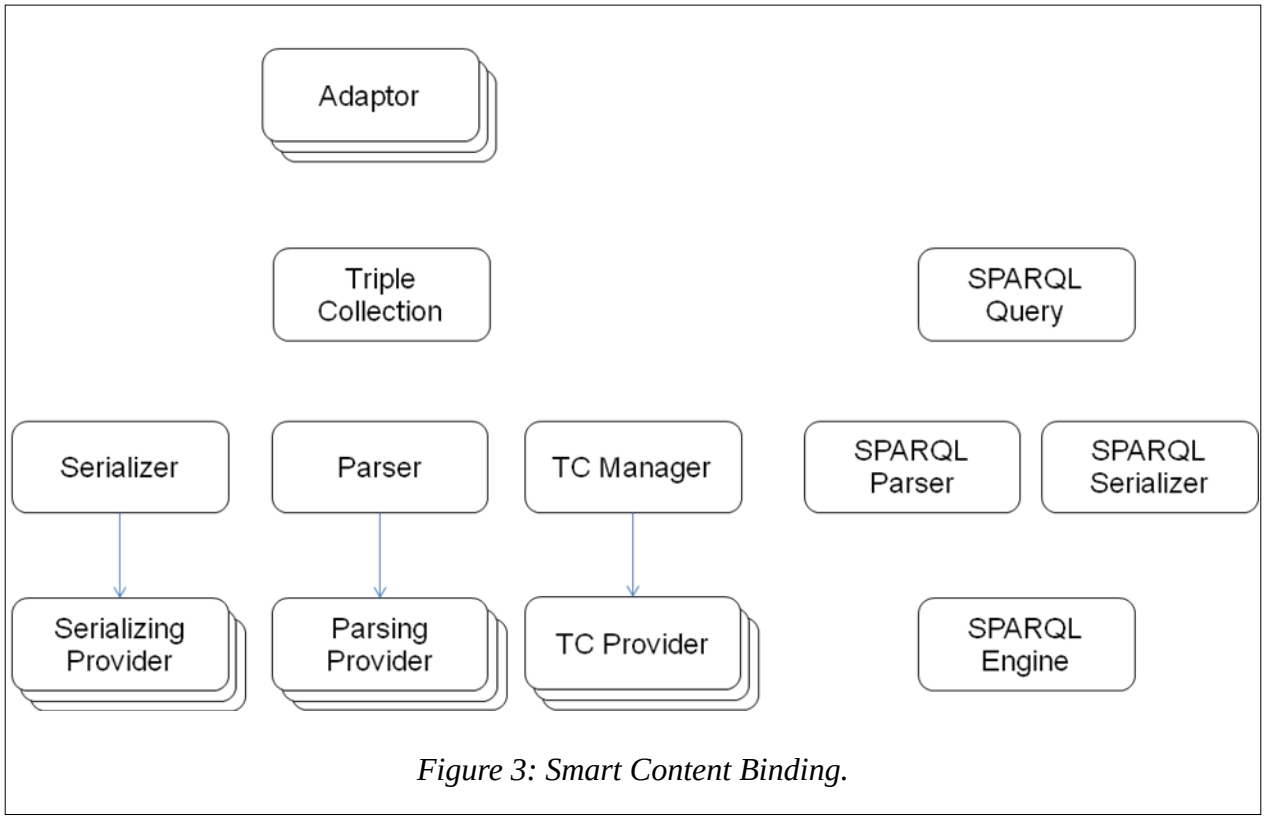

Figure 3: Smart Content Binding.

3.1 Triple Collections

A Triple Collection is a *set* of triples, and thus, does not contain duplicates. A Triple Collection can be seen as a database in Relational Database Management Systems. Each triple comprises a subject, an object, and a predicate which relates the subject to the object. Note that, this relation is a directed relation. Possible data types (classes) that a subject, a predicate, or an object can have, is defined in Table 1. Figure 4 depicts the class diagram of those data types, which are described in detail in [9].

Table 1: Data types for subject, predicate, and object of a triple.

	Subject	Predicate	Object
Plain Literal	-	-	x
Typed Literal	-	-	x
Blank Node	x	-	x
URI Reference	x	x	x

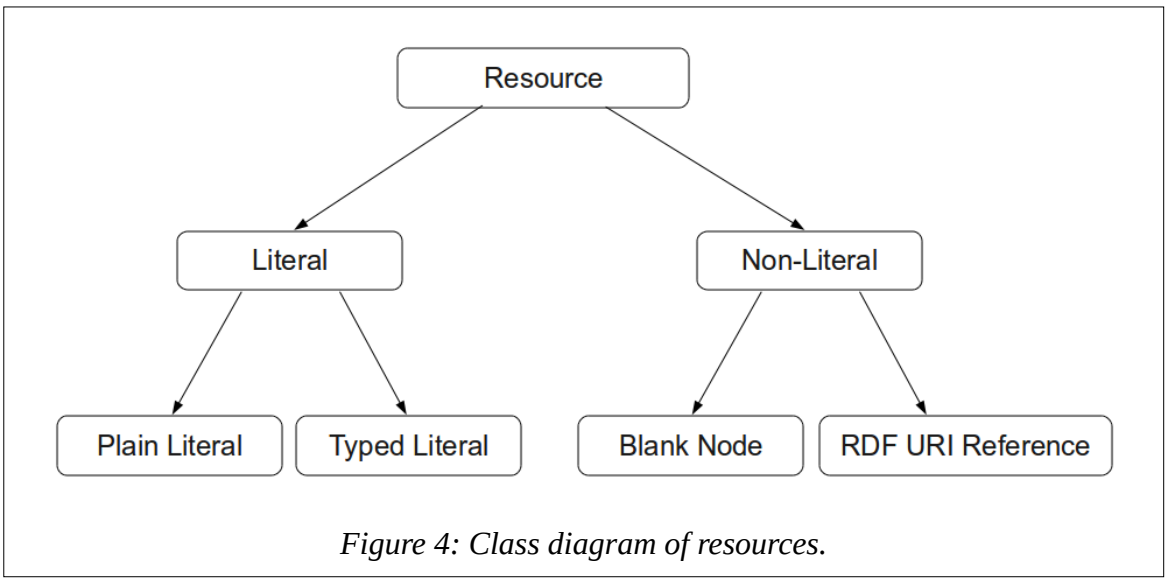

Figure 4: Class diagram of resources.

The triples in a Triple Collection constitute a directed graph. Principally all triples can be placed within a single graph. However, it is practical to group triples and give them a name, resulting in a named graph. This name allows a graph to be referred to when accessing it. In many cases, it is useful to have graphs which are not modifiable, i.e., triples cannot be added to neither removed from the graph. Graphs which are not modifiable, are called Immutable Graphs (or just Graphs), whereas modifiable graphs are called Mutable Graph (or in short MGraph). Table 2 lists functions to be supported by MGraphs and Graphs.

Table 2: Main functions supported by MGraphs and Graphs.

Function	Description	Triple Collection
Filter	Given a triple pattern, this operation must return all triples that match the pattern.	MGraphs and Graphs
Add Listener	Register a listener which will be notified if there is a change in the MGraph which match the specified pattern.	MGraphs
Remove Listener	Deregister a listener.	MGraphs
Equals	Test on isomorphism of two graphs.	Graphs

In order to prevent concurrent modifications on an MGraph by different threads, a graph locking mechanism is required. Setting a read-lock on an MGraph prevents other threads from writing the MGraph, whereas setting a write-lock prevcnts other threads from reading and writing it.

3.2 GraphNode

A GraphNode is an object which represents a node (RDF resource) in a Triple Collection. It provides useful methods to obtain information about the node. Table 3 lists these methods and their descriptions.

Table 3: Main functions supported by GraphNodes.

Function	Description
Get Node	The context of a node are the triples containing the node

Context	as subject or object and recursively the context of the blank nodes in any of these statements (triples). This method returns a Graph containing these triples. Blank nodes in this Graph are the same instances as in the original Triple Collection.
Delete Node Context	Delete the context of the node.
Get Objects	Get the objects of statements with this node as subject and a specified property as predicate.
Get Subjects	Get the subjects of statements with this node as object and a specified property as predicate.
Get Properties	Get all available properties of this node as subject.
Get Inverse Properties	Get all available properties of this node as object.
Add Property	Add a property to the node with the specified predicate and object.
Delete Properties	Delete all statement with the node as subject and the specified predicate.
Delete Property	Delete a statement with the node as subject, the specified property as predicate, and the specified resource as object.

3.3 Triple Collection Providers

A Triple Collection (TC) Provider provides a service to access and manipulate Triple Collections implemented in a specific technology. Table 4 lists the main functions that a TC Provider must support.

Table 4: Main functions supported by a TC Provider.

Function	Description
Create MGraph	Create an MGraph for the specified name.
Create Graph	Create a Graph for the specified name with triples of the specified Triple Collection.
Delete Triple Collection	Delete a Triple Collection, i.e. a Graph or MGraph of the specified name.
Get Graph	Return a Graph of the specified name.
Get MGraph	Return an MGraph of the specified name.

To allow choosing a certain TC Provider from a set of available TC Providers, a configurable weight is assigned to each TC Provider.

3.4 Parser and Serializer

A Parsing Provider provides the functionality to parse a Triple Collection from its serialized form into a Graph. Each Parsing Provider is characterized by its supported format. A Parser is a singleton that offers the parsing function by delegating it to registered Parsing Providers.

A Serializing Provider provides the functionality to serialize a Triple Collection into the specified format. Each Serializing Provider is characterized by its supported format. A Serializer is a singleton that offers the serialization function by delegating it to registered Serializing Providers.

3.5 SPARQL Support

SPARQL [11] is a protocol and query language for RDF. The SCB architecture defines 4 components to support SPARQL: SPARQL Query, SPARQL Parser, SPARQL Serializer, and SPARQL Engine. In SPARQL Query, Java classes are defined to model the 4 forms of queries defined in SPARQL: Select, Construct, Ask, and Describe. The SPARQL Parser provides the functionality to parse a String into a Query object, whereas the SPARQL Serializer does the opposite. The SPARQL Engine provides a function to execute the specified SPARQL Query on the specified Triple Collection.

3.6 Triple Collection Manager

The Triple Collection (TC) Manager is a singleton that provides access to Triple Collections through registered TC Providers. It also provides methods to execute SPARQL queries through a registered SPARQL Query Engine.

4 RESTful Web Service Framework

The main component of the framework is an implementation of the JSR-311 (JAX-RS) specification [2], a Java API for RESTful Web services. The specification enables easy implementation of RESTful Web services using JAX-RS annotations, based on Java annotation mechanism. There are several implementations of this specification available, but trialox implementation called Triaxrs provided extended functionality, especially Type Handling and Type Rendering, which are described in detail in this section. Type Handling is a mechanism to select a Web service to process the Web request based on the RDF type of the requested resource, and Type Rendering is a mechanism to select a rendering definition to process the Web response based on the RDF type of the returned GraphNode.

4.1 JAX-RS Resources

A Web request is handled by a certain method of an intance of a certain Java class. This Java class is called a JAX-RS resource (in JAX-RS terminology, it is called a resource class), and the method is called a resource method. JAX-RS resources and their resource methods are annotated with JAX-RS annotations in order to allow Triaxrs to find and instantiate the right JAX-RS resource and invoke the right resource method to process an incoming Web request. JAX-RS annotations are used to specify URL paths, HTTP methods, consumed and produced media types, and Web request parameters.

4.2 JAX-RS Providers

The functionality of a JAX-RS runtime is extended using application-supplied provider classes. JAX-RS specifies 3 types of providers: Entity Providers, Context Providers, and Exception Mapping Providers, as depicted in Figure 5. Entity providers supply mapping services between representations and their associated Java classes. There are 2 types of Entity Providers: Message Body Reader (MBR) and Message Body Writer (MBW). Context Providers supply context to resource classes and other providers, while Exception Mapping Providers map a checked or runtime exception to an instance of JAX-RS Response. Interested readers are recommended to study [2] for detail descriptions of JAX-RS providers and applicable JAX-RS annotations.

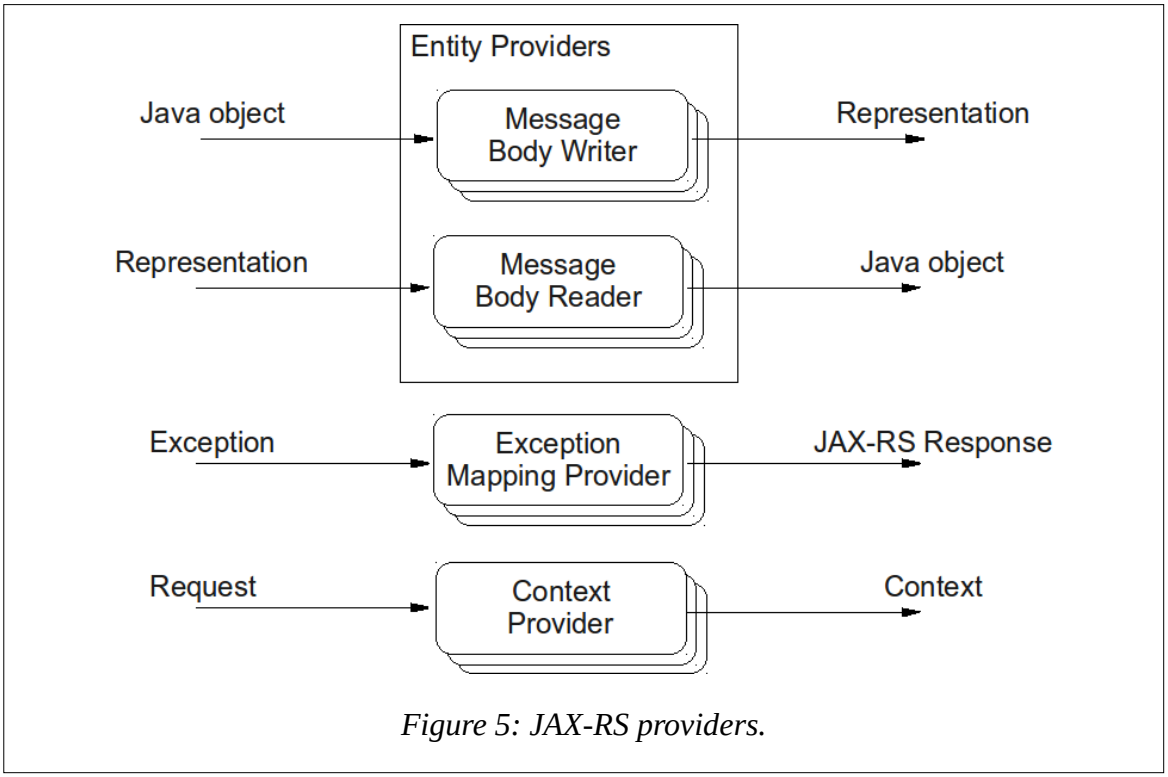

Figure 5: JAX-RS providers.

A number of MBRs and MBWs are to be implemented within the framework in order to support Java classes which are often used. They are listed in Table 5.

Table 5: MBRs and MBWs to be implemented by the framework.

Java Class	MBR	MBW
String	x	x
byte[]	x	x
java.io.File	x	x
javax.ws.rs.core.MultivaluedMap<String, String> (defined by JAX-RS for HTML form's content type application/x-www-form-urlencoded)	x	x
java.io.InputStream	x	x
java.io.Reader	x	x
javax.xml.transform.stream.StreamSource	x	-
javax.xml.transform.sax.SAXSource	x	-
javax.xml.transform.dom.DOMSource	x	-
javax.xml.transform.Source	-	x
javax.ws.rs.core.StreamingOutput	-	x
org.apache.clerezza.jaxrs.utils.form.MultiPartBody (defined by apache clerezza for HTML form's content type multipart/form-data	x	-
org.apache.clerezza.rdf.core.Graph (implemented within SCB)	x	x

4.3 Web Request Processing

Figure 6 shows the typical sequence of interactions between various components involved in the processing of a Web request after delivered to the Triaxrs Core. The Triaxrs Core selects a resource method by matching annotated resource methods to the Web request. Before invoking the matched method, all parameters of the method are constructed by using relevant Context Providers and Message Body Readers. The result of the method invocation is a Java object (an instance of a certain Java class). The Triaxrs Core looks for a matching MBW to render the resulting Java object. Finally, a Web response is generated and sent to the requesting client through the Web server.

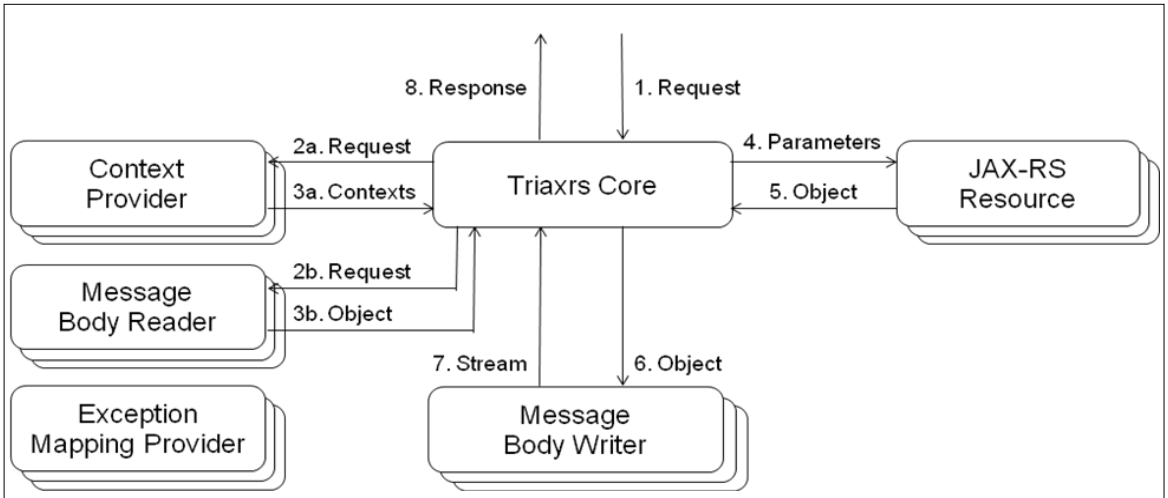

Figure 6: Web request processing.

4.4 Type Handling

SciMantic provides Web services which access and manipulate RDF graphs. It is beneficial to have a mechanism which allows for selecting a service based on the RDF type of an RDF resource. This mechanism is termed Type Handling. A Java class which provides a service for processing RDF resources of specific RDF types is called a Type Handler. To support Type Handling using Triaxrs, a JAX-RS resource with the respective resource method is needed which is able to find the right Type Handler to handle the specified (requested) RDF resource. This JAX-RS resource is called Type Handler Space.

In JAX-RS specification, the URL of a Web request determines candidate JAX-RS resources and methods, whereas RDF uses a URI Reference (which can also be used as URL) to identify a resource. Note that an RDF resource can also be a blank node, in which case it is not processable through Type Handling. Therefore, a Type Handler Space can be annotated to match any URL, and it can use the URL as the URI Reference of the RDF resource to be processed. However, this requires:

- The URI Reference of an RDF resource must use the same scheme and authority part as the URL of the Web service.
- A minor change in the resource matching algorithm in the JAX-RS Maintenance Release Specification 1.1, in order to ease the implementation of the Type Handling mechanism. This minor change was proposed to JSR 311 Project by trialox and received a positive feedback.

The sequence of a Web request processing involving a Type Handler is depicted in Figure 7. The Type Handler Space matches any URL and returns a Type Handler supporting an RDF type of the RDF resource requested. In order to find the right Type Handler, the Type Handler Space needs to access the Triple Store which has the triple stating the RDF type of the requested RDF resource. Furthermore, Type Handlers are annotated with information on supported RDF types and a property denoting that the Java class is a Type Handler. The Type Handler acts as a JAX-RS sub-resource to be matched with the HTTP method of the Web request. The matching sub-resource method is then invoked with the required parameters by the Triaxrs Core. The remaining steps are the same as in the previous Section.

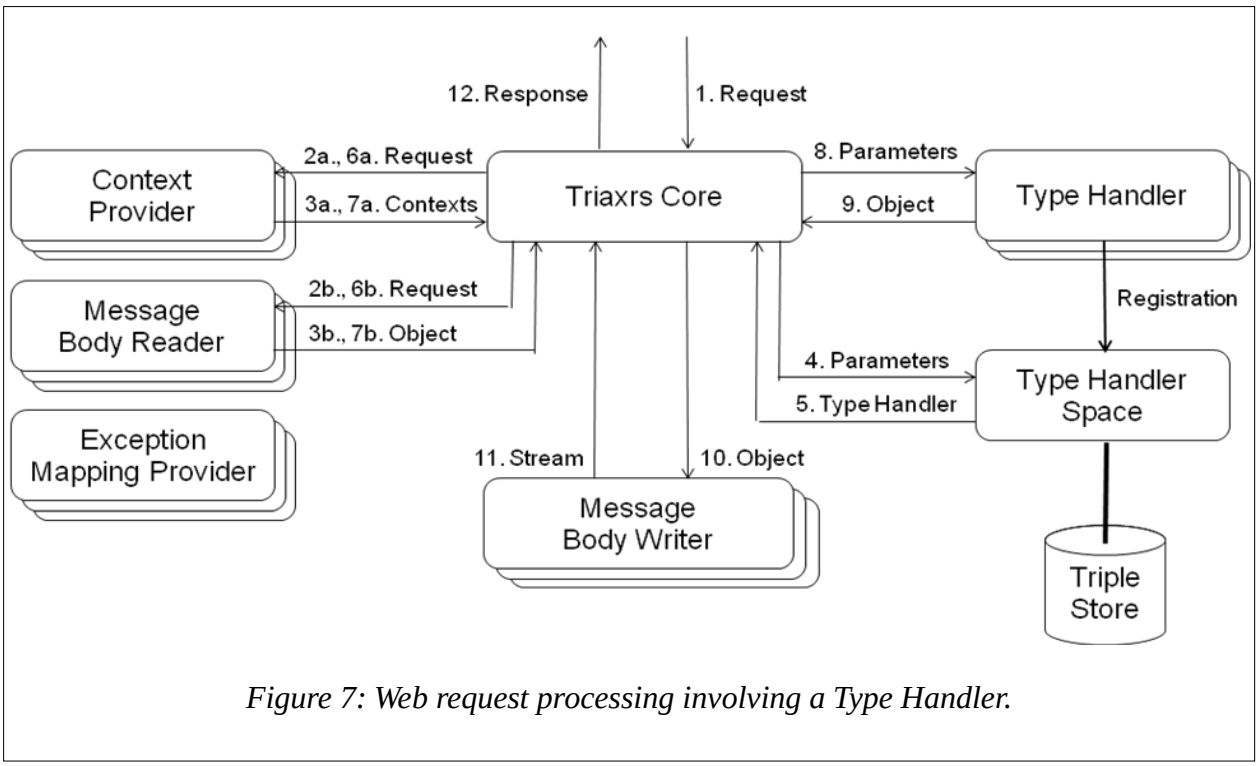

Figure 7: Web request processing involving a Type Handler.

4.5 Type Rendering

Similar to Type Handling a mechanism to map the RDF type of an RDF resource to a rendering definition will be useful. This mechanism is termed Type Rendering. A rendering definition stores the following information:

- Rendering Specification: a template for rendering a GraphNode.
- Renderlet: an object that provides the functionality to render a GraphNode based on the Rendering Specification
- Media Type: the media type of the resulting document.
- Rendering Mode: a parameter to configure the rendering behavior.
- Rendered Type: the applicable RDF type.

In order to support Type Rendering in Triaxrs, a generic Message Body Writer for GraphNodes is required, which is annotated as being capable to produce any media type. Based on the accept header of the Web request, the optional query parameter "mode" in the Web request, and the RDF type of the GraphNode to be rendered, a matching rendering definition is selected. The renderlet is extracted from the rendering definition and its render method is invoked to render the GraphNode according to the rendering specification. Figure 8 shows the Web request processing which involves the use of renderlets.

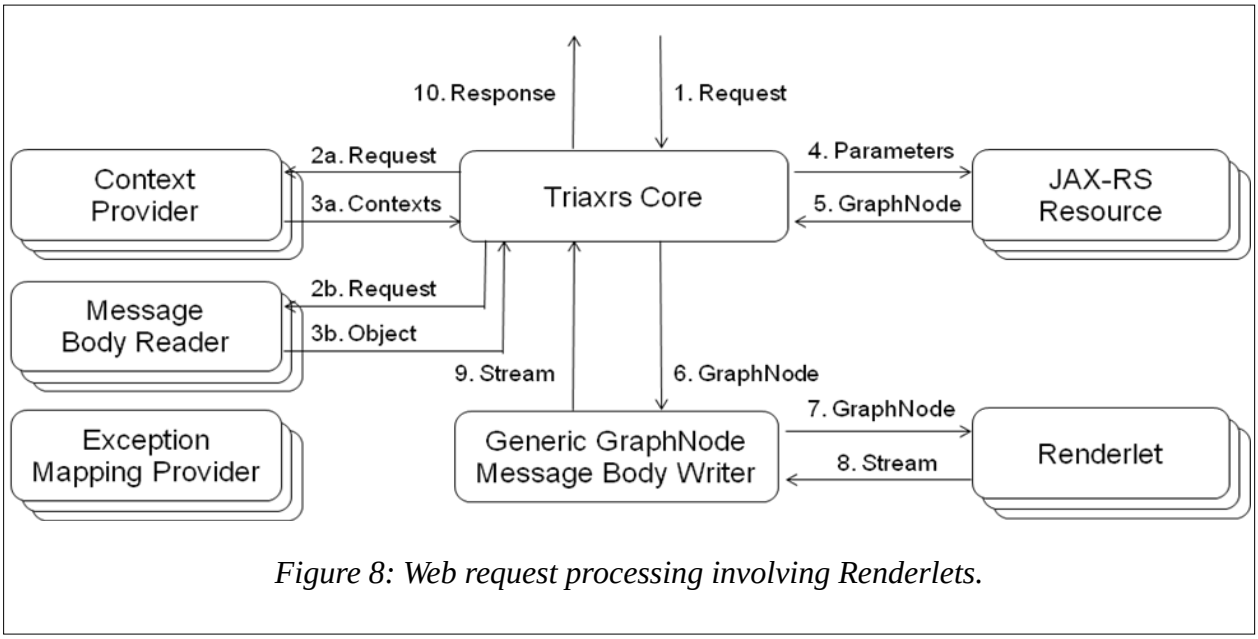

Figure 8: Web request processing involving Renderlets.

5 Security

Some services provided by SCSS are restricted to specific users only. Therefore, SCSS must support user authentication and authorization.

5.1 Authentication

Figure 9 shows the authentication process. Before a Web request is delivered to Triaxrs Core for processing, the user who submitted the Web request must first be authenticated. The Web Request Handler forwards the Web request received from the Web server to the Authentication component. This component invokes registered Authentication Methods in a sequence according to their priority (weight). Each Authentication Method extracts user credentials from the Web request and returns the user name if the user can be authenticated successfully. Otherwise, it generates a Web response to inform the client about the failed authentication. The platform should support HTTP basic authentication and cookie-based authentication. In case of HTTP basic authentication, if the Web request does not contain the user credentials, the UNAUTHORIZED response status code is sent to the client. In case of cookie-based authentication, a failed authentication leads to a redirection to a login page.

After the user (a.k.a subject in JAAS) is successfully authenticated, the Web request is delivered to the Triaxrs Core to be processed. This Web request processing is carried out within the method Subject.doAsPrivileged, a JAAS authorization mechanism. This method receives three parameters: a subject, an action, and an access control context. The specified action is carried out as the specified subject within the specified access control context. This means, Triaxrs Core processes the Web request on behalf of the authenticated user (subject). Doing this is necessary to enable checking the rights of a subject to perform a particular action, as described in the next section.

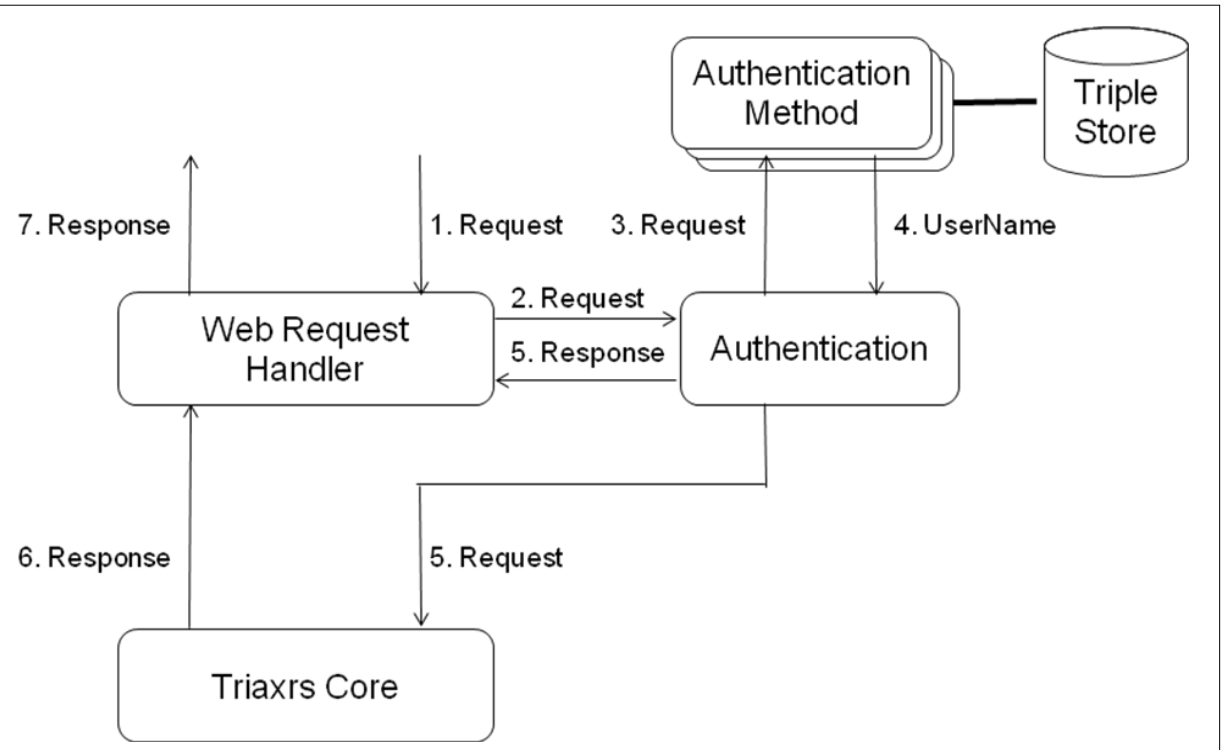

Figure 9: Authentication.

5.2 Authorization

Authorization in SCSS is based on JAAS. The following steps are performed to use JAAS for authorization:

1. A Security Manager must be activated. SCSS activates a default Security Manager provided by the Java Virtual Machine when SCSS starts.
2. A security policy must be set. The User-Based Authorization component of SCSS sets a security policy which implements the Java Policy API function getPermissions to deliver permissions (access rights) for a given subject. These permissions are stored in a specific Triple Collection in SCSS.
3. Before executing a code segment that requires an authorization, e.g., when a user wants to modify an MGraph, an access control is triggered by invoking the method AccessController.checkPermission and passing the respective Java Permission object as the parameter (in this case, a Java Permission object for modifying an MGraph). Since the code segment is performed on behalf of the authenticated subject, JAAS consults (retrieve permissions of a subject from) currently installed security policy, in order to check, whether the subject has the requested permission.

Figure 10 shows a typical interaction of components involved in authorization.

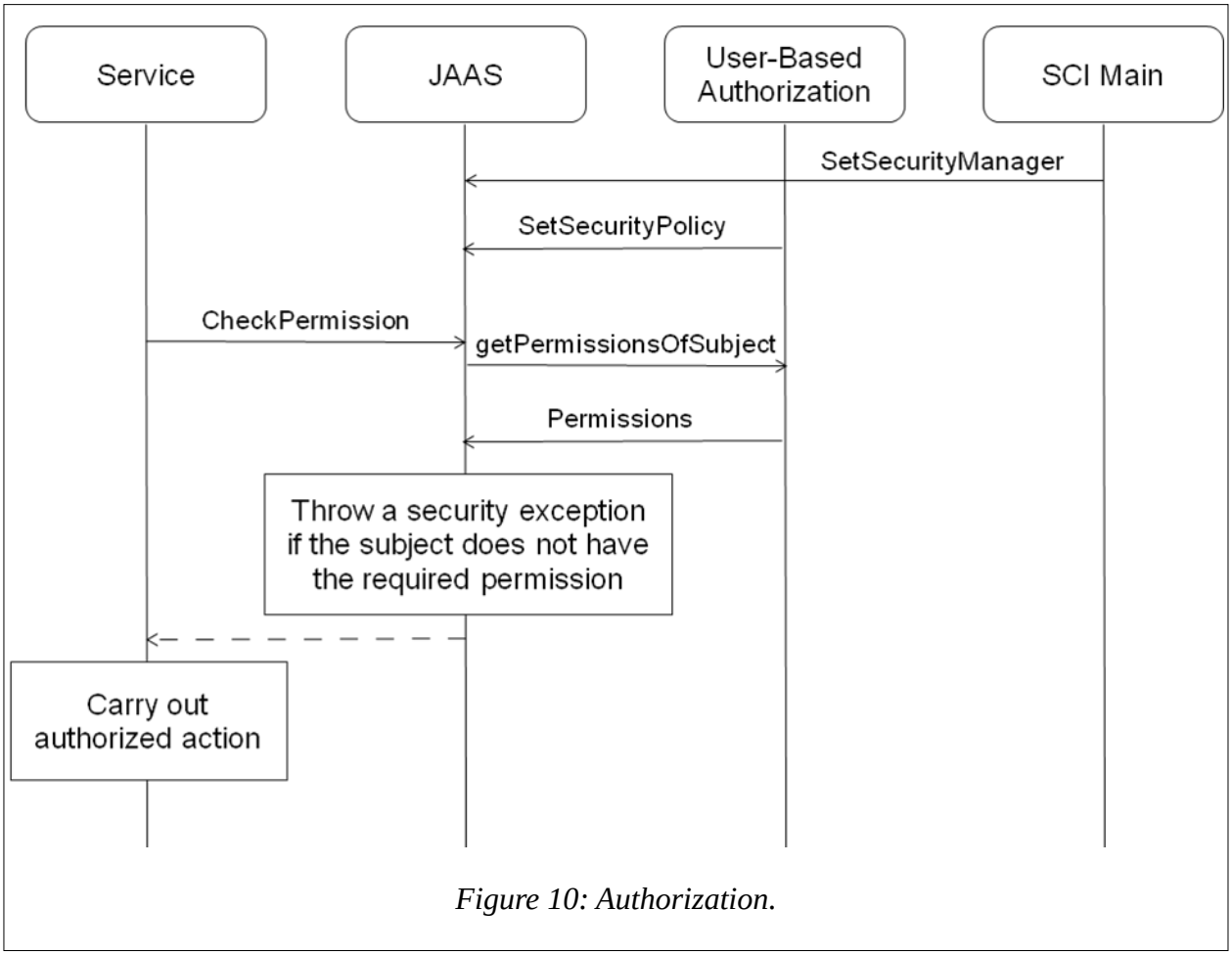

Figure 10: Authorization.

6 User Management

The User Manager provides a service to create users and roles, and assign permissions to them. A role represents a set of rights which are needed by a user having a certain function to perform her tasks. To describe users and roles, a set of ontologies is used as depicted in Figure 11. A user is defined as a FOAF Agent, and she can be assigned a set of roles. A user inherits the permissions of her roles. This means, assigning a user a certain role has the effect of assigning the permissions of this role to the user. Furthermore, a user can be assigned role-independent permissions. The property has_function of SIOC is used to assign roles to users. By applying this property also to roles, all permissions of a role can be passed on to other roles. Since properties of the class Role are also properties of the class Agent, the class Role is defined as a superclass of the class Agent.

Two own ontology namespaces are defined, as can be seen in Figure 11: PLATFORM and PERMISSION. This is due to the fact, that existing ontologies do not provide required properties and classes. Instead of FOAF name, the property userName, defined in own ontology termed PLATFORM, is used to identify a user, because unlike FOAF name, the value of userName must be unique. To allow usage of permission definitions besides Java-based permissions, the class Permission in the namespace PERMISSION is specified. Therefore, the property hasPermission points to an object of the class Permission.

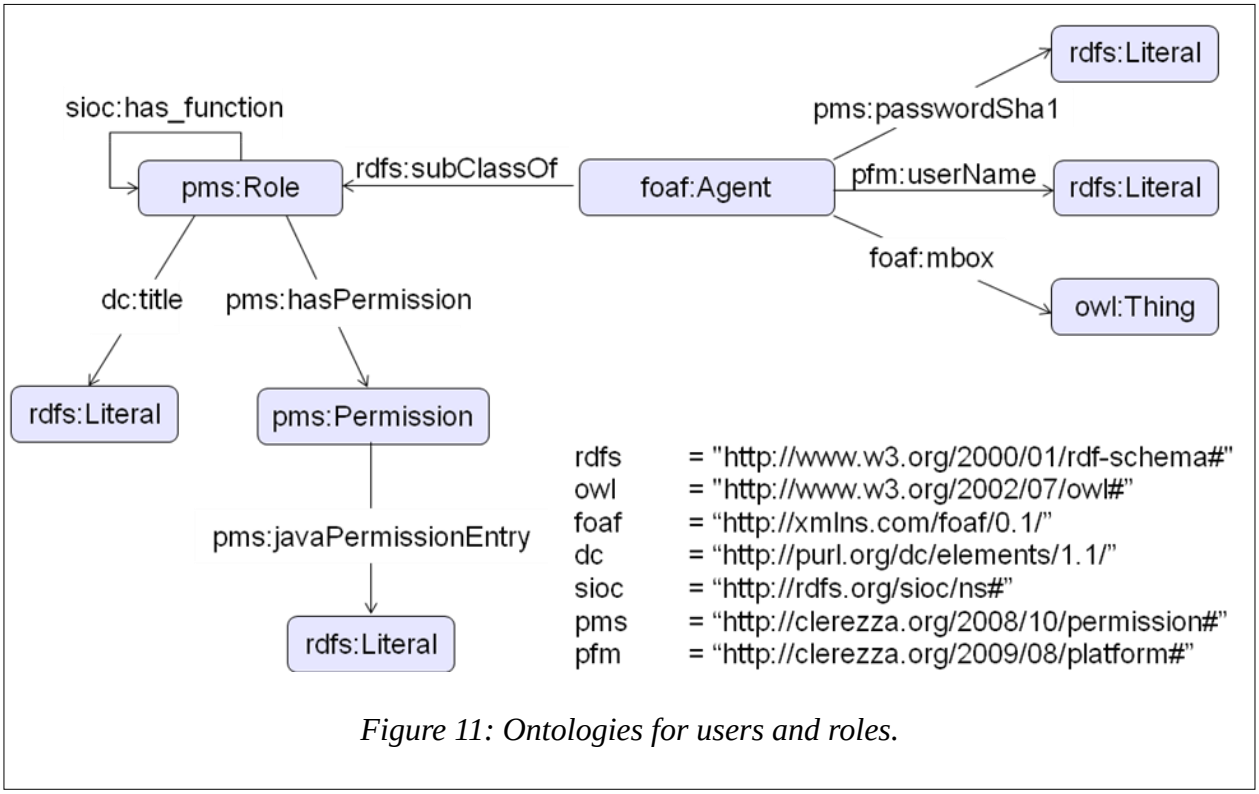

Figure 11: Ontologies for users and roles.

SCSS pre-defines two roles: base-role and default-role. The base-role has a set of permissions which allow a user having this role to use services made public by SCSS. This requires read access to various graphs and the following Java permission specification: (java.util.PropertyPermission "*" "read") and (org.osgi.framework.AdminPermission "*" "resource"). Any user implicitly has the base-role. A user with a default-role has additionally the permissions to change her password and access her account control panel to view or modify her account data. Besides roles, SCSS also pre-defines two users: anonymous and admin. The user anonymous has only the base-role, whereas the user admin has all permissions.

7 Concepts

Concepts are keywords which are used to tag (annotate) resources. The SKOS [10] ontology is widely used to express concepts and their relations, such as broader and narrower. The synopsis in [10] tells how concepts are described: "Using SKOS, concepts can be identified using URIs, labeled with lexical strings in one or more natural languages, assigned notations (lexical codes), documented with various types of note, linked to other concepts and organized into informal hierarchies and association networks, aggregated into concept schemes, grouped into labeled and/or ordered collections, and mapped to concepts in other schemes." The Concept Manager in SCSS provides a service to define and relate concepts based on SKOS. Furthermore, it also allows resources to be tagged with concepts. Table 6 summarizes main functions supported by the Concept Manager.

Table 6: Main functions supported by the Concept Manager.

Function	Description
Create Concept Scheme	Create a concept scheme with the specified label and description, and return the URI reference identifying the concept scheme.
Delete Concept Scheme	Delete a concept scheme identified by the specified URI reference.

Create Concept	Create a concept with the specified preferred label, a list of alternative labels, and description; assign the concept to the specified concept scheme, and return the URI reference identifying the concept.
Delete Concept	Delete a concept identified by the specified URI reference.
Link Concepts	Link two concepts with the specified predicate.
Remove Link	Remove the specified link between two concepts identified by the specified URI references.
Remove All Links	Remove all links between two concepts identified by the specified URI references.
Annotate Resource	Annotate the specified resource with the specified list of concepts.
Delete Annotations	Remove the annotations from the specified resource. Annotations to be removed are identified by the specified list of concepts.

8 Content Sharing

The Content Sharing component allows an SCI node to share contents with other SCI nodes in a Peer-to-Peer fashion. Shared contents are indexed and stored in a distributed manner among a set of co-operating SCI nodes. To obtain a **scalable** indexing of shared contents, this component implements a distributed hash table (DHT).

Shared contents consist of sharable content units. Each content unit is represented by a GraphNode. The node in this GraphNode identifies the content unit and has a globally unique URI reference. The graph in this GraphNode contains triples which constitute the content unit. The structure of a content unit is application specific. For example, in Knowledge Sharing System (KSS) scenario presented in D2.1 [3], a content unit is a knowledge unit whose structure is defined by the KSS.

The URI reference of a content unit, i.e. the URI reference of the node in the GraphNode of a content unit, is used to calculate the index for the DHT. Since content units are annotated with concepts, searching a content unit in SCI should not only be based on the content unit's URI reference, but also on concepts. A simple solution for concept-based searching of content is a two level search. First, with the hash of a concept as the index in a DHT, find a list of URI references of content units tagged with this concept. The second step is the search of the content unit using the found URI references. This solution is not efficient, especially if multiple keywords are used in the search. SciMantic partners are looking for a more efficient solution within the next project period.

Table 7 lists the main functions supported by the Content Sharing component.

Table 7: Main functions supported by the Content Sharing component.

Function	Description
Share Content	Share the content unit represented by the specified GraphNode.
Search Content By URI	Search a content unit identified by the specified URI reference and return a GraphNode describing the content unit if found.
Search Content By Concepts	Search content units annotated by the specified concepts and return a list of URI references identifying the content units found.

9 Summary

Based on Deliverable D2.1 (Functional Requirements and Analysis of Mechanisms for a Semantic Content Infrastructure) the architecture for a secure and scalable Semantic Content Infrastructure (SCI) has been designed. SCI comprises a network of co-operating nodes, where each node may join and leave the network at any time. Each SCI node manages its own semantically linked contents as well as contents shared among the nodes in SCI. To perform this function, an SCI node runs an instance of a Semantic Content Sharing System (SCSS), whose architecture is described in detail in this deliverable. The main components of the SCSS architecture include a Smart Content Binding layer for accessing triple stores in a generic manner, a RESTful Web Service Framework, Security (Authentication and Authorization), Content Sharing (including distributed indexing), User Management, and Concept Management. The architecture is based on OSGi allowing modularization and service-oriented design. An Apache Project called Clerezza has been initiated by Trialox to develop functionality required by SCSS.

References

[1] Aduna B.V.: *User Guide for Sesame 2.3*; 2010, URL: http://www.openrdf.org/doc/sesame2/users/.

[2] Marc Hadley, Paul Sandoz: *JAX-RS: Java API for RESTful Web Services*; Version 1.1, September 2009.

[3] Hasan (Edt.): Functional Requirements and Analysis of Mechanisms for a Semantic Content Infrastructure; Deliverable D2.1, Project SciMantic, March 2010.

[4] Jena Development Team: *Jena – A Semantic Web Framework for Java*; 2010, URL: http://jena.sourceforge.net/documentation.html

[5] OASIS: *SCA Service Component Architecture: Assembly Model Specification*; SCA Version 1.00, March 2007.

[6] Oracle Corporation: *Java™ Authentication and Authorization Service (JAAS) Reference Guide*; 2006, URL: http://java.sun.com/javase/6/docs/technotes/guides/security/jaas/JAASRefGuide.html

[7] The OSGi Alliance: *OSGi Service Platform Core Specification*; Release 4, Version 4.2, June 2009.

[8] The OSGi Alliance: *OSGi Service Platform Service Compendium*; Release 4, Version 4.2, August 2009.

[9] W3C: *Resource Description Framework (RDF): Concepts and Abstract Syntax*; W3C Recommendation, February 2004.

[10] W3C: *SKOS Simple Knowledge Organization System Primer*; W3C Working Group Note, August 2009.

[11] W3C: *SPARQL Query Language for RDF*; W3C Recommendation, 2008.

Acknowledgments

The authors would like to thank all participants of the SciMantic project who have contributed to this deliverable through constructive discussions and feedback. This project is funded by The Innovation Promotion Agency CTI of the Swiss Federal Office for Professional Education and Technology (Das Bundesamt für Berufsbildung und Technologie BBT).

