ISMI
Developer's Guide
April, 2015

Table of Contents

Getting Starting....1
Setup....1
Setup of Openmind....1
Setup of ISMI....2
Deployment....2
Database Configuration....2
Packaging....2
Starting Tomcat....3
Data Model....3
Business Objects....3
Entity....3
Relation....4
OR Mapping....4
Database....7
Versioning....9
Forms....9
Own Value Generation....11
Search....11
Filtering....11
Simple Search....11
Authors Filters....11
Titles Filters....12

Getting Starting

Setup

Setup of Openmind

In order to extend or to fix the ismi-server, we must previously setup the Openmind-Project. Follow the following steps to setup this project:

Checkout the source code

```
svn co https://it-dev.mpiwg-berlin.mpg.de/svn/openmind/
```

```
cd openmind/
```

Install the framework called „hashMapping“ in your local maven repository.

```
mvn install:install-file -Dfile=~/openmind/lib/hashMapping.jar -DgroupId=cl.talca
-DartifactId=hashMapping -Dversion=1.0 -Dpackaging=jar
```

Install the Openmind framework in your local maven repository.

```
mvn install
```

Setup of ISMI

Now, you will be able to setup the ismi-server.

Checkout the source code:

```
svn co https://it-dev.mpiwg-berlin.mpg.de/svn/ismi-richfaces/
```

```
cd ismi-richfaces/
```

Database Setup :

The ISMI server stores the data in a MySQL database. The connection to the database is configured in src/main/resources/hibernate.cfg.xml.

URLs for local deployment

Some URLs used by the AJAX frontend are hard-coded in the sources. To run a local development instance you need to change the rest_url variable in src/main/webapp/imageServer/resources/js/diva4ismi.js.

Run ismi-server:

```
mvn tomcat7:run
```

Release:

```
mvn clean package
```

Deployment

Database Configuration

Before we generate the distributable war file of the ISMI web application, we have to check that the file hibernate.cfg.xml contains a valid MySQL account.

Packaging

The first step of the deployment is the compilation of the source code and the generation of the distributable war file. The following command will generate a war file called ' ismi-richfaces-1.0.war' within the folder target of the project.

```
mvn package
```

Starting Tomcat

Currently, ISMI is running in tuxserve01. In order to deploy the war file, you should copy it into your tomcat distribution. For tuxserve01, you should execute following command:

```
cp /home/workspace/ismi-richfaces/target/ismi-richfaces-1.0.war /usr/local/java/tomcat/webapps/om4-ismi.war
```

It is worth noting that until now we call the ISMI web application 'om4-ismi', therefore we must rename the war file from 'ismi-richfaces-1.0.war' to 'om4-ismi.war'.

Sometimes tomcat has problem trying to redeploy an application. For this reason, I suggest to restart tomcat, always when a new application is deployed.

For this, you can run following command:

```
sudo service tomcat7 restart
```

If this command does not run successfully, you can get the process id to kill it with this command:

```
ps aux | grep tomcat
```

Data Model

Business Objects

Openmind is database that stores data in form of a directed graph. In order to do this, Openmind uses basically the following three classes:

- org.mpi.openmind.repository.bo.Entity,
- org.mpi.openmind.repository.bo.Attribute and
- org.mpi.openmind.repository.bo.Relation.

Entity

An entity represents a node in the graph. This element can be related to other elements using relations and can be described using attributes. Figure 1 illustrates an entities and its attributes. The class Entity has a relation one-to-many to the class Attribute. The way to connect an entity with its attributes is using the field of the Attribute class called sourceId. This field references the identifier of the entity.

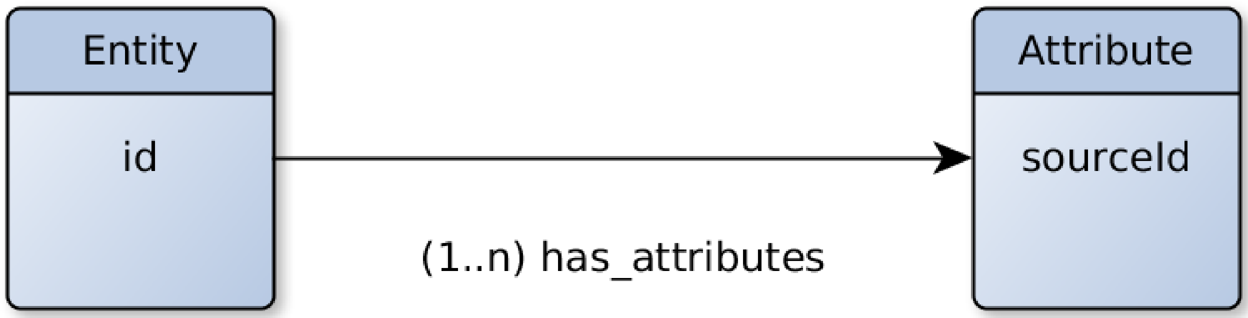

Figure 1. Representation of an entity and its attributes

Relation

A relation represents an edge in a graph. This class has the same attributes that an entity has plus sourceId and targetId to identify the entities that the relation links.

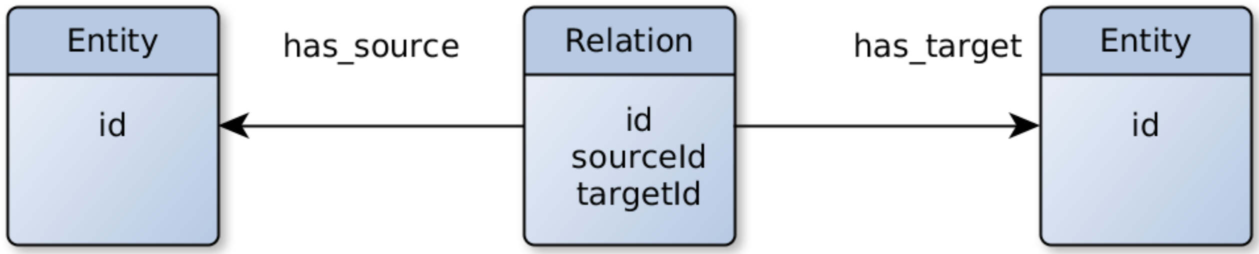

Figure 2. Representation of a relation

OR Mapping

Openmind uses Hibernate as persistent framework. Hibernate is responsible for the storage of the data in the database and for the loading of data from the database. Hibernate uses OR Mapping for the loading of data from the database. The current version of Openmind maps the following classes:

- org.mpi.openmind.repository.bo.Node
- org.mpi.openmind.repository.bo.Attribute
- org.mpi.openmind.repository.bo.Entity
- org.mpi.openmind.repository.bo.Relation
- org.mpi.openmind.repository.bo.View
- org.mpi.openmind.repository.bo.ViewerAttribute
- org.mpi.openmind.repository.bo.ViewerPage
- org.mpi.openmind.repository.bo.utils.Sequence
- org.mpi.openmind.security.bo.User

- org.mpi.openmind.security.bo.Group
- org.mpi.openmind.security.bo.Role
- org.mpi.openmind.security.bo.Permission
- org.mpi.openmind.security.bo.utils.UserRole
- org.mpi.openmind.security.bo.utils.GroupRole
- org.mpi.openmind.security.bo.utils.UserGroup
- org.mpi.openmind.security.bo.utils.RolePermission

Hibernate is able to map classes in two possible ways: using XML files and using Java Persistence API[1] (JPA). We decided to use JPA for the mapping of classes. JPA is a API based on annotations, it means that we must describe directly in the Java Class the mapping to the relational database. If we want to add a new class to the OR Mapping, we must modify the hibernate configuration file. This file is located in: project_root/src/main/resources/hibernate.cfg.xml. In this file, you add add a lines like this:

```
<mapping class="org.mpi.openmind.repository.bo.YourClass" />
```

As mentioned before, the classes Entity, Relation and Attribute are the most important in the Data Model. Figure 3 illustrates the relation between these three classes. This class diagram shows that these classes extend another class called Node. The Class Node contains attributes that are common for these three classes like rowId, id, objectClass, etc. If we take a look at the hibernate configuration file hibernate.cfg.xml, we will see that the class Node is also mapped. This is necessary, because we use as Inheritance Type the strategy called SINGLE_TABLE. It means that many classes with relations of generalization are stored in the same table (for more information see: https://docs.jboss.org/hibernate/orm/3.5/reference/en-US/html/inheritance.html).

The following table list the mapping between the Java Class and the database. It is worth to notice that these three classes are mapped to the same table in the data base. This table is called *node* and it is described in detail in the next Section.

1https://docs.jboss.org/hibernate/orm/3.6/quickstart/en-US/html/hibernate-gsg-tutorial-jpa.html

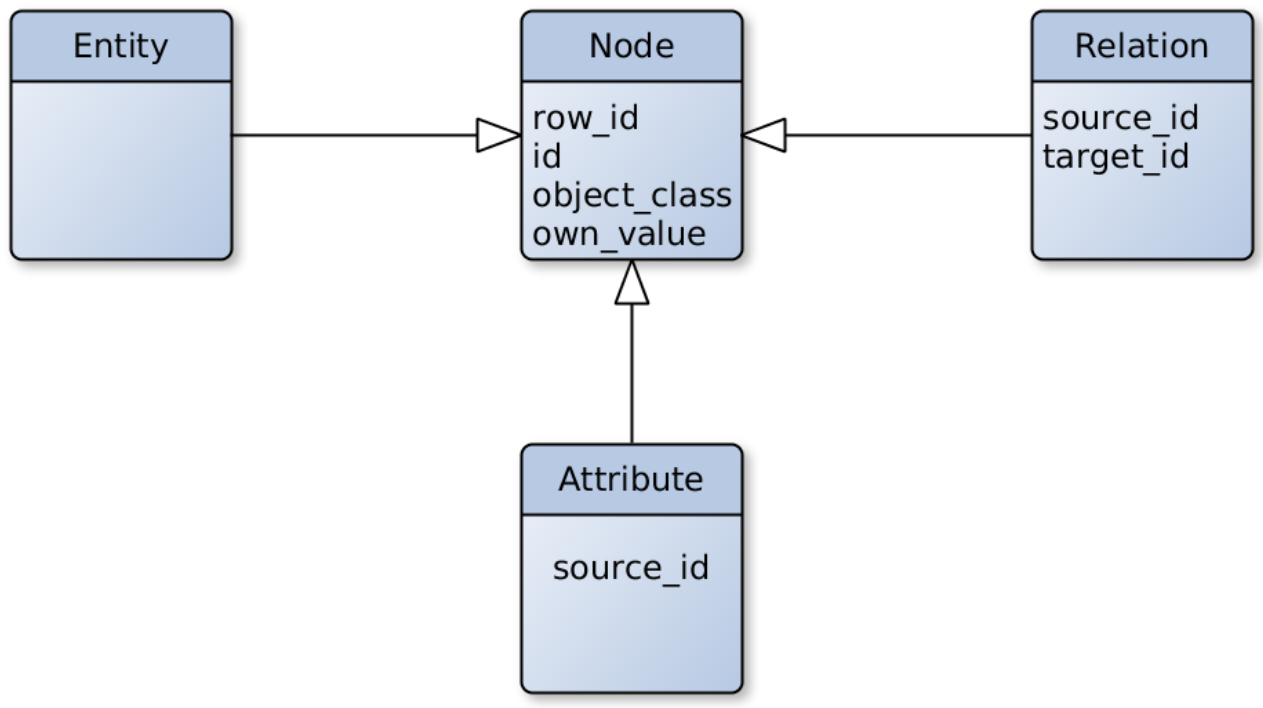

Figure 3. Class digram of Openmind.

Class Node	**Table node**
rowId	row_id
id	id
longValue	long_value
binaryValue	binary_value
objectClass	object_class
user	user
isPublic	public
ownValue	own_value
normalizedOwnValue	normalized_own_value
normalizedArabicOwnValue	normalized_arabic_own_value
version	version
status	status
systemStatus	system_status
type	type
contentType	content_type

Class Attribute	**Table node**
sourceId	source_id

sourceModif	source_modif
sourceObjectClass	source_obj_class

Class Relation	**Table node**
sourceId	source_id
sourceModif	source_modif
sourceObjectClass	source_obj_class
targetId	target_id
targetModif	target_modif
targetObjectClass	target_obj_class

Database

As we could see in the last Section, the most import table of the database is called *node*, because this table contains all the data related to the graph. In this context, the *node* table can be used to represent either as a node or as a edge. The following list describes each field of the mentioned table.

Table Node

- row_id: this is the identifier of the row. This value is generated automatically by the database.
- id: this is the identifier of an object. In the scope of the data model, an object can be an entity, a relation or an attribute. This value is generated by Openmind (see: org.mpi.openmind.repository.services.AbstractPersistenceService). Many row of this table can have the same id; in this case these rows should have different row_id, modification_time and version. These set of rows represents the different versions of the same object.
- node_type: this field is relevant for the data model. It indicates if the row represents an entity, a relation or an attribute.
- content_type: this field is relevant for the representation of the value of this node in web page. Until now, we did not agree a set of possible value, however normally we use this to distinguish between: plain text, arabic text, url, json and html.
- modification_time: this field stores the time stamp that indicates when this row has been saved.
- Type: Openmind is a graph that represents in parallel an ontology and the instances of this ontology. This field help to distinguish between both set of objects. This field is equal to TBox, when the row is part of the ontology's set, while this field is equal to ABox, when the row is part of the instances set. ABox and TBox are terms using in Description Logic. TBox

means terminological box and ABox means assertions box.

- object_class: the data model can distinguish between definitions and entities (or instances of definitions). On the one hand, the definitions are objects that describe the ontology. On the other hand, the entities are concrete instances of definitions. In the scope of Semantic Web, a definition is call *class* and an entity is called *object*.
- own_value: depending on the type of node, this field can contain different kind of value.
    1. If the node_type is ENTITY and the object_class is DEFINITION, then this field contains the name of a definition (like TEXT, PERSON, ALIAS, etc.).
    2. If the node_type is ENTITY and the object_class is the name of a definition (like TEXT, PERSON, ALIAS, etc.), then this value is not relevant.
    3. If the node_type is ATTRIBUTE, then this field contains the value of the attribute.
    4. If the node_type is RELATION, then this field contains the name of the relation.
- normalized_own_value: this field contains the same value of the field own_value after the execution of a normalization. The normalization is executed by Openmind (see: org.mpi.openmind.repository.utils.NormalizerUtils.java).
- normalized_arabic_own_value: this field contains the same value of the field own_value after the execution of an arabic normalization. The normalization is executed by Openmind (see: org.mpi.openmind.repository.utils.ArabicNormalizerUtils.java).
- system_status: this field can contains two possible values: PREVIOUS_VERSION and CURRENT_VERSION. We explained before that an object can have several versions, where the id is the same, but the version field is different. In order to get the last version of an certain object, we can just select the row, whose field system_status is equal to VERSION.
- version: this value indicates the version of the object. The bigger is this value, then newer is the obejct. The biggest value indicates the last version of the object. The last version of an object can be also found using the field system_status.
- source_id: this field is used in two cases:
    1. When the node_type is equal to RELATION. In this case, this field indicates the source of the relation. In this point, it is worth to remember, that Opemind represents a directed graph.
    2. When the node_type is equal to ATTRIBUTE. In this case, this field indicates the entity that is the owner of this attribute.
- source_modif: this field is used in the same way that the field source_id. This field contains the time stamp of the modification of the source. This value is useful to reconstruct an object by its version.

- source_obj_class: this field is used in the same way that the field source_id. This field is useful for the search methods.
- target_id: this field is only use, when the node_type of the row is equal to RELATION. In this case, this field indicates the id of the target of the relation.
- target_modif: this field is used in the same way that the field target_id. This field contains the time stamp of the modification of the target. It is used for the reconstruction an object by its version.
- target_obj_class: this field is used in the same way that the field target_id. This field is useful for the search methods.
- User: this field contains the user name that inserted this row in the database.
- possible_value: in some cases, we want to predefine a list of possible values for an attribute. In this case, these values can be stored in this field in JSON format.
- public: this field indicates if the content can be display in a public page or nor.

Versioning

TODO

Forms

A requirement of this project was the implementation of forms for all relevant definitions of the ISMI's data model. Figure 4 illustrates some the definitions and some of their relations. In principle, a central idea of Openmind was the automatic generation of a form for each definition, however, in the scope of ISMI, it was not possible, because the requirements for the forms became very complex and we were not able to encapsulate all these requirements. For this reason, we create tailored forms for the following definitions:

- Witness
- Codex
- Collection
- Repository
- Place
- Text
- Person
- Alias

- Subject
- Role
- Digitalization

Every Form is composed of a JSF page and a Java Bean. For example, the page for the Witness form is called witness.xhtml and the Bean is called CurrentWitnessBean.java. The same pattern is used for the the other definitions.

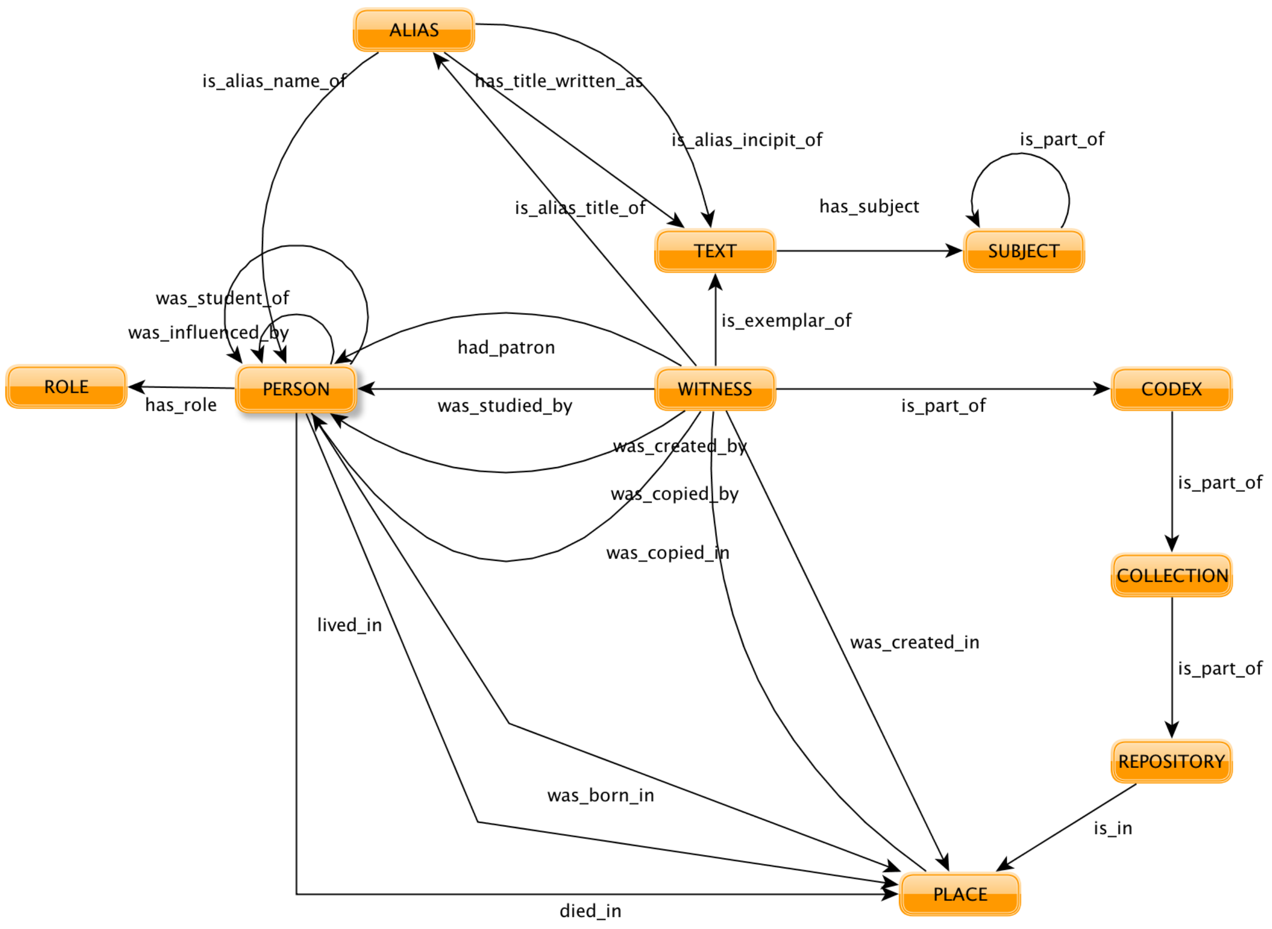

Figure 4. Simplification of the ISMI's data model.

Although we can not create a method that generates automatically forms for the definitions, there are several methods that are necessary for all forms. For this reason, all these common methods have been encapsulated in a common class called AbstractISMIBean and all Beans are extensions of this class.

The ISMI web page uses the Framework IceFaces for the generation of pages. If we create a new Bean, this Bean should be declared in the file faces-config.xml to be available in the JSF pages. This file is the configuration file of JSF and contains all the beans, some listeners and a set of navigation rules. More information related to this file can be found here: http://www.icesoft.org/java/projects/ICEfaces/documentation.jsf.

Own Value Generation

Always, when a entity is saved by Openmind, the system generates automatically a name for it. This name is generated from a set of rules that can be configured in the file own-value.cfg.xml. The execution of task is done in the class org.mpi.openmind.repository.utils.OwnValueGenerator.java before the entity is made persistent. When an instance of this class is created, the file own-value-cfg.xml is readed and all rules are saved in a HashMap. This class will use the first print rule found in the configuration file. The own-value-file is composed of two list of elements:

- own-value-rules: a own-value-rules is a rule that indicates the path to find a particular value. For example, the snippet above is a rule created for entities of class TEXT. This rule returns the value of the attribute called *alias*. This attribute belongs to a entity of class ALIAS that is connected to the root entity through the relation *is_prime_alias_title_of*.

```
<own-value-rule id="text0">
        <target-relation name="is_prime_alias_title_of" substring="false">
                <source object-class="ALIAS">
                        <attribute name="alias" end-node="true"/>
                </source>
        </target-relation>
</own-value-rule>
```

- print-rules: a print rule is a rule composed of several own-value-rules that generates a particular value. The print rule above is a rule that will be used for the class CODEX. This rule requires two own-value-rules that, for this example, are called *codex2* and *codex_identifier2*. These own-value-rules are used to print the final value that whose format is defined by the element *formantstr*.

```
<print-rule for="CODEX">
        <formatstr value="%codex2%_%codex_identifier2%"/>
        <entry own-value-rule="codex2"/>
        <entry own-value-rule="codex_identifier2"/>
</print-rule>
```

Search

Filtering

The class WrapperService has a method called searchEntityByAttributeFilter0. This method searches for entities using a list of filters and an string that we will call *search term*. A filter is implemented by the class org.mpi.openmind.repository.services.utils.AttributeFilter and it is basically composed of only two fields: *class* and *name*.

On the one hand, the field *class* indicates which kind of entities should be returned by the search (e.g. PERSON, WITNESS, etc.). On the other hand, the field name references one attribute of the entity. For example, if we are looking for entities of the class PERSON, the field name could be: name, name_translit, etc.

The search method will return a entity only if this entity matches at least one of the filters. A certain entity matches a filter only if:

1. the class of the entity is equal to the *class* of the filter,
2. this entity has a attribute X, whose name is equal to the *name* field of the filter and
3. the *search term* of the search method is a substring of the value of the attribute X.

The result set for this method is the union of the result set of each filter. It is worth to mention that the *search term* is normalized as well as every attribute in the database.

Simple Search

The simple search is a bean created to find either persons, titles or both. The page that uses this bean is called simpleSearch.xhtml and the class the implements these search methods is: de.mpiwg.itgroup.ismi.search.beans.SimpleSearchBean.java.

The Simple Search bean hard codes two set of attributes filters called:

- Authors Filters
- Titles Filters

These set were explicitly required by Jamil and Sally.

Authors Filters

These filters look for entities of the class PERSON, where at least one of the following attributes contain the string of the input search, in other words the input is a substring of of the the following attributes:

- PERSON.name
- PERSON.name_translit
- ALIAS.alias is_alias_of PERSON

Titles Filters

These filters look for entities of the class TEXT, where at least one of the following attributes contain the string of the input search:

- TEXT.title
- TEXT.title_translit
- TEXT.full_title_translit
- ALIAS.alias is_alias_of TEXT

The internal method of the class SimpleSearchBean that executes the search is called search0. The user of the system required a special format for the representation of the result of the search. This format can be easy observed by doing a search. In order to organize the result set in concordase with the requirements, we implemented the class de.mpiwg.itgroup.ismi.auxObjects.ResultSet. This class contains all the relevant information necessary for the representation of the result of the search.