



OKKAM – Enabling a Web Of Entities

Grant Agreement No. 215032

D5.7: OKKAM developers guide

Document Number	D5.7
Document Title	OKKAM developers guide
Version	5
Status	Working
Work Package	WP5
Deliverable Type	Report
Contractual Date of Delivery	M30
Actual Date of Delivery	M30
Responsible Unit	EPFL
Contributors	All partners
Keyword List	
Dissemination level	PU

Change History

Version	Date	Status	Author (Company)	Description
1	20/5/2010	Draft	Zoltan Miklos (EPFL)	Initial version
2	11/6/2010	Draft	Daniel Giacomuzzi (UNITN), Zoltan Miklos, Michele Catasta (EPFL)	Input from developers
3	29/6/2010	Draft	Hristo Koshutanski (UMA), Ernesto J. Pérez (UMA), Antonio Maña (UMA), Zoltan Miklos, Michele Catasta (EPFL)	Further input included
4	9/7/2010	Draft	George Giannakopoulos, Stefano Bortoli (UNITN, Trento), John O'Flaherty (MAC), Zoltan Miklos (EPFL), Hristo Koshutanski (UMA), Ernesto J. Pérez (UMA), Antonio Maña (UMA),	Further input from developers
5	15/7/2010	Final	Hristo Koshutanski (UMA), Ernesto J. Pérez (UMA), Zoltan Miklos, Heiko Stoermer, Stefano Bortoli (UNITN, Trento),	Final version

Executive Summary

In this page, the executive summary of the document (one page maximum) is reported. The executive summary includes a brief introduction to the document, the results and, eventually, how they were achieved.

Table of Contents

1. INTRODUCTION.....	6
1.1. WHAT IS OKKAM?.....	6
1.2. THE SCOPE OF THE DEVELOPERS GUIDE	6
1.3. DOCUMENT STRUCTURE.....	7
2. PUBLIC OKKAM API.....	8
2.1. ENS WEB SERVICES API	8
2.2. ENS RESTFUL SERVICES	16
2.2.1. Search Service.....	16
2.2.2. Entity Service.....	18
2.3. APPLICATION SERVICES API	19
2.3.1. Named Entities extraction.....	19
2.3.2. Named entities with Okkam Entities (from plain text)	20
2.3.3. Named entities with Okkam Entities (from URL).....	20
2.3.4. Okkam Queries	20
2.3.5. RDFa enrichment.....	21
2.3.6. OKKAMize Web Source.....	21
2.4. BULK ENTITY FINDER API	22
2.4.1. Input File	22
2.4.2. Output File.....	22
2.4.3. The service.....	23
3. INTERNAL OKKAM API.....	24
3.1. OKKAM CORE API.....	24
3.2. STORAGE API.....	25
3.2.1. Functionalities	25
3.2.2. Examples.....	27
3.2.3. Functionality.....	29
3.2.4. Example	31
3.3. MATCHING API	33
3.3.1. Requesting Entities	33
3.3.2. Result format.....	36
3.4. ENTITY LIFECYCLE API	37
3.4.1. Creating a New Entity.....	37
3.4.2. Entity Evolution	39
3.4.3. Entity Delete	42
3.4.4. Entity Get.....	42
3.4.5. Other methods.....	43
3.5. SECURE ACCESS TO PROTECTED ENS APIS	43
3.5.1. Security proxy	43
3.5.2. Accessing protected ENS APIs.....	45
3.5.3. Security-specific APIs.....	48
3.5.4. Certificate revocation	51
3.5.5. Using OKKAM security proxy inside java code.....	52
3.5.6. How to run security proxy with already registered certificate data (P12 format).....	53
4. CONCLUSIONS AND FUTURE PLANS.....	55
5. REFERENCES.....	56
6. APPENDIX A: HOW TO CREATE AN ENTITY?.....	57
6.1. HOW TO CREATE AN ENTITY BY HAND?	57

6.2. HOW TO CREATE AND ENTITY THROUGH THE APIS?..... 58

7. APPENDIX B: ENTITY PROFILE XML EXAMPLE 60

Glossary

OKKAM	IST 7 th Framework Research Project
ENS	Entity Name System (developed by the OKKAM project)
EPFL	Ecole Polytechnique Fédérale de Lausanne, Swiss Federal Institute of Technology
UNITN	University of Trento (Italy)
MAC	National Microelectronic Application Center (Ireland)
DERI	Digital Enterprise Research Institute (Ireland)
ECSSE	Entity Centric Semantic Search Engine
UMA	University of Malaga (Spain)
API	Application Programming Interface
URL	Uniform Resource Locator
XML	Extensible Markup Language
RDF	Resource Description Framework
RDFa	Resource Description Framework in attributes
WSDL	Web Service Description Language
UTF-8	Unicode Transformation Format (8 bit)
Sindice	Sindice is a research project at DERI

1. Introduction

1.1. What is OKKAM?

OKKAM is a Large-Scale Integrating Project funded by the European Commission under the 7th Framework Program (FP7) until **June 2010**.

The **OKKAM** project aims at *enabling* the **Web of Entities**, namely a virtual space where any collection of data and information about any type of entities (e.g. people, locations, organizations, events, products, ...) published on the Web can be integrated into a single virtual, decentralized, open knowledge base (like the Web did for hypertexts.)

OKKAM will contribute to this vision by supporting the convergence towards the use of a *single and globally unique identifier* for any entity which is named on the Web. The intuition of the project is that the concrete realization of the Web of Entities requires that we enable tools and practices for cutting to the root the proliferation of unnecessary new identifiers for naming the entities which already have a public identifier (the **OKKAM's razor**). Therefore, **OKKAM** will make available to content creators, editors and developers a global infrastructure and a collection of new tools and plugins which support them to easily find public identifiers for the entities named in their contents/services, use them for creating annotations, build new network-based services which make essential use of these identifiers in an open environment (like the Web or large Intranets).

To realize this vision, **OKKAM** proposes the following roadmap:

- Providing a scalable and sustainable infrastructure, called the **Entity Name System (ENS)**, for making the systematic reuse of global and unique entity identifiers not only possible, but easy and straightforward. The ENS will be a distributed service which permanently stores identifiers for entities and provides a collection of core services (e.g. entity matching, ID mapping and resolution) needed to support their pervasive reuse;
- bootstrapping and enabling the fast growth of Web of Entities by fostering the creation of **OKKAMized content** (i.e. content where entities are named or annotated with OKKAM IDs) in **OKKAM-empowered applications** (i.e. applications which can interact with the ENS for getting and reusing identifiers);
- showcasing the benefits of enabling the Web of Entities and, more in general, of an entity-oriented approach to content and knowledge management by building **relevant applications** on top of the new infrastructure in three important areas: *information retrieval and semantic search*, *content authoring* (more specifically, in *scientific publishing* and *news production*) and *organizational knowledge management*.

More up-to-date information can be found on the OKKAM webpage: <http://www.okkam.org> and on the community website <http://community.okkam.org> .

1.2. The scope of the developers guide

This deliverable gives an overview of the OKKAM ENS APIs, for developers. First we describe the APIs which are available for applications to consume the services of the ENS, and then we describe the internal APIs. The description of the external API is intended to help developers writing

OKKAM related applications, for example retrieving unique identifiers or extracting entities from a text source.

The current version of software supports both Web-service style and REST-style communication for accessing the ENS functionality. The Application Services API, which enables to extract entities from documents or Web-resources, also foresees communication via Web Services. Examples and tutorials how to use the services is available online for our users. We also included a chapter in this deliverable on Sindice, in particular the description of APIs related to OKKAM for realizing an ECSSE (Entity-centric Semantic Search Engine).

The internal APIs include the storage-, the matching-, the lifecycle, the security and the OKKAM core APIs. The document is not intended to be the documentation of the APIs, rather a developer's guide, which helps to orient in the code.

The deliverable is complemented by further documentations, available on the OKKAM community portal <http://community.okkam.org>

1.3. Document Structure

The document structured as follows:

In Chapter 2, we present the public OKKAM APIs, and the related APIs of Sindice ECSSE. Chapter 3, gives an overview of the internal OKKAM APIs, in particular, Storage-, Matching-, Entity Lifecycle- Security, and OKKAM core APIs. The annex contains basic guidelines for commenting and documenting the code developed by OKKAM partners. An organic part of this deliverable is the javadoc documentation of the internal APIs, which is available at <http://community.okkam.org>

2. Public Okkam API

The ENS stores unique identifiers and offers services to applications to retrieve them. The ENS services are available via Web Services. This API is described in the following sections, together with some reference to the examples. The Application Services API offers a service to automatically extract entities from documents and Web resources. These services are described in the following sections. We also describe the OKKAM relevant APIs of the Sindice ECSSE. ¹

Examples and tutorials for creating entities both independently from programming language and for java programmers are available at <http://www.okkam.org/apis/tutorials/create-entity>.

2.1. ENS Web Services API

Developers should refer to the following information for enabling access to ENS WS APIs:

1. Accessing and conforming to input messages of WS APIs according to the following WSDL description <http://api.okkam.org/okkam-core/WebServices?wsdl>

Developers should use standard WS libraries for automatic construction of corresponding function calls and their required input messages. In this section we cover XML data structures of input/output messages of ENS APIs.

2. Defining input data structures for WS APIs from inside Java-based systems is described in sections 3.4.1 – 3.4.4 and the following documentation on the community portal: <http://community.okkam.org/index.php/Documentation/APIs/java-okkam-client.html>
3. Accessing protected WS APIs from Java-based and non-java based systems through the Okkam proxy is described in Section 3.5.

WS APIs evolution. WS APIs will evolve beyond project lifetime with explicit support for native message data types (non-java based) integration into software development, such as for .Net. Next evolution of ENS WS APIs will handle direct input/output data structures on the level of WS APIs, as part of WSDL description, so that any standard WS library will parse the WSDL and automatically generate respective data objects native for a current programming language (which supports WS). In this way we avoid any additional conversion (marshalling/unmarshalling of XML data) of input/output message formats and free the APIs to any WS client.

In the following we describe the available WS APIs.

Operation "getEntity"

¹ We decided to include some details about Sindice APIs in this document, because they are already implemented and functional and they might play an important role to populate the OKKAM repository.

Input type:

```
<xs:complexType name="getEntity">
  <xs:sequence>
    <xs:element name="oid" type="xs:string" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
```

Response type:

```
<xs:complexType name="getEntityResponse">
  <xs:sequence>
    <xs:element name="return" type="xs:string" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
```

Fault message name="OkkamCoreException"

The 'return' element contained in the response 'getEntitiesResponse' complex type must be interpreted as a serialized version of the XML code describing an Entity object as described in Appendix B: Entity Profile XML example

Operation "getEntities"**Input type:**

```
<xs:complexType name="getEntities">
  <xs:sequence>
    <xs:element name="oids" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

Response type:

```
<xs:complexType name="getEntitiesResponse">
  <xs:sequence>
    <xs:element name="return" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

Fault message name="OkkamCoreException"

The 'return' element contained in the response 'getEntitiesResponse' complex type must be interpreted as a serialized version of the XML code describing an Entity object as described in Appendix B: Entity Profile XML example.

Operation "lockEntity"**Input type:**

```
<xs:complexType name="lockEntity">
  <xs:sequence>
    <xs:element name="oid" type="xs:string" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
```

Response type:

```
<xs:complexType name="lockEntityResponse">
  <xs:sequence>
    <xs:element name="return" type="xs:string" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
```

```
</xs:sequence>  
</xs:complexType>
```

Fault message name="OkkamAuthorizationException"

Fault message name="OkkamCoreException"

Operation “unlockEntity”

Input type:

```
<xs:complexType name="unlockEntity">  
  <xs:sequence>  
    <xs:element name="oid" type="xs:string" minOccurs="0"/>  
    <xs:element name="ticket" type="xs:string" minOccurs="0"/>  
  </xs:sequence>  
</xs:complexType>
```

Response type: none

Fault message name="OkkamAuthorizationException"

Fault message name="OkkamCoreException"

Operation “updateEntity”

Input type:

```
<xs:complexType name="updateEntity">  
  <xs:sequence>  
    <xs:element name="ticket" type="xs:string" minOccurs="0"/>  
    <xs:element name="certificate" type="xs:string" minOccurs="0"/>  
  </xs:sequence>  
</xs:complexType>
```

Response type: none

Fault message name="OkkamAuthorizationException"

Fault message name="OkkamCoreException"

The ‘certificate’ element part of the input complex type ‘updateEntity’ must be interpreted as a simple string certifying that the entity submitted for update is compliant with the quality constraints defined by the ENS. This ‘certificate’ is not related with the security setting necessary to the access the ENS functionalities.

Operation “deleteEntity”

Input type:

```
<xs:complexType name="deleteEntity">  
  <xs:sequence>  
    <xs:element name="oid" type="xs:string" minOccurs="0"/>  
    <xs:element name="ticket" type="xs:string" minOccurs="0"/>  
  </xs:sequence>  
</xs:complexType>
```

Response type: none

Fault message name="OkkamAuthorizationException"

Fault message name="OkkamCoreException"

Operation "splitEntity"

Input type:

```
<xs:complexType name="splitEntity">
  <xs:sequence>
    <xs:element name="oid" type="xs:string" minOccurs="0"/>
    <xs:element name="splitEntities" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="ticket" type="xs:string" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
```

Response type:

```
<xs:complexType name="splitEntityResponse">
  <xs:sequence>
    <xs:element name="return" type="nsl:NewEntityResultClient" minOccurs="0"
maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="NewEntityResultClient">
  <xs:sequence>
    <xs:element name="newEntityURI" type="xs:string" minOccurs="0"/>
    <xs:element name="similarEntities" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

Fault message name="OkkamAuthorizationException"

Fault message name="OkkamCoreException"

The 'splitEntities' string parameter of the complex type 'splitEntity' must be interpreted as a serialized version of the XML code describing an Entity object as described in Appendix B: Entity Profile XML example. The 'similarEntities' string response part the 'NewEntityResultClient' complex type must also be interpreted as a serialized version of the XML code describing an Entity object as described in Appendix B: Entity Profile XML example.

Operation "mergeEntities"

Input type:

```
<xs:complexType name="mergeEntities">
  <xs:sequence>
    <xs:element name="oids" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="mergedEntity" type="xs:string" minOccurs="0"/>
    <xs:element name="tickets" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

Response type:

```
<xs:complexType name="mergeEntitiesResponse">
  <xs:sequence>
```

```

        <xs:element name="return" type="ns1:NewEntityResultClient" minOccurs="0"/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="NewEntityResultClient">
    <xs:sequence>
        <xs:element name="newEntityURI" type="xs:string" minOccurs="0"/>
        <xs:element name="similarEntities" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>

```

Fault message name="OkkamAuthorizationException"

Fault message name="OkkamCoreException"

The ‘mergedEntity’ string parameter of the complex type ‘mergeEntities’ must be interpreted as a serialized version of the XML code describing an Entity object as described in Appendix B: Entity Profile XML example. The ‘similarEntities’ string response part the ‘NewEntityResultClient’ complex type must also be interpreted as a serialized version of the XML code describing an Entity object as described in Appendix B: Entity Profile XML example.

Operation “validateEntity”

Input type:

```

<xs:complexType name="validateEntity">
    <xs:sequence>
        <xs:element name="entity" type="xs:string" minOccurs="0"/>
        <xs:element name="ignoreDuplicates" type="xs:boolean"/>
    </xs:sequence>
</xs:complexType>

```

Response type:

```

<xs:complexType name="validateEntityResponse">
    <xs:sequence>
        <xs:element name="return" type="ns1:EntityValidationReport" minOccurs="0"/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="EntityValidationReport">
    <xs:sequence>
        <xs:element name="candidateDuplicates" type="xs:string" nillable="true" minOccurs="0"
maxOccurs="unbounded"/>
        <xs:element name="certificate" type="xs:string" minOccurs="0"/>
        <xs:element name="permission" type="xs:string" minOccurs="0"/>
        <xs:element name="qualityReport" type="tns:EntityQualityReport" minOccurs="0"/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="EntityQualityReport">
    <xs:sequence>
        <xs:element name="badQualityAttributes" type="tns:AttributeQualityReport" nillable="true"
minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="brokenRules" type="xs:string" nillable="true" minOccurs="0"
maxOccurs="unbounded"/>
        <xs:element name="qualityCheckPassed" type="xs:boolean"/>
        <xs:element name="qualityEvaluation" type="xs:double"/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="AttributeQualityReport">
    <xs:sequence>
        <xs:element name="attributeIndex" type="xs:int"/>
        <xs:element name="brokenRules" type="xs:string" nillable="true" minOccurs="0"
maxOccurs="unbounded"/>

```

```

<xs:element name="message" type="xs:string" minOccurs="0"/>
<xs:element name="originalAttributeName" type="xs:string" minOccurs="0"/>
<xs:element name="originalAttributeValue" type="xs:string" minOccurs="0"/>
</xs:sequence>
</xs:complexType>

```

Fault message name="OkkamAuthorizationException"

Fault message name="OkkamCoreException"

The ‘entity string parameter of the complex type ‘validateEntity’ must be interpreted as a serialized version of the XML code describing an Entity object as described in Appendix B: Entity Profile XML example. The ‘candidateDuplicates’ string response part the ‘EntityValidationReport’ complex type must also be interpreted as a serialized version of the XML code describing an Entity object as described in Appendix B: Entity Profile XML example.

The ‘certificate’ element part of the input complex type ‘EntityValidationReport’ must be interpreted as a simple string certifying that the entity submitted for update is compliant with the quality constraints defined by the ENS. The value of this parameter must be used as input of the ‘create/update’ operations. This ‘certificate’ is not related with the security setting necessary to the access the ENS functionalities. Similarly, the ‘permission’ element part of the ‘EntityValidationReport’ must be interpreted as a Boolean value that is set to ‘true’ if the serialized entity object submitted as parameter passed the validation tests, or it is set to ‘false’ if the serialized entity submitted failed to pass one of the validation tests. The element ‘permission’ is not related with the security setting necessary to the access the ENS functionalities.

The ‘brokenRules’ elements are meant do contain strings representing the names of the quality rules not respected by the entity object submitted or, more in details, by some of its attributes.

Operation “lockEntities”

Input type:

```

<xs:complexType name="lockEntities">
  <xs:sequence>
    <xs:element name="oids" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

```

Response type:

```

<xs:complexType name="lockEntitiesResponse">
  <xs:sequence>
    <xs:element name="return" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

```

Fault message name="OkkamAuthorizationException"

Fault message name="OkkamCoreException"

Operation “getTypeTemplate”

Input type:

```
<xs:complexType name="getTypeTemplate">
  <xs:sequence>
    <xs:element name="type" type="xs:string" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
```

Response type:

```
<xs:complexType name="getTypeTemplateResponse">
  <xs:sequence>
    <xs:element name="return" type="xs:string" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
```

Fault message name="OkkamCoreException"

The ‘type’ element of the complex type ‘getTypeTemplate’ must be a string representing one of the entity types supported by the ENS (e.g. person, organization, etc.). The ‘return’ element of the complex type ‘getTypeTemplateResponse’ must be interpreted as a serialized version of the XML code describing an Entity object as described in Appendix B: Entity Profile XML example. This entity objects contains as attribute names, the default attribute names for the chosen entity type, and in the attribute values are available descriptions for the corresponding attribute names.

Operation “getAlternativeIds”

Input type:

```
<xs:complexType name="getAlternativeIds">
  <xs:sequence>
    <xs:element name="oid" type="xs:string" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
```

Response type:

```
<xs:complexType name="getAlternativeIdsResponse">
  <xs:sequence>
    <xs:element name="return" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

Fault message name="OkkamCoreException"

Operation “findEntity”

Input type:

```
<xs:complexType name="findEntity">
  <xs:sequence>
    <xs:element name="query" type="xs:string" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
```

Response type:

```
<xs:complexType name="findEntityResponse">
  <xs:sequence>
    <xs:element name="return" type="ns2:MatchingCandidate" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="MatchingCandidate">
  <xs:sequence>
```

```

<xs:element name="oid" type="xs:string" minOccurs="0"/>
<xs:element name="sim" type="xs:double"/>
<xs:element name="XML" type="xs:string" minOccurs="0"/>
</xs:sequence>
</xs:complexType>

```

Fault message name="OkkamCoreException"

MatchingCandidate is a list of matching entities and their relevance (closeness) where:

XML: a string containing URL-encoded XML that represents an entity. In Java we use JAXB² to convert an Entity to XML and backwards.

oid: the Okkam ID, aka ENS ID or ENS Identifier, of the entity given in URL-encoded XML above

sim: similar measure, closeness, etc. A number <1 that describe the quality of a result.

Operation "createNewEntity"

Input type:

```

<xs:complexType name="createNewEntity">
  <xs:sequence>
    <xs:element name="certificate" type="xs:string" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>

```

Response type:

```

<xs:complexType name="createNewEntityResponse">
  <xs:sequence>
    <xs:element name="return" type="xs:string" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>

```

Fault message name="OkkamAuthorizationException"

Fault message name="OkkamCoreException"

The 'certificate' element part of the input complex type 'EntityValidationReport' must be interpreted as a simple string certifying that the entity submitted for update is compliant with the quality constraints defined by the ENS. The 'return' element of the complex type 'createNewEntityResponse' contains the URI for the newly created entity.

Operation "getOidsByAlternativeId"

Input type:

```

<xs:complexType name="getOidsByAlternativeId">
  <xs:sequence>
    <xs:element name="alternativeId" type="xs:string" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>

```

Response type:

```

<xs:complexType name="getOidsByAlternativeIdResponse">
  <xs:sequence>
    <xs:element name="return" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

```

² <https://jaxb.dev.java.net>

```
</xs:sequence>
</xs:complexType>
```

Fault message name="OkkamCoreException"

Operation "logSelectedEntity"

Input type:

```
<xs:complexType name="logSelectedEntity">
  <xs:sequence>
    <xs:element name="oid" type="xs:string" minOccurs="0"/>
    <xs:element name="query" type="xs:string" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
```

Response type: none

Fault message: none

2.2. ENS restful services

The restful services works exactly like the soap web services (see <http://www.okkam.org/apis/web-service-api>), the difference is the architecture that we use to transfer the data from the server to the client

2.2.1. Search Service

Service address: :

<http://api.okkam.org/okkam-core/jsearch>

Input

- *q* : a string with your query. It can either contain keywords, e.g. "Paolo Bouquet", or a structured query like "QUERY{firstname=paolo}METADATA{entityType=person}"

This call will return a JSONArray object with all the entities that match the input query

The array is a list of 3 JSONObject:

- *o* : the okkamID (String)
- *p* : probability
- *e* : [an entity object, see below]

Example:

```
[{"e":{"entity_object},"p":0.5742205904305446,"o":"http://www.okkam.org/ens/idd3d18304-
```

```

570f-4437-b57a-709f27a632f0" },
— {"e":{entity_object},"p":0.5040192033630743,"o":"http://www.okkam.org/ens/idb0d21218-
57cd-4405-9560-1ba6828a0872"},
— ... ]
—

```

ENTITY object:

- *al*: an array of alternative ids (JSONArray)
- *e*: an array of equivalent ids (JSONArray)
- *p*: preferred id (String)
- *o*: okkam id
- *pr*: entity profile (JSONObject)
 - *s*: semantic type (String)
 - *r*: references (JSONArray)
 - *c*: reference type (String)
 - *p*: the url of the reference (String)
 - *a*: attributes (JSONArray)
 - *v*: value (String)
 - *vi*: the okkam id of the attribute (e.g. the ID of European Union if the value is "UE")
 - *n*: the prefix of the attribute

Example

```

— {"al":["http://sws.geonames.org/6251999/"],
— "e":[],
— "p":"http://www.okkam.org/ens/idd3d18304-570f-4437-b57a-709f27a632f0",
— "o":"http://www.okkam.org/ens/idd3d18304-570f-4437-b57a-709f27a632f0",
— "pr":{
—         "s":"location",
—
— "r":[{"c":"application/rdf+xml","p":"http://www.geonames.org/6251999/about.rdf"}],
—         "a":[
—
—         {"v":"Canada","vi":null,"n":"http://www.geonames.org/ontology#name"},
—
—         {"v":"60.0000000","vi":null,"n":"http://www.w3.org/2003/01/geo/wgs84_p
os#latitude"}],

```

```

—         [...]
—         ]
—         }
—    }
—

```

2.2.2. Entity Service

Service address:

<http://api.okkam.org/okkam-core/jentity>

Input

- *oid* : the okkam id of the entity to load

This call will return a JSONObject with the following structure:

ENTITY object:

- *al*: an array of alternative ids (JSONArray)
- *e*: an array of equivalent ids (JSONArray)
- *p*: preferred id (String)
- *o*: okkam id
- *pr*: entity profile (JSONObject)
 - *s*: semantic type (String)
 - *r*: references (JSONArray)
 - *c*: reference type (String)
 - *p*: the url of the reference (String)
 - *a*: attributes (JSONArray)
 - *v*: value (String)
 - *vi*: the okkam id of the attribute (e.g. the ID of European Union if the value is "UE")
 - *n*: the prefix of the attribute

Example:

```

—    {"al":["http://sws.geonames.org/6251999/"],
—      "e":[],
—      "p":"http://www.okkam.org/ens/idd3d18304-570f-4437-b57a-709f27a632f0",
—      "o":"http://www.okkam.org/ens/idd3d18304-570f-4437-b57a-709f27a632f0",

```

```

—  "pr":{
—      "s":"location",
—
—  r":[{"c":"application/rdf+xml","p":"http://www.geonames.org/6251999/about.rdf"}],
—      "a":[
—
—        {"v":"Canada","vi":null,"n":"http://www.geonames.org/ontology#name"},
—
—        {"v":"60.0000000","vi":null,"n":"http://www.w3.org/2003/01/geo/wgs84_p
os#latitude"},
—
—          [...]
—          ]
—      }
—  }
—

```

2.3. Application Services API

The Application Services API allows that applications automatically extract the following elements from a text document:

- the named entities like People, Companies, Location, and other
- the OKKAM entities
- the OKKAM core queries generated with the extraction of the named entities
- the RDFa enrichment
- Asynch process of urls to generate new OKKAM entities

WSDL URL: <http://host.expertsystem.it/OkkamWebService/services/OkkamWebService?wsdl>

2.3.1. Named Entities extraction

The API call is as follows:

```
String highlightTextWithOkkamQueries(String activationKey, int language, String plainText)
```

where:

- *activationKey* is the API access key;
- *language* 0=Italian, 1=English

- *plainText* represent the content to be analyzed

The result is in XML format and contains the extracted named entities and the queries that allow you to execute OkkamCore calls.

2.3.2. Named entities with Okkam Entities (from plain text)

The API call is as follows:

```
String highlightTextWithOkkamResults (String activationKey, int language, String plainText)
```

where:

- *activationKey* is the API access key;
- *language* 0=Italian, 1=English
- *plainText* represent the content to be analyzed

The result is in XML format and contains the extracted named entities, the queries that allow you to execute OkkamCore calls and the result of the OkkamCore calls.

2.3.3. Named entities with Okkam Entities (from URL)

The API call is as follows:

```
String okkamizeUrl (String activationKey, int language, String url)
```

where:

- *activationKey* is the API access key;
- *language* 0=Italian, 1=English
- *url* represent the url to be analyzed

The result is in XML format and contains the extracted named entities, the queries that allow you to execute OkkamCore calls and the result of the OkkamCore calls.

2.3.4. Okkam Queries

The API call is as follows:

```
String okkamQueries (String activationKey, int language, String plainText)
```

where:

- *activationKey* is the API access key;
- *language* 0=Italian, 1=English
- *plainText* represent the content to be analyzed

The result is in XML format and contains the queries that allow you to execute OkkamCore calls.

2.3.5. RDFa enrichment

The API call is as follows:

```
String microformatTextEnricher (String activationKey, int language, String plainText)
```

where:

- *activationKey* is the API access key;
- *language* 0=Italian, 1=English
- *plainText* represent the content to be analyzed

The result is the plain text converted in RDFa standard.

2.3.6. OKKAMize Web Source

Start process

The API call is as follows:

```
String okkamizeWebSource (String activationKey, int language, String url, int depth, String[] inclusion, String[] exclusion)
```

where:

- *activationKey* is the API access key;
- *language* 0=Italian, 1=English
- *url* is the crawler start point
- *depth* of the crawler
- *inclusion* represents a list of the included URLs
- *exclusion* represents a list of the excluded URLs

The result is in XML format and contains the operation identifier, the operation status and a message. Below is an example of a possible response:

```
<?xml version="1.0" encoding="UTF-8"?>
<okkamResponse operationId="c30e4991ae434ee39d0294a08f5889c5" status="running">
  <message><![CDATA[operation started]]></message>
</okkamResponse>
```

Operation status

The function to determine the operation status is

```
String getOperationStatus (String activationKey, String operationId)
```

where:

- *activationKey* is the API access key;
- *operationId* is the identifier

The result is in XML format and it contains the operation identifier and the operation. Below is an example:

```
<?xml version="1.0" encoding="UTF-8"?>
  <okkamResponse operationId="c30e4991ae434ee39d0294a08f5889c5" status="running" />
```

Operation result

The operation result can be obtained as follows:

String getOperationResult (String activationKey, String operationId)

where:

- *activationKey* is the API access key;
- *operationId* is the identifier

The result is in XML format and it contains the local machine path where the extracted entities were saved in a defined format that can be inserted into the Okkam Repository.

2.4. Bulk Entity Finder API

The bulk finder service allows uploading a file containing many findEntity queries to an ftp server, and retrieving the result from the server when all the queries have been processed.

The procedure to use the service is the following:

- One has to upload a file containing queries to an ftp server,
- send the ftpUrl (with credentials if required) to the submit service
- Use the status service with ftpUrl to check progress
- Retrieve the processed file from the ftp server
- If required use the cancel service to cancel processing of a file.

The service is available at <http://okkam.sindice.com/bulk/finder>

For using each of the services, the *ftpUrl* parameter is required.

2.4.1. Input File

The Input file is a text file, which should contain one query per line. One has to use UTF-8 file encoding if there are any special characters contained. A sample input file is available at http://okkam.sindice.com/bulk/sample_input.txt

2.4.2. Output File

The output file will be placed onto the ftp server next to the input file with the file extension '.result'. During processing a file with the name '.result.part' may be present on the ftp server. An

example can be found at http://okkam.sindice.com/bulk/sample_results.xml. An other example is available at http://okkam.sindice.com/bulk/sample_results_without_candidates.xml, which is generated by including parameter `omitCandidateXml=true` in request.

2.4.3. The service

The restful service can be consumed using two different formats: the bulk entity finding service can produce html output, which is suitable for human interaction, and also xml output, which is more suitable for automated processing for applications. The three available operations are submit a request, check the status of an operation, cancel the processing.

The server can be configured with passwords for accessing ftp servers if required. Otherwise, one has to upload a file to ftp server and including the ftp credentials in the url.

For each of the operations, the parameter `ftpUrl=ftp://<user:password@>host:port/path/to/query.txt` is required. (username:password credentials are omitted if these have been already configured on the server)

2.4.3.1 Submit

The process can be started as follows:

```
htmlfinder?action=submit&ftpUrl=ftp://username:password@ftp.example.com:2021/my/query.txt  
xmlfinder?action=submit&ftpUrl=ftp://username:password@ftp.example.com:2021/my/query.txt
```

NB: To omitCandidateXml from the result, include the following parameter in the http request: `omitCandidateXml=true`

2.4.3.2 Status

One can obtain the status of the process as follows:

```
htmlfinder?action=submit&ftpUrl=ftp://username:password@ftp.example.com:2021/my/query.txt  
xmlfinder?action=submit&ftpUrl=ftp://username:password@ftp.example.com:2021/my/query.txt
```

2.4.3.3 Cancel

If necessary, the bulk entity finding process can be interrupted as follows:

```
htmlfinder?action=submit&ftpUrl=ftp://username:password@ftp.example.com:2021/my/query.txt  
xmlfinder?action=submit&ftpUrl=ftp://username:password@ftp.example.com:2021/my/query.txt
```

3. Internal OKKAM API

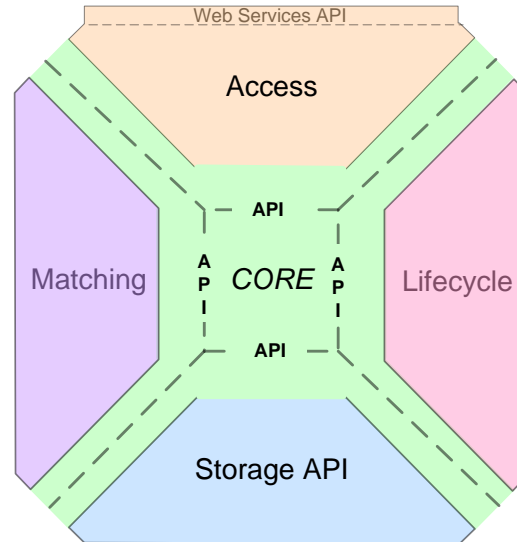


Figure 1 The core of the ENS, and its components

The internal architecture of a single ENS node consists of the following components:

- **Core:** the integrating component that provides interactions between the components (managed by WP5);
- **Matching:** responsible for entity matching (managed by WP3);
- **Access (security):** component responsible for security-related functionality, e.g. access control (managed by UMA, MAC and UNITN in the context of PCA-5 and WP5);
- **Life-cycle:** responsible for everything that concerns the evolution of data in the ENS (managed by WP6);
- **Storage:** responsible for storage of data in the ENS (managed by WP2).

In this chapter we give an overview of the internal OKKAM APIs. This overview should help OKKAM developers to orient in the javadoc documentation, which is available at <http://gforge.okkamdev.org/TechnicalDocs/index.html>, and is an organic part of this deliverable.

3.1. Okkam core API

The internal architecture of a single ENS node consists of the following components: OKKAM core, matching, lifecycle, access and storage. The components are connected via Java interfaces which are defined in the Core. This guarantees compatibility of a component with the overall

architecture at compilation level and prevents ad-hoc interface changes by component providers. The core performs runtime loading of the individual components, so that – in theory – these can be replaced seamlessly without recompilation or code changes of the complete system.

If a component needs to communicate with another one, it requests a channel to this component from the Core. The Core is furthermore responsible for managing sessions that require stateful processes which are exposed over a number of different web services (e.g. for entity creation).

OKKAM core and its components are bound together as a Web Application Archive (WAR) and deployed in an instance of the high-performance application server JBOSS.

The main role of the OKKAM core API is to realize the integration of the individual components.

3.2. Storage API

The goal of the OKKAM storage component is to enable entity persistence and retrieval. The main operations supported by the storage layer are the CRUD set (Create/Read/Update/Delete) plus extensive search capabilities.

3.2.1. Functionalities

3.2.1.1 Entity Creation/Update

"WHAT"

In AnormStore, Creation and Update have the same semantics. In both cases, the index entry for the given Entity is updated (or created, if it has a brand new OKKAM-Id) while a new version is included in the persistence layer. In this way, searching is based only on the latest entity version, while it is still possible to retrieve older versions of the entity.

"HOW"

Entities are represented as XML files which comply to the Entity schema defined in OKKAM-Core. In AnormStore, an entity is both stored and indexed. AnormStore writes the content of an entity as raw bytes into HBase. The OKKAM-Id is the row identifier, being it unique by definition. Auxiliary metadata is computed at insertion time, and stored alongside the entity representation. For indexing an entity, AnormStore parses the XML representation of the entity using the JAXB parser and the Java data-model mapping defined in OKKAM-Core. From the canonical Entity representation, AnormStore builds a Solr document, containing all the indexable fields of an OKKAM entity. Such document is submitted to the Solr master, which is in charge of including it in the Lucene index.

"WHERE"

Entity insertion functionalities are exposed in: `org.okkam.storage.api.legacy.EntityWriter`
In the new API (`org.okkam.storage.api.EntityWriter`) you can find also methods for batch creation/update.

3.2.1.2 Entity Deletion

"WHAT"

Given an OKKAM-Id, AnormStore deletes the related entity and won't include it in future search results.

"HOW"

As for Entity Creation/Update, AnormStore wraps a compound operation in a single method call. delete() is a blocking call which returns only when both the deletion command has been issued to the Solr master and to HBase.

"WHERE"

Entity deletion functionalities are exposed in: org.okkam.storage.api.legacy.EntityWriter
In the new API (org.okkam.storage.api.EntityWriter) you can find also methods for batch deletion.

3.2.1.3 Entity Reading

"WHAT"

Given an OKKAM-Id, AnormStore retrieves the latest stored version of the related entity.

"HOW"

Read operations go directly to HBase, skipping the index component. It's a simple random read, which most of the times returns a cached entry.

"WHERE"

Entity deletion functionalities are exposed in: org.okkam.storage.api.legacy.EntityReader
In the new API (org.okkam.storage.api.EntityReader) you can find also an experimental method for batch reads.

3.2.1.4 Entity Search

"WHAT"

After having submitted a storage query (that follows the Lucene query-language syntax), AnormStore will return a fixed number of tuples, ordered by relevance score. The tuple has the following structure:

<OKKAM-Id, <StorageEntity, score>>.

"HOW"

The given query is executed on one of the Solr replicas, and it returns a list of OKKAM-Ids sorted in order of relevance. With such list, the AnormStore executes a multithreaded read operation on HBase to retrieve the raw content of each entity.

Currently we are using a standard TF-IDF scoring function extended with the Okkam-specific boosting of attributes and values. The number of results to be returned can be specified in the AnormStore configuration file.

"WHERE"

Entity search functionalities are exposed in: org.okkam.storage.api.legacy.EntitySearcher

3.2.1.5 Synonyms and boosting

The storage component provides support for synonyms for attribute names. Moreover, it is easy to manually define boosting factors for particular attribute names. A simple self-explanatory example file (boosting-fields.xml) for the boosting is depicted below:

```
<root>
  <field boost="2.0">name</field>
  <field boost="1.9">city</field>
  <field boost="1.6">value</field>
  ...
</root>
```

A similarly self-explanatory example (of synonym-schema.xml) for defining the synonyms (or multiple-language attribute names) is depicted below:

```
<root>
  <wordGroup>
    <word lang="eng">city</word>
    <word lang="eng">town</word>
    <word lang="french">ville</word>
  </wordGroup>
  <wordGroup>
    <word lang="eng">name</word>
    <word lang="german">name</word>
    <word lang="french">nom</word>
  </wordGroup>
  <wordGroup>
    <word lang="eng">value</word>
    <word lang="french">valeur</word>
  </wordGroup>
</root>
```

3.2.2. Examples

org.okkam.storage.Read and org.okkam.storage.Search are two quick debug utilities that can be used from the command line. Being really simple, they represent a good example of API usage:

```
Read.java
[... ]
EntityReader reader = new EntityReader();
long start = System.currentTimeMillis();
StorageEntity<?, ?, ?, ?> result;
try {
  result = reader.read(okkamId);
} catch (FailedCRUDSOperation e) {
```

```
System.err.println("AnormStore was not able to execute the read task: "
    + e.getMessage());
e.printStackTrace();
throw new RuntimeException();
}
long stop = System.currentTimeMillis();

// presenting the result
System.out.println("\nOKKAM ENTITY\n");
System.out.println(result);
System.out.println("Elapsed time (in ms): " + (stop - start));
[...]
```

```
Search.java
[...]
```

```
AnormQuery query = new AnormQuery(queryString,
EntityType.toEnumMember(entityType));
EntitySearcher searcher = new EntitySearcher();
long start = System.currentTimeMillis();
List<ScoredEntityResult> results;
try {
    results = searcher.search(query);
} catch (FailedCRUDOperation e) {
    System.err.println("AnormStore was not able to execute the search task: "
        + e.getMessage());
    e.printStackTrace();
    throw new RuntimeException();
}
long stop = System.currentTimeMillis();
// reverse the result list to show the first result at the end (avoids
scrolling)
Collections.reverse(results);

// presenting the results
System.out.println("QUERY: " + queryString + "\n");
int i = results.size();
for (ScoredEntityResult result : results) {
    System.out.println("RESULT #" + i + "\n");
    System.out.println(result.getEntity());
    System.out.println("Score: " + result.getScore() + "\n\n");
    i--;
}
System.out.println("Elapsed time (in ms): " + (stop - start));
[...]
```

This section describes the Storage API. The main functionalities of the storage layer are to provide means to insert and to delete entities or to update an entity in the storage and also to enable efficient the querying of the entity store. In the future versions, the storage API will also provide functions which return some statistical information about the stored entities and also procedures which are specifically needed by the Lifecycle component.

3.2.3. Functionality

3.2.3.1 Insertion of an Entity

The current entity representation defines an entity in an XML format. In AnormStore entity is both stored as well as indexed. AnormStore writes the contents of an entity as bytes identified by Okkam ID, into the Voldemort database. For indexing an entity, AnormStore parses the XML representation of the entity using the Xerces SAX parser [1] and creates a document that temporarily stores all index able fields of an entity. This document is then passed to the SolrMaster for indexing. The SolrMaster to be used by AnormStore API is known from the SolrBroker. Certain fields from the XML entity can be blocked for indexing by the security layer of the Okkam.

3.2.3.2 Deletion of an Entity

One can delete an entity by specifying its Okkam ID to the AnormStore API. AnormStore API provides a single delete method which takes the Okkam ID as an argument and deletes the entity from Solr Index and Voldemort.

AnormStore API gets the SolrMaster IP address from the SolrBroker, since only the SolrBroker knows the particular SolrMaster where the entity is stored. AnormStore API will later delete the entity from that SolrMaster. SolrSlaves will update their indices by pulling the new snapshot from the SolrMaster, while Voldemort handles the deletion of all the copies of an entity.

3.2.3.3 Update of and Entity

One can update also an entity. In this case one has to specify ID of the entity to be updated.

3.2.3.4 Querying the Entities

One has to specify a query using the Lucene query language [2]. In AnormStore, once the query is specified, AnormStore API firstly contacts the SolrBroker to get the SolrSlaves so that the entire repository is covered. AnormStore API then passes the query to each of the SolrSlaves. All the results (Okkam IDs and scores) from each of the SolrSlaves are accumulated by AnormStore API. These results are ranked based on the scoring function used by Lucene [3]. Currently we are using a standard TF-IDF scoring function extended with the Okkam-specific boosting of attributes and values. The number of results to be returned can be specified in the AnormStore's configuration file. Once the Okkam IDs are obtained, AnormStore API retrieves the profiles of all entities from HBase by specifying the Okkam ID as the key.

3.2.3.5 Statistics about the Entities

Several statistics are computed over the datasets stored in the repository. Here are the main functions to query these statistics:

Function	Description
getNumberOfEntitiesWithTerm(String attribute, String term)	The number of entities, where "term" is contained in "attribute" at least once.
getNumberOfEntitiesWithTermInType(String type, String attribute, String term)	The number of entities, where "term" is contained in "attribute" at least once for a specific type.

<code>getNumberOfOccurrences(String attribute, String term)</code>	The overall number of occurrences of "term" in an "attribute". Unlike in <code>getNumberOfEntitiesWithTerm()</code> , multiple occurrences are counted multiple times.
<code>getNumberOfOccurrencesInType(String type, String attribute, String term)</code>	The overall number of occurrences of "term" in an "attribute" for a specific type . Unlike in <code>getNumberOfEntitiesWithTerm()</code> , multiple occurrences are counted multiple times.
<code>getNumberOfEntitiesWithAttribute(String attribute)</code>	Number of entities that have "attribute" filled (at least once). Multiple attributes per entity count as one.
<code>getNumberOfEntitiesWithAttributeInType(String type, String attribute)</code>	Number of entities that have "attribute" filled (at least once) for a specific type . Multiple attributes per entity count as one.
<code>getNumberOfAttributes(String attribute)</code>	Every occurrence of "attribute" in an entity counts. Together with <code>getNumberOfEntitiesWithAttribute()</code> this can be used to determine the average cardinality of an attribute.
<code>getNumberOfAttributesInType(String type, String attribute)</code>	Every occurrence of "attribute" in an entity counts for a specific type. Together with <code>getNumberOfEntitiesWithAttribute()</code> this can be used to determine the average cardinality of an attribute.
<code>getNumberOfEntitiesWithTerm(String term)</code>	Number of entities, where "term" is contained in some attribute.
<code>getNumberOfEntitiesWithTermInType(String type, String term)</code>	Number of entities, where "term" is contained in some attribute for a specific type.
<code>getNumberOfOccurrences(String term)</code>	The overall number of terms. Multiple occurrences are counted multiple times.
<code>getNumberOfOccurrencesInType(String type, String term)</code>	The overall number of terms. Multiple occurrences are counted multiple times for a specific type.

<code>getNumberOfEntities()</code>	The overall number of entities in the repository
<code>getNumberOfEntitiesInType(String type)</code>	The overall number of entities in the repository for a specific type
<code>getNumberOfDistinctTerms(String attribute)</code>	The number of distinct terms in an attribute. Together with <code>getNumberOfAttributes()</code> this can be used to estimate the overall selectivity of an attribute.
<code>getNumberOfDistinctTermsInType(String type, String attribute)</code>	The number of distinct terms in an attribute for a specific type. Together with <code>getNumberOfAttributes()</code> this can be used to estimate the overall selectivity of an attribute.
<code>getNumberOfDistinctTerms()</code>	The overall number of distinct terms
<code>getNumberOfDistinctTermsInType(String type)</code>	The overall number of distinct terms for a specific type
<code>getAttributes()</code>	List of all attributes
<code>getAttributes(String type)</code>	List of all attributes for a specific type
<code>getTypes()</code>	List of all entity types

3.2.4. Example

The following simple code example explains the typical usage of the API functions.

```

public void test(){
    try {

        StorageManager manager = StorageManager.instance();

        // Instantiate the configuration. If you wish to change the
        // configuration file then pass a different filename.
        // This is a key step. Since it setups parameters used by the
        // system.
    }
}

```

```

Configuration conf = Config.getDefault();
Iterator it = conf.getKeys();
while(it.hasNext()){
    String key = (String)it.next();
    System.out.println(key + " : " + conf.getString(key));
}

// Create a new entity writer.
IEntityWriter writer = manager.getWriter();

// Create a new entity searcher.
IEntitySearcher searcher = manager.getSearcher();

// Create a new entity reader
IEntityReader reader = manager.getReader();

// Filename that contains the entity in XML format.
String filename = "ok0204ee7e-74af-4e63-8ca8-
b4b93133f956.xml";

// Reading contents of the file which holds the entity.
String contents = "";
BufferedReader br = null;
try{
    br = new BufferedReader(new FileReader(filename));
    String str;
    while( (str = br.readLine() )!= null)
        contents += str;
    br.close();
}catch(Exception e){
    System.out.println(e.getMessage());
}

// Extract the EntityID from the filename.

// CAUTION: entityID should be the complete ID:
http://www.okkam.org/entity/ok0204ee7e-74af-4e63-8ca8-
b4b93133f956
//String entityID =
filename.substring(0,filename.lastIndexOf("."));
String entityID = "http://www.okkam.org/entity/ok0204ee7e-
74af-4e63-8ca8-b4b93133f956";

// Create new Entity by giving the entity ID and XML contents.
IStorageEntity entity = new Entity(entityID,contents);

// Write entity to the disk store. This indexes the entity
using Lucene and also stores it as a string in
// Hbase with entityID as a key.
writer.write(entity,null);

// Update an entity. Replaces an old entity with a new one.
//writer.update(entity.getID(),entity);

// Query for "california". This supports keyword queries and
Boolean queries like Lucene.
// For more information look at
http://lucene.apache.org/java/2_3_2/queryparsersyntax.html

```

```
    IStorageQuery query = new Query("california");
    Vector<IScoredEntity> entityVector =
    searcher.search(query,null);
    int size = entityVector.size();
    for(int i = 0; i < size; i++) {
        IScoredEntity se = entityVector.get(i);
        System.out.println("Entity ID : " + se.getID());
        System.out.println("Entity Score : " + se.getScore());
        System.out.println("Entity Contents : " +
        se.getXMLEntity());
    }

    System.out.println("\nGet alternative IDs");

    System.out.println(reader.getAlternativeIds("http://www.okkam.
    org/entity/ok0204ee7e-74af-4e63-8ca8-b4b93133f956",null));

    System.out.println("\nGet Okkam IDs by alternative ID");
    System.out.println(reader.getOidsByAlternativeId("08008335",
    null));

} catch(OkkamCoreException e) {
    System.out.println(e.getMessage());
}
}
```

3.3. Matching API

This section describes the objects and methods defined in the OKKAM Match component that have to be used for employing the OKKAM Match functionality. Section 3.2.1 explains how to request for entities, whereas Section 3.2.2 describes the format of the results of such an entity request.

3.3.1. Requesting Entities

The core functionality provided by the Matching API is answering entity requests. This means it accepts a description of an entity, checks if the described entity is available in the OKKAM entity repository and, if this is the case, returns one or more matching candidates. The entity description is given in the form of an entity request following the rules of the OKKAM Request language. The matching candidates are those entities in the repository that according to the employed matching algorithm are possible matches for the entity described by the entity request.

3.3.1.1 Methods for Requests

The OKKAM Match component offers two main methods for requesting entities (Table 1 provides the signatures of these methods):

- **Single Resolution Request.** This method is used for retrieving the entities (matching candidates) that match the entity request. The entity request is given as a string. The result of a single resolution request is a matching candidates bundle; a ranked list with entities that have been matched to the entity request.
- **Bulk Resolution Request.** This is a method for requesting the resolution of a collection of (interlinked) entities. Similar to the single resolution request, each entity is specified by a request, and the result is a collection of matching candidate bundles.

<pre>IMatchingCandidateBundle getEntity (String stringQuery, ISessionMetadata sessionMetadata);</pre>
<pre>Collection<IMatchingCandidateBundle> getEntities(List<String> stringQueryCollection, ISessionMetadata sessionMetadata);</pre>

Table 1. The main methods available in OKKAM Match for making entity requests.

In the next paragraphs we describe the format of the requests itself (Section 3.2.2.2) and how to select a matching module for a request (Section 3.2.2.3).

3.3.1.2 Formulating Requests

Formulating requests is done using the OKKAM Request language which is described by the grammar in Table 2. A full request consists either of a query, or a query together with metadata and context. Metadata is used for selecting the matching module (see Section 3.2.1.3).

A well-formed query consists of (eventually nested) conjunctions and disjunctions of words or URIs (interpreted as values) to which following information can be added:

- relevance of the value within the whole query, and
- an attribute for this value.

<pre>/** Atomic tokens */ OR ::= "OR" AND ::= "AND" AND ::= "NOT" OPERATOR ::= "=" URI_LEFT_DELIMITER ::= "<" URI_RIGHT_DELIMITER ::= ">" RELEVANCE_OPERATOR ::= "*" LEFT_PARENTHESIS ::= "(" RIGHT_PARENTHESIS ::= ")" LEFT_CURLY ::= "{" RIGHT_CURLY ::= "}"</pre>
--

```

/** Structured tokens */
ATTRIBUTE_NAME_OR_VALUE ::= Nmtoken | RELEVANCE | QuotedString
DELIMITED_URI ::= URI_LEFT_DELIMITER URI URI_RIGHT_DELIMITER
RELEVANCE ::= float

/** Query */
QUERY_SECTION ::= "QUERY" LEFT_CURLY QUERY RIGHT_CURLY
QUERY ::= TERM (OR QUERY)*
TERM ::= NOT? FACTOR (AND? TERM)*
FACTOR ::=
    (ATTRIBUTE_NAME_OR_VALUE OPERATOR)? (
        ATTRIBUTE_NAME_OR_VALUE |
        DELIMITED_URI
    ) (RELEVANCE_OPERATOR RELEVANCE)? |
    LEFT_PARENTHESIS QUERY RIGHT_PARENTHESIS

```

Table 2. Request language grammar.

The following are three examples of requests which describe Einstein, the famous Nobel prize physicist:

- **R.1:** name = “Albert Einstein”
- **R.2:** Albert Einstein
- **R.3:** name = “Albert Einstein” Nobel Prize

3.3.1.3 Selecting Matching Module

A request can be also accompanied with metadata and context information. Metadata are used for selecting the matching module to be used during the processing of the request. The underlying matching framework provides the option to select matching modules at processing time. Context information is currently not used but provided for future extension. Table 3 provides the respective part of the request language grammar.

```

/** Top Level */
FULL_QUERY ::=
    QUERY_SECTION (METADATA_SECTION)? (CONTEXT_SECTION)? |
    QUERY

/** Metadata */
METADATA_SECTION ::= "METADATA" LEFT_CURLY PAIRS? RIGHT_CURLY

/** Context */
CONTEXT_SECTION ::= "CONTEXT" LEFT_CURLY PAIRS? RIGHT_CURLY

PAIRS ::= PAIR (AND? PAIR)*
PAIR ::=
    ATTRIBUTE_NAME_OR_VALUE OPERATOR (ATTRIBUTE_NAME_OR_VALUE |
    DELIMITED_URI )

```

Table 3. Request language grammar (matching module selection).

The following are two examples of requests which define the matching module that should be used for processing:

- **R.1:** QUERY { name = “Albert Einstein” } METADATA { matchingModule=glinkage }

- **R.2:** QUERY { Albert Einstein } METADATA { matchingModule=glinkage }

3.3.2. Result format

The result of each request is a list of candidate entities found currently in OKKAM, which were matched with the given request, i.e., which satisfy the conditions described in the request. The number of entities in this return list is ideally only one, since each request is made for retrieving a specific entity. As such, OKKAM Match returns an `IMatchingCandidateBundle` object that contains the matching entity candidates. The following table shows the methods available under this object.

<code>IMatchingCandidateBundle</code>	Comments
<code>Iterator<IMatchingCandidate> iterator();</code>	An iterator for navigating over the results of the specific request.
<code>int size();</code>	Gets the size of the bundle.

3.3.2.1 Matching Candidates

A Match candidate encodes an entity matched with a specific request. It is given a `IMatchingCandidate` object and it allows to access three related information, the OKKAM identifier of the specific entity, the entity profile, and matching probability computed during matching. The following table shows the methods available under this object.

<code>IMatchingCandidate</code>	Comments
<code>String getOkkamId();</code>	Gets the matching candidate profile.
<code>Entity getEntityProfile();</code>	Gets the OKKAM identifier of this matching candidate.
<code>double getProbability();</code>	Gets the matching probability.

3.3.2.2 Match probability

Matching probability depends on two main factors: similarity and distinctiveness. If the values of a request attribute and an entity attribute are similar (w.r.t. an underlying error-model), the match probability increases. But in addition, a match on an attribute with fairly unique values, such as name, contributes much more evidence for a match of the entity than a match on an attribute with very few values, such as gender = male or female.

More formally, this is captured by the Fellegi-Sunter criterion for duplicate detection³. Adapted to the OKKAM scenario the criterion can be stated as follows:

³ Fellegi and Sunter formalized this criterion, aka "Bayes Decision Rule for Minimum Error"

$$(r, e) \in \begin{cases} M, & \text{if } P(M|r, e) > P(U|r, e) \\ U, & \text{otherwise} \end{cases}$$

This criterion says that entity e should be accepted as a match for request r , if the probability for a match M given $(r; e)$ is larger than the probability for no match U given $(r; e)$.

The method `getProbability` under each Matching Candidate returns the matching probability for this entity given the respective request.

3.4. Entity Lifecycle API

This section aims at providing an overview of the objects and the methods defined in the Lifecycle Manager Component in order to manage entity creation and update tasks, entity description default attributes and other auxiliary functions. In Section 3.4.1 we give an overview of entity creation methods. Section 3.4.2 gives a brief description of the entity editing operations. In particular, section 3.4.2.2 will describe the entity update APIs, and sections 3.4.2.3 present overviews over the entity split and merging operation. Section 3.4.3 will provide a description of the delete procedures, Section 3.4.4 described the APIs used to get entities and finally Section 3.4.5 provides a description of other auxiliary functions currently implemented within lifecycle manager component.

3.4.1. Creating a New Entity

One of the core functionalities of the currently implemented lifecycle manager component is the creation of new entity in the ENS. A proper management of this operation is essential to guarantee minimum number of entity duplicates in the ENS, an adequate quality of entity description. With this purpose, in the context of entity creation, a check for duplicates is performed every time before actually writing a new entity in the ENS, and entity description default attributes are made available for the external application.

3.4.1.1 Methods for Default Attributes management

The ENS currently supports a set of Entity Semantic Types aiming at defining a high level categorization of the entities within the ENS. One of the goals of this categorization is to provide for each entity type a set of default attributes which can be used to create a core template for the entity description. The aim of the default attributes associated with each entity type is to support the human user in defining unambiguous descriptions of the entity at creation time.

The method supporting the retrieval of the supported entity types is:

```
List<String> getEntityTypes()
```

The purpose of this method is to return a list of all the entity types currently supported by the ENS.

The method supporting the retrieval of the default attributes associated with each supported entity type is:

```
List<String> getDefaultAttributes(String entityType)
```

The purpose of this method is to support external applications in dealing with the default attributes, by returning a list of the default attributes of the specified entity type.

3.4.1.2 Methods for Entity Creation

The lifecycle manager component supports 2 ways to create new entity in the ENS: single entity creation and batch entity creation.

Single Entity Creation

Using single entity creator web application, a user can create a (single) new entity using the relying on two methods. The first method performs a formal validation of the entity to be created, and is:

```
EntityValidationReport validateEntity(Entity entity, boolean ignoreDuplicates)
```

The purpose of this method is to performs a general validation over the 'entity'. The parameters are the 'entity' object to be validated and a boolean parameter 'ignoreDuplicates' indicating whether entity validation should be perform a check against duplicate detection. This method returns a EntityValidationReport object containing

- a WriteEntityPermission value (e.g. Yes, No, Maybe)
- a certificate of validity of the submitted entity
- a list of candidate duplicates
- a EntityQualityReport object

If the entity validation is completed successfully, the WriteEntityPermission value is set to YES, the list of candidate duplicates is empty, and the EntityQualityReport presents at worst a list of “warning” respect to not mandatory broken quality rules, and a quality certificate is returned. If the duplication check detects candidate duplicate entities, the WriteEntityPermission is set either to MAYBE or NO and the list of candidate duplicates returned is filled. In this case, no validation certificate is returned. The check for duplication is performed by inquiring the ENS with the description of entity parameter 'entity'. The list of entities matching the entity description is the filtered according to some parameter determining the relevance of the entities retrieved. If the 'ignoreDuplicates' parameter is set to true, the entity validation process skips the check for duplicates and simply performs a syntactic and qualitative analysis of the entity submitted. If the entity validation was positive, and the WriteEntityPermission returned is set to YES, then the validated entity object is stored locally, tied with the validation certificate returned. If the user wants to confirm the creation of the entity with validated description, he can do it and by calling the method createNewEntity(), described here:

```
String createNewEntity(String certificate)
```

The purpose of this method is to confirm the creation of a previously validated entity. The invocation of this method writes in the storage the entity associated with the submitted certificate and previously stored locally. The parameter is a string certificate which identifies an entity object previously validated and temporary stored. Returns the Okkam Id of the freshly created entity if everything worked correctly, returns an exception otherwise.

Batch Entity Creation

Batch entity creation does not perform any duplication check because it is basically used only in the context of initial entity import, thus the no duplicates' existence is assumed.

```
List<NewEntityResult> createEntities(List<Entity> entities, boolean force)
```

The purpose of this method is to support the creation of a group of entities in sequence.

Entity batch import passes through three basic steps: 1) state the beginning of the import operations invoking the method *batchImportBegin()*, 2) iterated over the list of entities to create invoking for each entity the method *batchImport()*, 3) state that the import process is complete invoking the method *batchImportCommit()*.

In case the imported entities is carrying a valid entity identifier, it is possible to execute a batch import step without overwriting the entity identifier by invoking the *batchImportKeepOkkamIds()* method on place of the *batchImport()* method. This method basically executes the batch import of the entity preserving the entity identifier passed with the 'entity' parameter.

3.4.2. Entity Evolution

The main goal of the lifecycle manager is to support in a consistent way the evolution of an entity. With this aim, lifecycle manager presents methods and procedures supporting the update of an entity description, the merge and split of existing entities. All these operations must be performed in a context where only one process/user can have exclusive editing rights over an entity description. In order to make this possible, the lifecycle manager implemented a sort of locking system that, used consistently, would avoid concurrent entity editing.

3.4.2.1 Entity Locking System

The entity locking system aims the supports the lock, unlock and lock verification operations. When a user, or a process, needs to lock an entity to perform some editing operation, it must invoke the following method:

```
String lockEntity(String oid)
```

The purpose of this method is perform and attempt of locking an entity giving "editing" rights only to one application/process/user per time. The parameter is the entity identifier of the entity that should be locked. Returns a locking ticket if the entity wasn't previously locked, throws an

exception otherwise. If the lock is positive, a semaphore object is stored with a certain timestamp and a locking ticket. The locking ticket is returning to the locking process, guaranteeing that only one process per time can perform any operation over the semaphore object (e.g. unlock the entity). Every editing operation involving the locked entity id, needs also a valid locking ticket. If the editing operation does not use a valid locking ticket, the operation is aborted and an exception is returned.

Indeed, in order to unlock an entity, a user/process must invoke the following method:

```
void unlockEntity(String oid, String ticket);
```

The purpose of this method is to perform an attempt of unlock a previously locked entity. The parameters are the oid of the locked entity, and the locking ticket guaranteeing the the exclusive editing right over the locked entity. If the ticket is not valid, or the lock was expired, an exception is thrown.

```
List<String> lockEntities(List<String> oids)
```

The purpose of this method is to perform an attempt of locking a set of entities in a batch. The parameter is a list of entities' identifier. Returns a list of locking tickets if the attempt of locking all the entities was successful, throws an exception otherwise.

3.4.2.2 Entity Update

Another important functionality implemented in the lifecycle manager component is the update of an entity description. The current business logic behind the entity update method is an extended version of the one defined for entity creation. Namely, in order to perform an entity update, the user must lock the entity, and get a valid entity lock ticket. After editing operation are performed, the entity description must be validated following the same procedure as the for entity creation. If the entity was positively validated, the the user/process can complete the process actually writing the update in the repository by invoking the method:

```
void updateEntity(String ticket, String certificate)
```

The purpose of this method is to trigger the writing of the update of a previously validated and locked entity. The parameters are the locking ticket guaranteeing exclusive editing right over the updated entity, and the validation certificate that is tied with the valid updated description to be written in the repository. If the locking ticket is not valid or the validation certificate is expired, then an exception is thrown.

3.4.2.3 Entity Merge and Entity Split

An important novelty introduced by this new version of the lifecycle manager is the possibility of merging and splitting entities description. The merging of entities is necessary to reduce the number of duplicates in the repository, indeed it is plausible that in time, several entities might be created with different sets of information but referring to the same real world entity. In this case it is possible to merge the entities description in a new single entity, and keep the old entity identifiers as equivalent identifiers of the newly created entity. For the opposite reason, in time it might be necessary to split an entity description in two different entities. As for entity update, also entity merge and entity split need an exclusive editing right over the entity. For this reason, entity merge and split require a valid locking ticket to be successfully executed. Entity merge can be formed invoking the following method:

```
NewEntityResult mergeEntities(List<String> oids, Entity mergedEntity,  
List<String> tickets)
```

The purpose of this method is to perform the merging of a set of entities previously locked. The method works verifying the locking rights of the merged entities, writes a new entity which description is the result of the merging, and delete the merged entities. The oid of the original entities is set as equivalent oid of the merged entity. The entities presenting references to the deleted entities are updated with the oid of the newly created entity. The parameters are the list of oids of the merged entities, the entity object containing the merged entity description, and the list of locking ticket guaranteeing exclusive editing rights over merged entities. It returns a *NewEntityResult* object which presents the oid of the merged entity and a list of potential duplicate candidates.

Entity split can be performed executing the following method:

```
List<NewEntityResult> splitEntity(String oid, List<Entity> splitEntities, String  
ticket)
```

The purpose of this method is to perform the splitting of a previously locked entity in a set of entities containing a sub part of the description of the original entity. The method verifies the locking rights over the split entity, writes the new entities, and deletes the original one. The oid of the original entity is set as equivalent oid of the newly created entities. The parameter are the oid of the split entity, the list of entities' object containing the subpart of the description of the split entity, and the locking ticket. Returns a list of objects containing the oid of the new entity result of the split, and list of candidate duplicates potentially already present in the repository.

3.4.3. Entity Delete

Another important novelty introduced by the new version of lifecycle manager, is the possibility of removing an entity from the repository. This method is meant exclusively for administration use, and will not be exposed publicly. This method is meant to support administrators in removing malformed entities introduced in the repository for example during unfortunate testing tasks. As for any editing operation, also entity deletes requires the entity to be locked. In order to delete an entity it is necessary to invoke the following method:

```
void deleteEntity(String oid, String ticket)
```

The purpose of this method is to remove from the repository the entity previously locked. The parameters are the oid of the entity that must be removed from the repository and the locking ticket that guarantees the editing rights to the process calling this method.

3.4.4. Entity Get

Lifecycle manager is responsible for offering APIs to get entities stored in the repository. The methods available are:

```
Entity getEntity(String oid)
```

The scope of this method is to retrieve an entity object given its entity identifier. The parameters are the entity identifier of the target entity, and the session metadata. Returns the entity identified by the 'oid' parameter. If error or exception are caught, the method returns 'null'.

```
List<Entity> getEntities(List<String> oids)
```

The purpose of this method is to retrieve a list of entity object, given the list of their identifiers. Basically this method is used to perform the fetching of a group of entities. The parameters are a list of entity identifier. Returns a list of entities identified by the list of entity identifier parameters 'oids'. Basically it iterates on the list of oids invoking the method *getEntity(oids[i])*.

```
List<String> getAlternativeIds(String oid)
```

The purpose of this method is to retrieve the list of alternative identifier associated with an entity. The parameters are the identifier of the entity target 'oid'. Returns the alternative ids of the entity identified by 'oid' parameter.

```
List<String> getOidsByAlternativeId(String aid)
```

The purpose of this method is to retrieve all the entities that present a specific alternative identifier. The parameters are the alternative identifier. Returns the list of entities' identifier presenting the parameter 'aid' as alternative identifier.

3.4.5. Other methods

```
void logEntitySelection(String oid, String query)
```

The scope of this method is to add an entry in the log about the selection of the entity among a set of others returned to the user with respect to a query. The parameters is the oid of the entity selected, and the query that caused its retrieval.

3.5. Secure Access to Protected ENS APIs

This section discusses how access to protected ENS APIs has been designed, and how to use OKKAM security proxy in your Java applications to enable remote WS APIs access. It first presents the motivation of adopting the proxy component followed by how the security proxy facilitates secure and trusted communications with ENS APIs. The section then presents security specific APIs of ENS. We note that the information of this section is subject to change depending on actual ENS state of development. Most up-to-date information can be found on the ENS community Web portal⁴.

The OKKAM consortium has decided to provide ENS functionalities as both REST-style web services interfaces, and as SOAP and WSDL based web services. Public non-protected services of ENS are exposed as both REST-style and WSDL-based web services interfaces. All services that have been defined as protected (controlled access) are accessible **only** by standard Web Services technologies, i.e. by using SOAP protocol and WSDL.

Security of OKKAM considers exclusively access to ENS services provided as standard Web Services technologies. Access to ENS services provided as REST web services is beyond the security scope of OKKAM. The REST services are provided to facilitate ENS usage in mashup applications. As said above, all REST services are public non-protected services that have equivalent service functionality provided via standard Web Services interfaces.

3.5.1. Security proxy

Access to ENS APIs requires Web services technologies. There are well-defined Web services security standards that facilitate secure and trusted interactions between entities. Some of adopted Web services standards are WS-Security, WS-Trust, WS-SecurityPolicy, WS-SecureConversation etc. To facilitate secure and trusted interactions with ENS, it has been designed and developed a security proxy component on both client-side and service-side to provide necessary authentication

⁴ <http://community.okkam.org/index.php/Documentation/APIs/>

and access control mechanisms. The main goal behind the proxy design is to hide complexity and facilitate management of security technologies to OKKAM users, application developers, as well as, to ENS core developers. It is designed to make secure communications as transparent as possible to OKKAM-enabled applications.

Another main achievement of the proxy is the encapsulation of authentication and access control mechanism for accessing ENS APIs. The main goal here remains the same – achieving as much as possible transparent authentication and access control process to OKKAM-enabled applications.

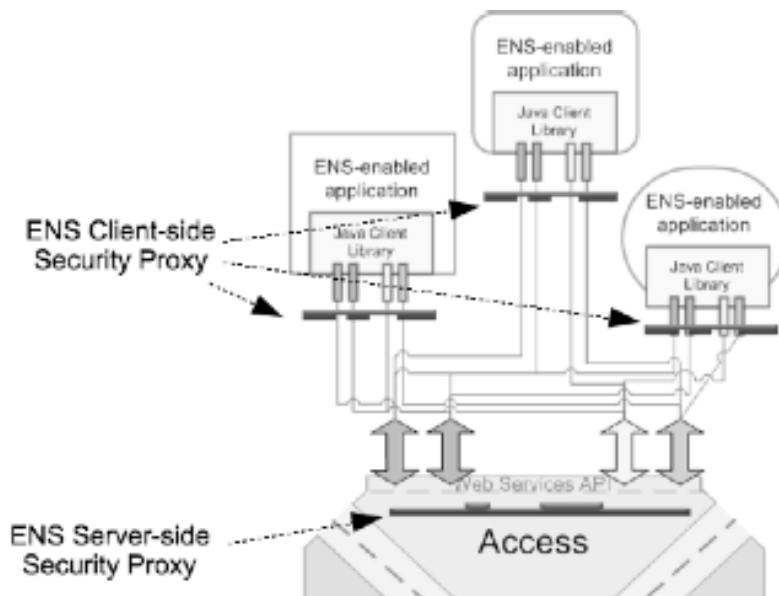


Figure 2 Client-Server Model of Security Proxy

Figure 2 shows the security proxy approach deployed in the ENS. All messages to and from the ENS, which are part of security communications, are communicated via the proxy components.

To achieve transparent and intuitive interactions with ENS it has been defined that the proxy component implements all remote ENS APIs as they were locally deployed. Thus the proxy component becomes as local replica of ENS APIs with respect to user-side applications.

Technically, the proxy approach provides confidential channel between the user-side proxy and the server-side proxy components. The confidential channel is established with a mutual certificate security mechanism that uses both, client and server-side public-key certificates.

On a user side, a confidential channel is established if a user is a registered OKKAM user, i.e. a legitimate OKKAM user. We note that the user-side proxy component works always on behalf of a registered entity, i.e. the proxy authorizes to ENS as being the user itself.

Once mutual authentication and a confidential channel are established, there is a mandatory access control process that controls if a registered user is allowed to access requested ENS APIs. The server-side proxy and a client-side proxy automatically negotiate if more access rights (certified

attributes) are necessary for a client to get access to the requested API. We refer to OKKAM deliverable D2.1⁵ for details on the proxy architecture and designed access control process.

3.5.2. Accessing protected ENS APIs

The ENS provides all functionalities via Web Services APIs divided in two sets: **Public WS APIs**⁶ and **Protected WS API**⁷. The public APIs are available for public access by standard web services means. The set of public APIs contains those APIs available for open and uncontrolled access (e.g., all query-related APIs fall in this category).

All ENS APIs are made publicly accessible but those of them that require protected access have no functional “body” but instead always return an authorization exception with a message “Unauthorized public access”. The reason to leave all ENS APIs public accessible is that any developer should use the public APIs to know what functionalities ENS provides and what interface the developer has to conform to in order to use core ENS services and what input message structure an API accepts.

The protected APIs require secure and confidential communications with appropriate access rights for their execution. An important security aspect here is that all public APIs are also provided as protected APIs (with no controlled access but accessible over a secure communication channel). In that way, a user can enforce privacy and confidentiality on all interactions with the ENS APIs including the non-protected APIs. For example, when a user wants to avoid any intermediary third parties reading what he queries ENS for (or what he interacts with ENS for).

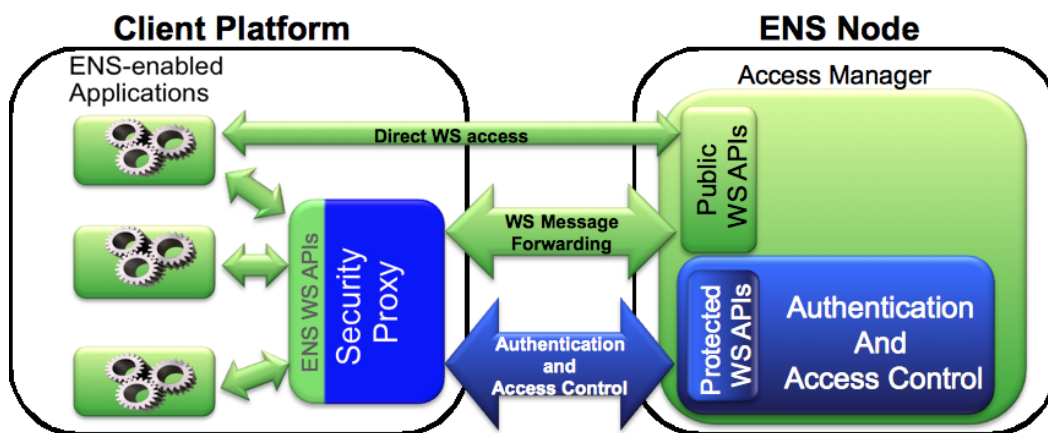


Figure 3 API-level Proxy-based Communications

All interactions with protected ENS APIs are to be performed via the security proxy. Figure 3 shows the API-level message flow of proxy-based communications. Applications can establish direct communications or indirect via proxy communications with the public APIs. The proxy component replicates **all** ENS APIs, both public and protected ones, as locally accessible to

⁵ <http://www.okkam.org/deliverables/OKKAM-D2.1.pdf>

⁶ <http://api.okkam.org/okkam-core/WebServices?wsdl>

⁷ <http://api.okkam.org/okkam-core/WebServicesSecured?wsdl>

applications in order to make access to those API as much transparent as possible by application developers.

The goal is to provide developers with uniform access management allowing all APIs to be accessed via the proxy component without keeping in mind which of those require protected access and which do not. The proxy component is aware of which APIs are protected and for those of no required protection the proxy just dispatches the respective requests/responses to the public ENS APIs as shown in the figure.

If the proxy is configured to enforce secure communications for all interactions to the ENS, then it will dispatch all APIs invocations to the protected APIs. In that case, even if a non-protected API is locally invoked, the proxy will dispatch the request to the protected version of this API at the ENS side.

The proxy component has two main usage modes: as a local host service and as a library. The local host service implements the same WS APIs (interfaces only) but accessible on local host connection⁸. This is an important mode addressing the needs of so-called "thin" applications, such as plug-ins for third party applications, or applications that cannot use the proxy as a java library, such as non-java based applications. In such case, an application invokes the WS APIs on localhost for accessing the remote WS APIs. The library usage, in contrast, provides the respective ENS APIs as java APIs signatures for accessing the remote WS APIs, as described later.

We list below the set of public WS APIs provided by ENS. We omit the input and output messages of APIs for the sake of simplicity of presentation. We mark in bold those WS APIs that require protected WS APIs access.

Public WS APIs
<code>getEntity(...)</code> throws <code>OkkamCoreException</code>
<code>findEntity(...)</code> throws <code>OkkamCoreException</code>
<code>getAlternativeIds(...)</code> throws <code>OkkamCoreException</code>
<code>getOidsByAlternativeId(...)</code> throws <code>OkkamCoreException</code>
<code>getEntities(...)</code> throws <code>OkkamCoreException</code>
<code>getTypeTemplate(...)</code> throws <code>OkkamCoreException</code>
<code>logSelectedEntity(...)</code> throws <code>RemoteException</code>
<code>lockEntity(...)</code> throws <code>OkkamAuthorizationException</code> , <code>OkkamCoreException</code>
<code>lockEntities(...)</code> throws <code>OkkamAuthorizationException</code> , <code>OkkamCoreException</code>
<code>unlockEntity(...)</code> throws <code>OkkamAuthorizationException</code> , <code>OkkamCoreException</code>
<code>validateEntity(...)</code> throws <code>OkkamAuthorizationException</code> , <code>OkkamCoreException</code>
<code>createNewEntity(...)</code> throws <code>OkkamAuthorizationException</code> , <code>OkkamCoreException</code>
<code>deleteEntity(...)</code> throws <code>OkkamAuthorizationException</code> , <code>OkkamCoreException</code>
<code>mergeEntities(...)</code> throws <code>OkkamAuthorizationException</code> , <code>OkkamCoreException</code>
<code>splitEntity(...)</code> throws <code>OkkamAuthorizationException</code> , <code>OkkamCoreException</code>
<code>updateEntity(...)</code> throws <code>OkkamAuthorizationException</code> , <code>OkkamCoreException</code>

⁸ The proxy component is not necessarily to be run on a localhost but could also be run on a separate machine with intranet access. However, in this case one needs to additionally secure access between the application and the proxy.

Below we list the set of WS APIs available as protected APIs. We note that some of the protected APIs are not in the public space of ENS APIs due to their specific to OKKAM needs, for example, those APIs dedicated to be used by the ENS web toolkit and administration console front-end applications on behalf of users.

In the set of protected APIs there are also all APIs from the public space, which do not require protected access. The decision of doing so is motivated by the need of privacy control of users when interact with the ENS. In that way, by configuring the security proxy, end users can achieve complete security and privacy of their interactions with the ENS where the proxy will tunnel all calls to the respective APIs in the set of protected services (named below as non-protected APIs for secure access).

Protected Access WS APIs	Access Control Requirements
lockEntity(...) throws OkkamAuthorizationException, OkkamCoreException	Administrator or trusted entity creator privileged access.
lockEntities(...) throws OkkamAuthorizationException, OkkamCoreException	Administrator or trusted entity creator privileged access.
unlockEntity(...) throws OkkamAuthorizationException, OkkamCoreException	Administrator or trusted entity creator privileged access.
validateEntity(...) throws OkkamAuthorizationException, OkkamCoreException	OKKAM user or administrator or trusted entity creator privileged access.
createNewEntity(String certificate) throws OkkamAuthorizationException, OkkamCoreException	OKKAM user or administrator privileged access.
deleteEntity(String oid, String ticket) throws OkkamAuthorizationException, OkkamCoreException	Administrator privileged access.
mergeEntities(String[] oids, String mergedEntity, String[] tickets) throws OkkamAuthorizationException, OkkamCoreException	Administrator privileged access.
splitEntity(String oid, String[] splitEntities, String ticket) throws OkkamAuthorizationException, OkkamCoreException	Administrator privileged access.
updateEntity(String ticket, String certificate) throws OkkamAuthorizationException, OkkamCoreException	Administrator privileged access.
OnBehalf APIs	
createNewEntityOnBehalf (String certificate, byte[] x509UserCert) throws OkkamAuthorizationException, OkkamCoreException	Administrator or trusted entity creator privileged access.
updateEntityOnBehalf (String ticket, String certificate, byte[] x509UserCert) throws OkkamAuthorizationException, OkkamCoreException	Administrator or trusted entity creator privileged access.
deleteEntityOnBehalf (String oid, String ticket, byte[] x509UserCert) throws OkkamAuthorizationException, OkkamCoreException	Administrator privileged access.
mergeEntitiesOnBehalf (String[] oids, String mergedEntity, String[] tickets, byte[] x509UserCert) throws OkkamAuthorizationException, OkkamCoreException	Administrator privileged access.

<code>splitEntityOnBehalf</code> (String oid, String[] splitEntities, String ticket, byte[] x509UserCert) throws OkkamAuthorizationException, OkkamCoreException	Administrator privileged access.
Non-protected APIs for secure access	
<code>getEntity(...)</code> throws OkkamCoreException	Anonymous or OKKAM user
<code>findEntity(...)</code> throws OkkamCoreException	Anonymous or OKKAM user
<code>getAlternativeIds(...)</code> throws OkkamCoreException	Anonymous or OKKAM user
<code>getOidsByAlternativeId(...)</code> throws OkkamCoreException	Anonymous or OKKAM user
<code>getEntities(...)</code> throws OkkamCoreException	Anonymous or OKKAM user
<code>getTypeTemplate(...)</code> throws OkkamCoreException	Anonymous or OKKAM user
<code>logSelectedEntity(...)</code> throws RemoteException	Anonymous or OKKAM user

Important to note is the difference of *on behalf* functions from legal point of view. Those functions logs internally the authors of the respective act of execution of those functions the end-user specified in the input parameter, and *not* the entity authorized to access those functions. For example, upon execution of `createNewEntityOnBehalf(...)` by the ENS web toolkit, the security module will log the following information: “An entity creator ID was authorized to execute `createNewEntityOnBehalf` function where the author of the entity creation act is the end user ID”. Here the IDs are the identity of ENS web toolkit and end-user as described in the entity’s X.509 public-key certificate.

Another important issue is that all on behalf functions accept as input the same input parameters as their corresponding functions but require one more argument – the public-key certificate of the end-user wishing to execute those functions.

At this point, the security module does not consider any delegation aspects that the end-user does aim at accessing those functions via a front-end. The motivation is that the only entities currently allowed to access those APIs are trusted to ENS (namely, administrators or trusted entity creators).

The `createNewEntityOnBehalf(...)` and `updateEntityOnBehalf(...)` are provided to serve the purposes of the ENS web toolkit (entity creator), while delete, merge and split on behalf APIs are designed to serve only the purposes of the administration console front-end. In this way, we achieve separation of responsibilities of both front-ends.

3.5.3. Security-specific APIs

In addition to the ENS APIs of functional aspects, there are APIs dedicated to security needs only. Those APIs are dedicated to be used by the OKKAM security proxy, but not exclusively. Below we list those APIs.

Security-specific APIs	Functional Description/Access Control (AC) Requirements
Public APIs:	
<code>public byte[] getServerCertificate()</code>	Returns the public-key certificate of ENS (replica) server.
<code>public boolean isCertificateRevoked(BigInteger serialNumber)</code> throws OkkamCoreException	Returns true if the certificate serial number exists in the underlying certificate revocation

	<p>lists. Returns false otherwise. This method is provided for most accuracy of revocation status.</p>
<p>Protected Access APIs:</p>	
<pre>public byte[] registerUser(String firstName, String lastName, String username, String password, String email, String organization, String country, boolean storeUserPrivateKey) throws RemoteException, OkkamAuthorizationException</pre>	<p>Registers user into the ENS persistent storage. Returns a keystore file, in PKCS12 format, encrypted with the password of registration.</p> <p>Access to this API is given to any application implementing WS-Security with anonymous user-side certificate. For example, user registration via the security proxy GUI, where the security proxy generates anonymous public-key certificate for accessing the API over a security channel (formed by following mutual certificate security mechanism using a trusted server certificate and the anonymous client certificate).</p> <p>AC: Anonymous access</p>
<pre>public void removeRegisteredUser(String email) throws RemoteException, OkkamAuthorizationException</pre>	<p>Privileged removal of user data from the ENS. Before removing user profile, this function first <i>revokes</i> all user's associated certificates and then removes the user profile from the persistent storage.</p> <p>AC: Administrator privileged access.</p>
<pre>public byte[] certifyUserAttribute(byte[] x509UserCert, AttributeRoleName attribute, int validityInDays) throws RemoteException, OkkamAuthorizationException</pre>	<p>Generates an X.509 attribute (<i>attribute</i>) certificate bound to the user-distinguished name of the public-key certificate of the first argument (<i>x509UserCert</i>) for the validity of specified days. Validity of an attribute certificate can be maximum the validity of the specified user's public-key certificate.</p> <p>AC: Administrator privileged access.</p>
<pre>public CredentialResponse getAuthorizationWS(String requestedService, String requestedCredential, List<Credential> presentedCredentials, List<String> deniedCredentials)</pre>	<p>A policy decision point (PDP) interface of ENS, which is for internal to security proxy usage only. Implements an interactive access control model for negotiation of necessary credentials between a client and a server. The method is limited to use by OKKAM registered users only.</p> <p>AC: OKKAM user privileged access.</p>
<pre>public byte[] getUserRegistrationData(String email,String newPassword) throws RemoteException, OkkamAuthorizationException</pre>	<p>Retrieves user registration data from the ENS using user's email. The outcome is a keystore file in PKCS12 format encrypted with the specified password (<i>newPassword</i>).</p> <p>AC: Administrator privileged access.</p>

<pre>public long countRegisteredUsers() throws OkkamAuthorizationException</pre>	<p>Returns the number of currently registered users. To be used for statistics only. AC: Administrator privileged access.</p>
<pre>public boolean revokeUserOwnCertificate(byte[] certificate) throws OkkamAuthorizationException, OkkamCoreException</pre>	<p>Revokes user's own certificate. To be used by OKKAM users only. Input is the user's own certificate to be revoked. This function accepts an identity or attribute certificate as an input argument.</p> <p>If public-key certificate to be revoked, the function extracts the user's public-key certificate from Metro-level authentication mechanism and compares if the input certificate matches the certificate the user has authenticated to ENS. If match succeeds the user's certificate is revoked otherwise returns false.</p> <p>Note that when a user revokes his public-key certificate, this function has a <i>side</i> effect of first revoking all user's assigned attribute certificates and then revokes the public-key certificate of the user. Furthermore, when the user's public-key certificate is revoked, the function <i>removes</i> user's registration data from the system.</p> <p>If a user revokes an attribute certificate, the function checks if the user of the authentication process does have the corresponding attribute certificate and revokes it then, otherwise returns false. AC: OKKAM user privileged access.</p>
<pre>public boolean revokeUserCertificate(byte[] certificate) throws OkkamAuthorizationException, OkkamCoreException</pre>	<p>Privileged revocation of user's public-key or attribute certificates. This API is provided only to administrators of ENS. ENS administrators are given the right to revoke user's certificates. When a public-key certificate is being revoked, the function first revokes all user's attribute certificates, then revokes the user's public-key certificate, and then removes user's profile from the ENS. If a attribute certificate is being revoked, the function revokes only that certificates.</p> <p>Returns true upon successful revocation process, false otherwise.</p> <p>This API is to be used <i>only</i> for revocation of end-users certificates. AC: Administrator privileged access.</p>

<pre>public boolean revokeAuthorityCertificate(byte[] certificate) throws OkkamAuthorizationException, OkkamCoreException</pre>	<p>Privileged revocation of OKKAM certificate authorities' public-key certificates. This API is to be also used for revocation of public-key and attribute certificates of Administration Console and ENS Web toolkit front-ends. Access to this API is restricted to ENS Managers only.</p> <p>AC: ENS Manager privileged access.</p>
<pre>public boolean isCertificateRevoked(BigInteger serialNumber) throws OkkamCoreException</pre>	<p>Certificate revocation status function. It checks if a given certificate's serial number has been revoked. Returns true if the serial number matches a revoked certificate, false otherwise.</p> <p>This API duplicates the functionality of a same API in the public service space of ENS. This function is to be used when confidentiality and data integrity are required for certificate status checking. For example, if a client agent wants to verify what current status the "OKKAM CA Class 2" public-key certificate has, it can do so by using this function ensuring confidentiality and integrity of response.</p> <p>AC: OKKAM user privileged access.</p>

Since ENS security is entirely driven by digital certificates, an important point of defining ENS APIs is the message encoding of digital certificates when exchanged between users and ENS. All input and output messages of ENS APIs that denote digital certificates are defined as binary array of bytes. There are two certificate's message formats supported as input of ENS APIs:

- Base64⁹ encoded DER¹⁰ certificate data, enclosed between "-----BEGIN CERTIFICATE-----" and "-----END CERTIFICATE-----"
- Binary DER format of certificate data.

All APIs output messages of certificates are in binary DER encoded certificate data (the second bullet above), such as the case of `certifyUserAttribute(...)` API. We note that the encoding settings above refer to both public-key and attribute certificates. ENS supports fully X.509 v3 standard of identity and attribute certificates.

3.5.4. Certificate revocation

There are two certificate revocation lists maintained by OKKAM. Below we list the two URLs of those:

- <http://api.okkam.org/okkam-core/crl/okkamall.crl>

⁹ <http://en.wikipedia.org/wiki/Base64>

¹⁰ http://en.wikipedia.org/wiki/Distinguished_Encoding_Rules

– <http://api.okkam.org/okkam-core/crl/okkamauthorities.crl>

The first CRL regards revocation certificates of all OKKAM entities – both end users and OKKAM authorities, while the second CRL regards revocation certificates of OKKAM authorities only. The `okkamall.crl` is to be used by entities requiring validation of end-users such as the ENS servers, the administration console and the ENS web toolkit, while the `okkamauthorities.crl` is to be used primarily by end-users when interacting with OKKAM front-ends or with the ENS.

The `okkamauthorities.crl` is a subset of `okkamall.crl`. Furthermore, `okkamauthorities.crl` is expected to have drastically fewer revocation certificate entries with regards to the `okkamall.crl`, as well as, having much less frequency of certificate revocation. The motivation for having `okkamauthorities.crl` is to facilitate end-users (the most delicate use case) of efficient check for certificate revocation status when interacting with the ENS and with the ENS front-ends.

End users should install `okkamauthorities.crl` into their web browsers, email clients etc. OKKAM front-ends should install `okkamall.crl` into the application server when configuring to accept end-users certificates. OKKAM security proxy is configured to use the dedicated ENS API `isCertificateRevoked(...)` when interacting with the ENS.

The WS API `isCertificateRevoked(...)` provides the most up-to-date response (“freshness”) of certificate revocation status compared to the CRL approach. The WS API is not an OCSP responder but it is provided to serve the security proxy functionality. The WS API approach is not based on any underlying CRL data but uses directly the persistent storage data (where each revoked certificate has a single entry in the persistent storage). However, there are several cases where locally cached CRLs have more benefits rather than being constantly online connected to ENS for certificate status check. For example, the OKKAM front-ends use `okkamall.crl` when accepting HTTPS connections with end-user authentication.

Both CRLs above have been set to automatic update once per week, which gives reasonable freshness of certificate revocation status.

A dedicated GUI of the security proxy facilitates management of CRLs in OKKAM. Access to the respective APIs of certificate revocation is controlled access by administrators or ENS managers.

3.5.5. Using OKKAM security proxy inside java code

You have to download OKKAM security proxy from the download section of the community portal¹¹. Unzip it to a local folder of your project space. We refer to that folder as `PROXY_ROOT` folder. You need the following java libraries in your Java classpath:

- BouncyCastle security library (for Java 6 or later) allocated in `PROXY_ROOT/dist/lib/bcprov-jdk16-145.jar`
- iAccess authorization system library (v2.0 or later) allocated in `PROXY_ROOT/dist/lib/iAccessClient20beta007.jar`
- METRO libraries (only by v1.5¹²) allocated in `PROXY_ROOT/dist/lib/webservices-rt.jar` and `PROXY_ROOT/dist/lib/webservices-tools.jar`
- Okkam security proxy library (v1.13 or later) allocated in `PROXY_ROOT/dist/okkam-proxy.jar`

¹¹ <http://community.okkam.org/index.php/Downloads/Okkam-Security-Proxy/security-proxy-download.html>

¹² Metro libraries with version higher than v1.5 are not compatible with the current release of the security proxy. Unless otherwise stated, Metro v1.5 is the recommended version.

Below is the code example of how to use the proxy inside your java code:

```
01 import org.iaccess.accesscontrollayer.IAccessManager;
02 import org.okkam.proxyclient.security.iaccess.AuthorizationManager;
03 import org.okkam.proxyclient.security.iaccess.IAccessClientOkkam;
04 ...
10 String OKKAM_SERVER = "http://api.okkam.org/okkam-core/WebServices";
11
12 try
13 {
14     AuthorizationManager authManager = AuthorizationManager.getInstance();
15     // username and password from a user registration process.
16     authManager.initIAccess(username, password);
17     // At this point, all the certificates are to be loaded
18     AccessManager iAccessManager = authManager.getIAccessManager();
19     IAccessClientOkkam client = new IAccessClientOkkam(
20         iAccessManager, OKKAM_SERVER);
21
22     //For example when the client application wants to invoke createNewEntity() API
23     //with an input message. Input is a data object according to ENS APIs.
24     String result = client.createNewEntity(input);
25 }
26 catch(Exception e)
27 {
28     e.printStackTrace();
29 }
```

In this case, the okkam-proxy library will look for the user's keystore and his certificates in the PROXY_ROOT folder where the okkam-proxy application is executed. The PROXY_ROOT folder should contain the data folder where the user's keystore and certificates, plus other configuration files are located.

When the proxy is running in a Web environment, it may be difficult to know where the actual PROXY_ROOT folder is located. For this reason, it is possible to select the PROXY_ROOT folder when the AuthorizationManager instance is initialized. To do this, we only have to initialize the AuthorizationManager instance in the following way:

```
14 AuthorizationManager authManager = AuthorizationManager.getInstance();
15 // username and password from a user registration process.
16 authManager.initIAccess(username, password, PROXY_ROOT);
```

More detailed examples of how to use the Java Okkam Client (which in turn uses the security proxy) to connect to ENS APIs can be found at the community portal¹³.

3.5.6. How to run security proxy with already registered certificate data (P12 format)

If you have registered to OKKAM via the Web front-end (<http://register.okkam.org>), you can use your .p12 registration data directly in the security proxy application. You have to copy the .p12 file into the PROXY_ROOT/data/keystores/ folder and run your application. The proxy should automatically detect that the specified username corresponds to a .p12 file and will convert it into java keystore format for compatibility with Metro component.

¹³ <http://community.okkam.org/index.php/Documentation/APIs/java-okkam-client.html>

If you get an exception of illegal key size you should install “The Cryptography Extension (JCE) Unlimited Strength on your JVM”. Follow the link¹⁴ and download the policy file. Inside the zip you will find the steps to replace the policy file.

¹⁴ https://cds.sun.com/is-bin/INTERSHOP.enfinity/WFS/CDS-CDS_Developer-Site/en_US/-/USD/ViewProductDetail-Start?ProductRef=jce_policy-6-oth-JPR@CDS-CDS_Developer

4. Conclusions and future plans

This deliverable contains a developer's guide to the public and private OKKAM APIs. The guide reflects the functionality of the APIs as of June 2010 (the end of the OKKAM).

An organic part of this deliverable is the automatically generated javadoc documentation which is available at the OKKAM community portal <http://community.okkam.org>. The community portal also contains complementary information, in particular, community discussion forums.

5. References

- [1] <http://xerces.apache.org/xerces-j/>
- [2] <http://lucene.apache.org/java/docs/queryparsersyntax.html>
- [3] <http://lucene.apache.org/java/docs/scoring.html>

6. Appendix A: How to create an entity?

There are two possible ways to create an entity:

- through a graphical interface requiring human interaction
- or through the use of software services (APIs).

6.1. How to create an entity by hand?

An entity, as described in section 1, is a complex object that can be divided in subparts. Each subpart of the entity has a specific meaning and goal in making the entity easily retrievable. In this part of the document, a description of the process leading to the creation of a new entity object is described. In order to make this process explicit, a wizard-like web based application was defined. This application, named EnsWebToolkit, is available online at <https://api.okkam.org/EnsWebToolKit/>. The access to Entity Creator wizard is allowed only to registered user. For a description about how to become a registered user of the ENS, follow the procedure explained at <http://register.okkam.org/>.

The entity creation can be divided in the following steps:

1. The first step for creating an entity is to provide a description for it. A description is composed by a set of pairs <name,value>. This description must be detailed enough to make the entity discernable from others. In order to facilitate the choice of an adequate set of attributes, a list of 'default attributes' is associated with each of the entity types supported by the ENS. For example, the entity type 'location' has, as default, attributes with name *location_name*, *latitude*, *longitude*, *address*, etc. Nevertheless, the usage of default attributes in the description of an entity is not mandatory, and users are free of using an arbitrary set of attributes to describe the entity. The only mandatory requirement is that any attribute name must be a valid QName (<http://en.wikipedia.org/wiki/QName>).
2. The second step of entity creation is called 'attribute refinement'. In order to make less ambiguous the description of an entity, the user is asked to specify the entity identifier of the attributes previously defined. This task is extremely important to improve the quality of entity descriptions, as it removes any possible aleatory interpretation of the 'what the attribute value means'. To clarify, take the example of the description of a person. It is quite natural to use the name of the city of residence to identify a person. Unfortunately, the name of a city alone is not sufficient to describe it unambiguously. For example, there many 'London' in the world and it is impossible to understand which one the user is referring to by simply using the name 'London'. For this reason, the user can be asked to assign an entity id to the attribute value used to describe an entity. This can be done by performing a simple query to the ENS and select the entity corresponding to the attribute value used (i.e. specify which 'London' was meant).
3. The third step for the creation of an entity is to provide the known alternative identifiers for the entity. The alternative identifiers are valid URIs used to identify the entity in some system outside the ENS. Examples of such identifiers are the Linked-Data identifiers, the

DBpedia identifiers, or the DBLP Identifiers. It is important to notice that an alternative identifier is meant to be a proper identifier resolved in a RDF document describing the entity. Other type of identifiers, as the fiscal code, can be considered as very powerful description attributes.

4. The fourth step of entity creation is about providing known web documents presenting references about the entity. An example of reference could be the wikipedia page for the entity. The value of this entity must be a valid URL.
5. Once the entity description is complete, the user must first validate the entity description defined. The validation step has the goal of guaranteeing quality and uniqueness of the entity under creation, before actually creating it. Indeed, the validation step is divided mainly in two sub steps defining the necessary and sufficient conditions for the creation of a new entity. First the quality of an entity description is evaluated. If the quality of the entity is compliant with the requirements of the ENS, then the entity could be created. Second, the uniqueness of the entity description is evaluated by performing a duplication check. If the entity is proven to be unique and with adequate quality, then it can be created. Whereas, if the ENS finds that the entity under creation already exists, than validation fails and the creation cannot be completed. The duplicates found by the ENS are returned to the user that must then decide whether among them there is actually the entity under creation. If the set of duplicates does not contain the entity under creation, the user then should try to refine the entity description provided to make it more distinct with respect of the existing one. In extreme cases, when the validation fails due to reiterate false positive duplicate detection, it is possible to validate the entity skipping the uniqueness verification.
 1. OPTIONAL. If the validation task finds correctly that the entity under creation already exists in the ENS, the user can choose to EDIT the existing entity by extending its entity description. When chooses to EDIT an exiting entity, the only operation the user is allowed is to do is appending new attributes to the entity description. It is important to notice that NO EXISTING ATTRIBUTE can be edited. For the rest, the editing of an entity follows the same steps as the one for entity creation.
 1. OPTIONAL. If the validation task finds that the entity under editing (update) has a duplicated in the ENS, the user can either refine the entity description or choose to MERGE the two existing entities. The entity merge task consists in composing, given the existing entities' description, the description of a new entity presenting information from both. This operation is done with the goal of de-duplicate entities previously erroneously introduced in the ENS.

A tutorial about how to manually creating an entity using EnsWebToolKit is available at <http://community.okkam.org/index.php/Documentation/Tutorials/documentation-entity-creation-tutorial.html>.

6.2. How to create and entity through the APIs?

Currently, the Okkam project released an open source version of a Java client library for the ENS (<http://community.okkam.org/index.php/Home/NewsFlash/ensclient-java-oss.html>). This section aims at describing how to handle the creation of an Entity Object relying on this Java library.

It is important to notice that in principle the same operation can be performed in any other programming language. The only prerequisites to build a new client in any programming language, are the availability of skills for building automatically Web Service clients given a Web Service Description (WSDL), and the capability of performing marshalling/unmarshalling operation of XML documents. In fact, the ENS Web Service relies on XML documents serialized as strings are parameter (or response) to service requests.

For guide about how to create an entity using the Java Client APIs, you can follow the example presented at <http://community.okkam.org/index.php/Documentation/APIs/java-okkam-client.html>.

7. Appendix B: Entity Profile XML example

In this Appendix we supply documentation on the representation of an entity in the ENS. In the following paragraphs we give a sample of the XML representation of an OKKAM entity.

```
<?xml version="1.0" encoding="UTF-8"?>
  <a:entity xmlns:a="http://models.okkam.org/ENS-client-entity-schema.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://models.okkam.org/ENS-client-entity-schema.xsd
  http://models.okkam.org/ENS-client-entity-schema.xsd">
    <a:oid/>
    <a:alternativeIds>
    <a:alternativeId>
    <a:id>http://anotheridentifier/fromAnotherSystem</a:id>
    <a:metadata>
      <a:provenance>
        <a:source>http://mycompany.com</a:source>
        </a:provenance>
      </a:metadata>
    </a:alternativeId>
    </a:alternativeIds>
    <a:preferredId>mypreferredID</a:preferredId>
    <a:profile>
    <a:semanticType>person</a:semanticType>
    <a:attributes>
      <a:attribute> <a:name>name</a:name> <a:value>John Doe</a:value> </a:attribute>
      <a:attribute> <a:name>worksFor</a:name> <a:value>Nonexistent S.A.</a:value>
    </a:attribute>
      <a:attribute> <a:name>livesIn</a:name> <a:value>Neverland</a:value> </a:attribute>
    </a:attributes>
    </a:profile>
    <a:metadata> <a:provenance> <a:source>http://mycompany.com</a:source>
    </a:provenance> </a:metadata>
  </a:entity>
```

So, let's check what we have written, line by line (or block by block).

```
<?xml version="1.0" encoding="UTF-8"?> <a:entity xmlns:a="http://models.okkam.org/ENS-
client-entity-schema.xsd" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://models.okkam.org/ENS-client-entity-schema.xsd
http://models.okkam.org/ENS-client-entity-schema.xsd">
```

In these header lines we declare that we have a XML file, the encoding, the vocabulary that we will use and how we are allowed to use it. We also indicate that here, we start to describe an entity. These lines should be left unchanged in the general case.

```
<a:oid/>
```

We indicate that we do not already have an OKKAM ID (this is the usual case) for the entity we are describing.

```
<a:alternativeIds>
  <a:alternativeId> <a:id>http://anotheridentifier/fromAnotherSystem</a:id>
  <a:metadata> <a:provenance>
    <a:source>http://mycompany.com </a:source>
</a:provenance> </a:metadata>
  </a:alternativeId>
</a:alternativeIds>
<a:preferredId>mypreferredID</a:preferredId>
```

Here, I have indented the lines to indicate the structure. In this block we supply a set of alternative IDs (i.e., from other ID providers). Actually this set only describes one id, since there is only one ... block. Within this block however, we supply a multitude of information, namely: The alternative ID itself: `<a:id> http://anotheridentifier/fromAnotherSystem#idnum </a:id>` A set of metadata, describing who provided the information for this id. In this case the information was taken from a corporate site (mycompany.com).

```
<a:metadata>
  <a:provenance>
    <a:source>http://mycompany.com</a:source>
  </a:provenance>
</a:metadata>
```

The following lines end the specific alternative ID block, as well as the whole set of alternative IDs

block.

```
</a:alternativeId> </a:alternativeIds>
```

The following line indicates the preferred ID for the given item, in case we have several alternative IDs and we mostly want to use one of them as representative of the entity.

```
<a:preferredId>mypreferredID</a:preferredId>
```

And now we go to the actual attribute-value pairs of information about the entity. So if I am describing myself (John Doe working for Nonexistent SA and living in Neverland), I use three attributes: name, company, location, with the corresponding values. But first, let's indicate that we start to describe the profile of the entity:

```
<a:profile>
```

We also indicate that we are talking about a person:

```
<a:semanticType> person </a:semanticType>
```

The allowed entity types are:

1. **Person:** a concrete enduring entity with desires, intentions and beliefs (agent).
2. **Organization:** a concrete collective entity, whose members are intelligent agents. As a collection of agents that operate together, an organization can be considered an agentive entity, characterized from desires, intentions and beliefs.
3. **Event:** a concrete individual entity that happens in time, perdurant.
4. **Artifact instance:** a concrete entity intentionally created by an agent (or a group of agents working together) to serve some purpose or perform some function. An artifact is a non-agentive enduring.
5. **Artifact type:** the model of a product, or a work of art, in general an entity that can be reproduced and have copies or instances.
6. **Location:** a concrete individual entity that has a spatial extent, enduring.
7. **Other:** all the rest

Now we are supply this person's attributes:

```
<a:attributes>
  <a:attribute>
    <a:name>name</a:name>
    <a:value>John Doe</a:value>
  </a:attribute>
  <a:attribute>
    <a:name>worksFor</a:name>
    <a:value>Nonexistent S.A.</a:value>
  </a:attribute>
  <a:attribute>
    <a:name>livesIn</a:name>
    <a:value>Neverland</a:value>
  </a:attribute>
</a:attributes>
```

This ends our description

```
</a:profile>
```

but we have one more step: to provide the metadata for the whole entity and not for one of its fields.
Thus:

```
<a:metadata>
  <a:provenance>
    <a:source>http://mycompany.com </a:source>
  </a:provenance>
</a:metadata>
```

```
</a:entity>
```

Some additional information (more technical) on other or already presented fields:

- The alternative IDs are identifiers that other systems (external to OKKAM) have assigned to the same entity. By listing these identifiers in this field, we establish an identity connection among these identifiers. An alternative ID is a unique ID for an entity which is issued by a system or authority other than OKKAM. We distinguish between two types of AIDs:
 - dereferenceable AIDs: mainly RDF URIs which can be dereferenced through a HTTP call.
 - other AIDs: any other identifier which is issued by an external system / authority which is not an RDF URI.
- There is a preferred id field. This is the identifier that the entity prefers to be known by, and this is the identifier that OKKAM will return in response to an id query about the entity. The default value for this field is the OKKAM identifier.
- Entity metadata contains the Source of information (i.e., where we found this piece of information), and the Agent that produced these data (i.e., name of software, manual process, etc.).
- The access control metadata can contain the following fields: Displayable. This property take values either ALL when we want that people can view the entity, or NONE when we do not. Searchable. TRUE when people can find it as a result of a search operation. Otherwise, FALSE. Writable. TRUE when we want to allow others to alter it. FALSE otherwise. One may supply a set of external references that describe this entity.

More info can be found in the schema for entity representation (<http://models.okkam.org/ENS-client-entity-schema.xsd>) and in upcoming advanced tutorials, which will be made available at <http://community.okkam.org>.