caBIG_logo_w_tm

**caGRID**

**Identifier Framework**

***Design v. 0.1***

|  |  |
| --- | --- |
| ***Contacts and Support*** | |
| Calixto Melean (Developer) | Calixto.Melean@osumc.edu |
| Scott Oster (Lead Architect) | Scott.Oster@osumc.edu |
| Shannon Hastings (Architect) | Shannon.Hastings@osumc.edu |

|  |  |  |  |
| --- | --- | --- | --- |
| ***Revision History*** | | | |
| **Version** | **Date** | **Author** | **Changes** |
| **0.1** | 2009-08-04 | **Calixto Melean** | **Initial Draft** |
|  |  |  |  |

Chapter 1 Introduction 5

Introduction 5

Identifier Framework 5

Globally Unique Identifiers 5

Identifier and Data-Object Properties 5

Identifier Values / Metadata 6

Conceptual Model of Identifier Framework 6

1. The Data Owner 6
2. The Naming Authority 6
3. The Identifier Curator 7
4. The User 7
5. The Prefix Authority 7
6. Putting it all together 7

The Resolution Process 8

The Data Retrieval Process 8

Chapter 2 High Level Design 9

The Identifier 9

The Naming Authority (NA) 9

The Prefix Authority 10

1. Persistent Uniform Resource Locator (PURL) as a Prefix Authority 10

Partial-redirect PURL 10

1. PURL-based Identifiers 11

The Resolution Process 12

The Data Retrieval Process 15

Use Case 15

Chapter 3 Toolkit 17

Identifiers-NamingAuthority 17

Package org.cagrid.identifiers.core 17

1. IdentifierMaintainer Interface 17
2. IdentifierUser Interface 18
3. IdentifierValues Class 18

Package org.cagrid.identifiers.namingauthority.http 18

1. HttpProcessor Class 18

Resolution Request 18

Resolution Response 19

Retrieving Naming Authority Configuration 19

1. NamingAuthorityConfig Class 20
2. HttpServer Class 20

Package org.cagrid.identifiers.namingauthority 20

1. NamingAuthorityConfig Interface 20
2. IdentifierGenerator Interface 21
3. NamingAuthority Class 21
4. NamingAuthorityLoader Class 21

Package org.cagrid.identifiers.namingauthority.impl 22

1. NamingAuthorityConfigImpl Class 22
2. IdentifierGeneratorImpl Class 22
3. NamingAuthorityImpl Class 22
4. NamingAuthorityService Class 23

Package org.cagrid.identifiers.namingauthority.util 23

1. Database Class 23
2. HibernateUtil Class 23
3. IdentifierUtil Class 24

Package org.cagrid.identifiers.namingauthority.hibernate 24

1. IdentifierValue Class 24
2. IdentifierValue.hbm.xml 25
3. Identifiers.hibernate.cfg.xml 26

Deploying the Naming Authority 26

Identifiers-Client 27

Package org.cagrid.identifiers.resolver 27

1. ResolverUtil Class 27

Package org.cagrid.identifiers.retriever 27

1. Retriever Class 27
2. RetrieverFactory Interface 28

Package org.cagrid.identifiers.retriever.impl 28

1. DefaultRetrieverFactory Class 28
2. RetrieverService Class 28
3. CQLRetriever Class 29

Using Identifiers-Client to Resolve and Retrieve a Data Object 30

Identifiers-Resolver-Context.xml 30

Identifiers-NamingAuthority-GridSvc 31

Deployment 31

Schema 32

API 32

# Introduction

## Introduction

### Identifier Framework

The functionality provided by caGrid’s Identifier Services Framework is related to having “identifiers” for individual data-objects. The identifier is essentially a forever globally unique name for the data-object such that it can be unambiguously used to refer to the data from different application contexts.

In order to create, modify, delete the name-object bindings, facilities and services have to be defined and provided. Furthermore, in order to find the data-object when only the identifier is known, global resolution services have to be defined to resolve the name to the object.

### 

### Globally Unique Identifiers

Once we have standardized data-object identifiers that can be globally resolved to the data-objects themselves, applications can reason about and communicate data-objects by references instead of by value.

The identifiers also allow applications to test for data-object equality through identifier-string comparison. This property enables applications to bind arbitrary meta-data to the data-objects through the identifiers.

### Identifier and Data-Object Properties

The identifier is essentially a string and a forever globally unique name for a single data-object. Furthermore, the identifier can be (globally) resolved to an associated data-object.

In order to abstract the identifier’s object properties, the data service implementations and the resolution mechanisms, the identifier’s value must be treated as a “meaningless” opaque string by the consumer applications. Any leaking of design and implementation choices for the identifier framework in the applications, is undesirable from an architecture point of view as it makes the implementations brittle and susceptible to future changes. Of course resolution information will have to be embedded in identifier name, but this should only be meaningful for resolution service related components that are layered below the application.

The implementation choice for the identifier format is the Universal Resource Identifier (URI). This enables the use of existing web standards and protocols, and provides a natural approach to identifier resolution. No special knowledge is needed to know how to resolve identifiers. In other words, an identifier can be resolved by simply “following it”.

### Identifier Values / Metadata

The framework defines *Indentifier Values or Metadata* as any information stored with the identifier and typically used to help locate the target data-object that is being identified.

### Conceptual Model of Identifier Framework

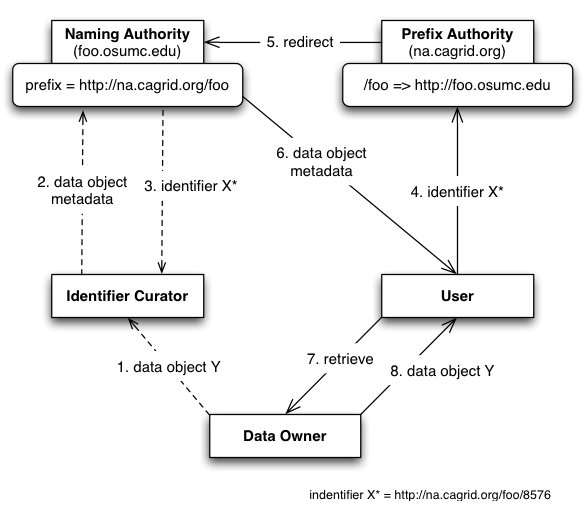


Figure Conceptual Model of Identifier Framework

The conceptual model of the identifier framework is depicted in Figure 1.

#### The Data Owner

This is the system or domain where the target data objects reside. These are the objects being identified (pointed at). *Data owners* specify how data objects are accessed when creating identifiers for their systems. The identifier framework uses this information to build tools that automate the retrieval of the corresponding data objects.

#### The Naming Authority

The *Naming Authority (NA)* is the entity that issues and manages identifiers and their metadata. This is the mapping between identifiers and data objects. A naming authority is identified by a prefix URI, which is part of the identifier as explained later.

#### The Identifier Curator

This entity is responsible for creating identifiers on behalf of the data owner. It could be the data owner itself. The curator is expected to understand the semantics of the data objects and knows how to retrieve data objects from the owner’s data service (e.g. Endpoint Reference). This information is sent to the naming authority represented as metadata.

#### The User

The *user* or consumer “somehow” has obtained access to the data-object’s identifier, and is interested in resolving the identifier and retrieving the data object.

#### The Prefix Authority

The *Prefix Authority* binds a domain/prefix to a *naming authority*. In Figure 1, the *prefix authority* *na.cagrid.org* binds the “*foo”* domain to the naming authority running at <http://foo.osumc.edu>. In other words, it binds the prefix <http://na.cagrid.org/foo> to <http://foo.osumc.edu>.

The prefix authority could maintain prefix binding for multiple naming authorities (e.g. a “*bar*” domain could be mapped to a naming authority running at <http://bar.osumc.edu>).

The identifiers framework does not require a prefix authority. However, for reasons discussed later, it is highly recommended. Figure 2 shows a model with no prefix authority. Notice in that case that the prefix reveals the naming authority location.

#### Putting it all together

Figure 1 shows how the actors cooperate to use the framework successfully. The data owner requests its curator to globally identify a new data object Y. The curator builds the metadata required to help retrieve data object Y later, and gives it to the naming authority as part of the “create identifier” request. The naming authority generates an identifier, stores the binding information, and returns the identifier to the curator. This completes the creation process.

Later, a user is given the identifier and wishes to retrieve the corresponding data object. Since the identifier is a URI that points to the *prefix authority*, it is simply “followed” to retrieve the associated metadata. The prefix authority notices the URL specifies the *foo* domain, and redirects the client (HTTP) to the correct naming authority. The naming authority responds to the HTTP GET request with the identifiers metadata. At this point, the metadata (e.g. EPR), can be used to retrieve the data object from the owner’s space.

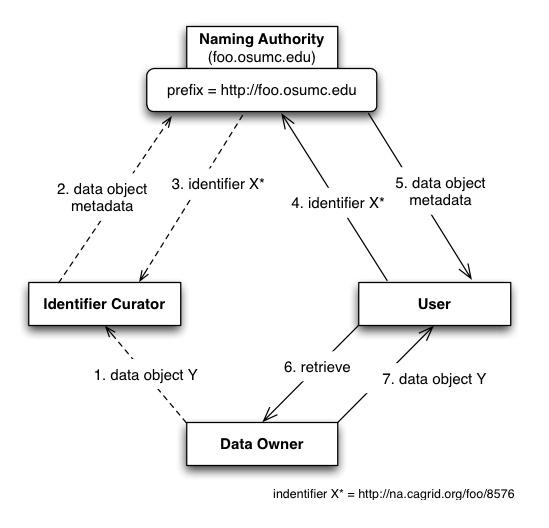


Figure Conceptual Model of Identifier Framework (No Prefix Authority)

### The Resolution Process

The framework defines *Resolution* as the process of finding the metadata associated or stored by a naming authority, given an identifier.

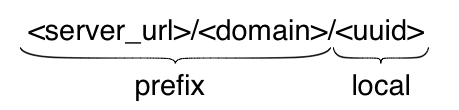
### The Data Retrieval Process

The framework defines Data Retrieval as the process of retrieving the object from the data owner space, giving an identifier’s values (metadata).

# High Level Design

## The Identifier

The general recommended structure of an identifier is as follows:



The purpose of the *prefix* is to uniquely identify the naming authority that hosts the identifier. The local part is unique within the naming authority, for example, a universally unique identifier (UUID).

## The Naming Authority (NA)

The NA maintains a database table of identifiers like the one shown below:

|  |  |  |
| --- | --- | --- |
| **Identifier** | **Data Type** | **Value** |
| <local\_identifier1> | EPR | <ns1:EndpointRerefence…> |
| <local\_identifier1> | CQL | <CQLQuery…> |

The table entries represent the metadata or identifier values associated with local identifier <local\_identifier1>[[1]](#footnote-3). As seen, identifiers can be associated with multiple resources or pieces of information.

The data type indicates the meaning that should be given to the data stored in the value column. This can be used by clients to decide on how data objects should be retrieved from their owners.

The framework’s default NA implementation serves identifier values (metadata) via HTTP. Values can be served in HTML format or serialized as XML. HTML is the default format, which is convenient for web browser users. Client programs request XML by setting the ACCEPT HTTP request header to “application/xml”.

The naming authority is deployed as a web application in servlet container, such as *Tomcat*.

## 

## The Prefix Authority

A likely use case is the potential move of the NA to a new location, with a different host name, or different port number. If this host information is used by identifiers as the *<server\_url>* component of the prefix, then the entire resolution process would be permanently impacted if the NA location were to change.

Therefore, even though the NA-issued identifiers are fully functional, they lack permanence/scalability. This is where a *prefix authority* comes to the rescue.

#### Persistent Uniform Resource Locator (PURL) as a Prefix Authority

“*A PURL is a URL that does not directly describe the location of the resource to be retrieved but instead describes an intermediate, more persistent location which, when retrieved, results in redirection to the current location of the final resource*.”

A PURL server maintains mappings that are used to match a request with a specific target location. For example:

* Suppose a PURL server can be is running at URL: *http://na.cagrid.org*
* Suppose a mapping is defined in the server as follows:
  + */illness/cancer.html* => *http://www.osumc.edu/illness/cancer.html*
* When a client (e.g. a web browser) attempts to navigate to *http://na.cagrid.org/illness/cancer.html*, the document *http://www.osumc.edu/illness/cancer.html* is retrieved.

In the example above, the target document *cancer.html* could be moved to a different URL, and could still be found by users if the mapping defined in the PURL server is updated to point to the new location.

##### Partial-redirect PURL

If the “osumc.edu” institution in the example above had a million known illnesses, using the above approach, a million mappings would have to be defined. This is where *partial redirects* help.

When a partial redirect is defined, the PURL server attempts to match as much of a URL as it can find in its database, and append the remainder (unmatched portion) to the end of the resolved URL.

For example:

* Supposed a **partial-redirect** is now defined as follows:
  + */illness* => *http://www.osumc.edu/illness*
* Now, when a client browses to *http://na.cagrid.org/illness/cancer.html*, the document *http://www.osumc.edu/illness/cancer.html* is retrieved.
* When a client browses to *http://na.cagrid.org/illness/swine-flu.html*, the document *http://www.osumc.edu/illness/swine-flu.html* is retrieved.

The partial redirect has allowed us to define the location of a million illnesses using a single mapping. Therefore, should all documents move to a different location, only one update has to be done in the PURL server.

#### PURL-based Identifiers

The above approach can be effectively used to protect the naming authority’s location; this is, by using a PURL server as the identifiers *prefix authority*.

The idea is to have the identifiers point to a prefix authority (PURL server), as opposed to pointing to the naming authority directly. For example:

* Suppose the naming authority runs at *http://foo.osumc.edu*
* Suppose a purl server runs at *http://na.cagrid.org*
* Suppose a partial-redirect PURL is defined as follows:
  + */foo* => *http://foo.osumc.edu*

Using the setup above, the NA’s prefix would be configured as *http://na.cagrid.org/foo*

Therefore, identifiers created by the NA would look like:

*http://na.cagrid.org/foo/c2581947-7c80-4330-9dd0-2761f6efdd41*

When such identifier is followed, the PURL server would redirect the client to:

*http://foo.osumc.edu/c2581947-7c80-4330-9dd0-2761f6efdd41*

The naming authority retrieves the identifier’s local name from the URL query string and looks up the metadata associated with it.

Should the NA move to a different URL, say *http://bar.osumc.edu*, the partial-redirect PURL has to be updated:

*/osumc* => *http://bar.osumc.edu*

The identifiers prefix remains the same; therefore, identifiers do not change, and the naming authority configuration does not change. In fact, nothing changes; except for the mapping in the prefix authority (PURL server).

## The Resolution Process

*Resolution* refers to discovering the identifier values (metadata) given a known identifier. As explained previously, an identifier can be resolved by “following it”, due to its HTTP-URI nature.

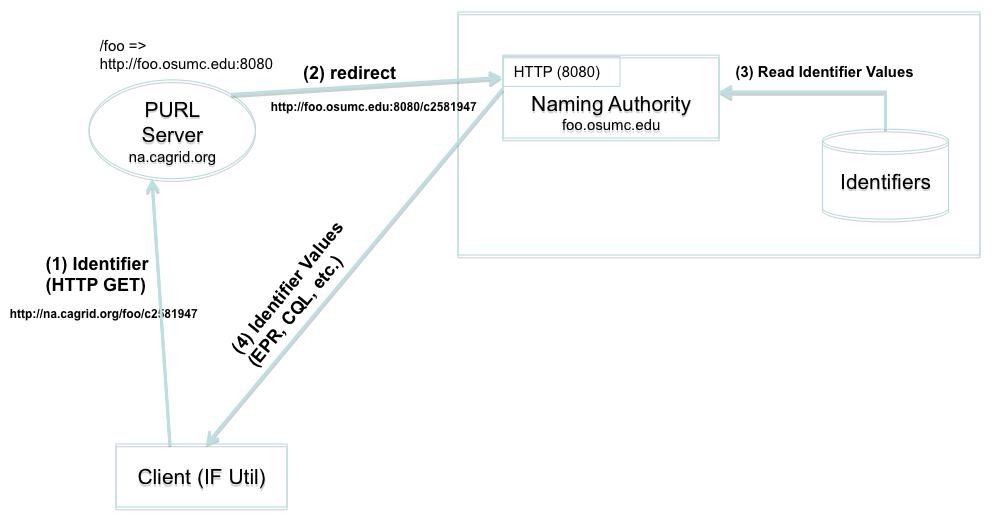


Figure HTTP Resolution

Figure 3 shows a resolution scenario using HTTP GET. When the identifier (URL) is followed, the PURL server redirects the request to the location (NA) that has been mapped. The NA extracts the local identifier name from the URL (*c2581947*), looks up the values from the identifiers table, and returns them to the client. As previously explained, the output of the response could be HTML or XML. Figure 4 shows a sample response as displayed by web browser.



Figure 4 HTTP Resolution (Web Browser)

A client could also use the framework’s grid service to resolve an identifier. Figure 5 depicts this scenario.

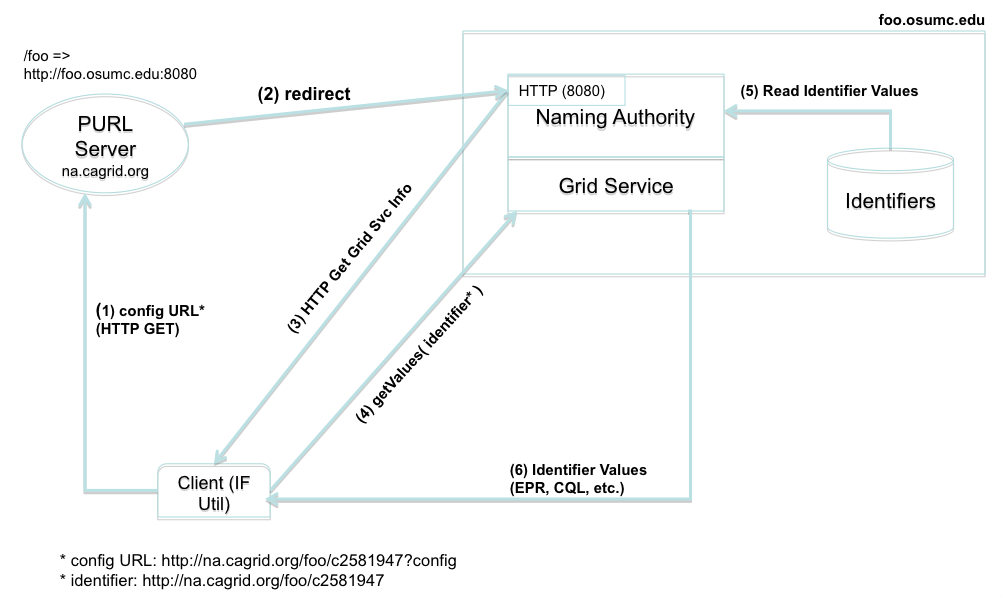


Figure Grid Resolution

The framework’s client retrieves the naming authority configuration by simply adding *?config* to the identifier URL and following it. The NA configuration is needed in this case in order to determine the grid service end point.

For example:

* A user wishes to resolve identifier *http://na.cagrid.org/foo/c2581947* using the grid service.
* Client toolkit appends an extra parameter (*?config*) to the query string, and issues an HTTP GET on *http://na.cagrid.org/foo/c2581947?config*
* PURL servers redirects to *http://foo.osumc.edu:8080/c2581947?config*
* NA recognizes that configuration is being requested and returns the information as XML. The response contains the location where the naming authority grid service is running, for example, *http://foo.osumc.edu/wsrf/services/cagrid/IdentifiersNAService*
* Client now acts as a grid service client and executes the *getValues* operation on the grid service located at the URL retrieved in the previous step.

## The Data Retrieval Process

This process involves retrieving the object from the data owner’s space, using the identifier metadata previously obtained from the resolution process.

The specifics of this process can’t be detailed in a generic way due to dependencies on the mechanisms made available by data owners to retrieve data from their space.

The framework retrieval process is driven by *retrieval profiles*. A profile defines two things:

* The metadata data types required to exist in the identifiers table maintained by the naming authority. Without these, the profile can’t be successfully executed.
* A formal definition of how to use the metadata to retrieve the data objects.

For example, consider a client system that associates the following two data types as values (metadata) for identifiers:

* ***EPR***: An end point reference. This includes the service address and port type of a deployed grid data server.
* ***CQL***: A string that can de-serialized into a CQL query.

A retrieval profile could be defined as requiring a *CQL* value and a *EPR* value from the identifier metadata, as well as a java implementation (say *CQLRetriever.java*) that effectively knows how to send the *CQL* query to the grid data service described by the *EPR*. In this example, the metadata is represented by the CQL and EPR values. The formal definition of how to use the metadata is represented by the implementation of a java class that makes use of the metadata.

The framework’s client toolkit provide aids in the definition of such profiles. In the example above, *CQLRetriever.java* implements the *Retriever* interface defined by the framework. Later in this document, we will see how profiles can be injected into the framework using the spring framework.

### Use Case

Figure 6 shows a use case where a data owner creates identifiers for *Person* objects that exists in a database application. The data owner provides access to the these objects via a grid data service.

A component in the data owner space (*TestModel Curator*) builds end point references (EPR) to the data service, and serialized CQL queries. This information is sent in the request to create an identifier to the naming authority. The naming authority creates the identifier and persists the EPR and CQL in the identifiers table as metadata. The identifier is returned to the client.

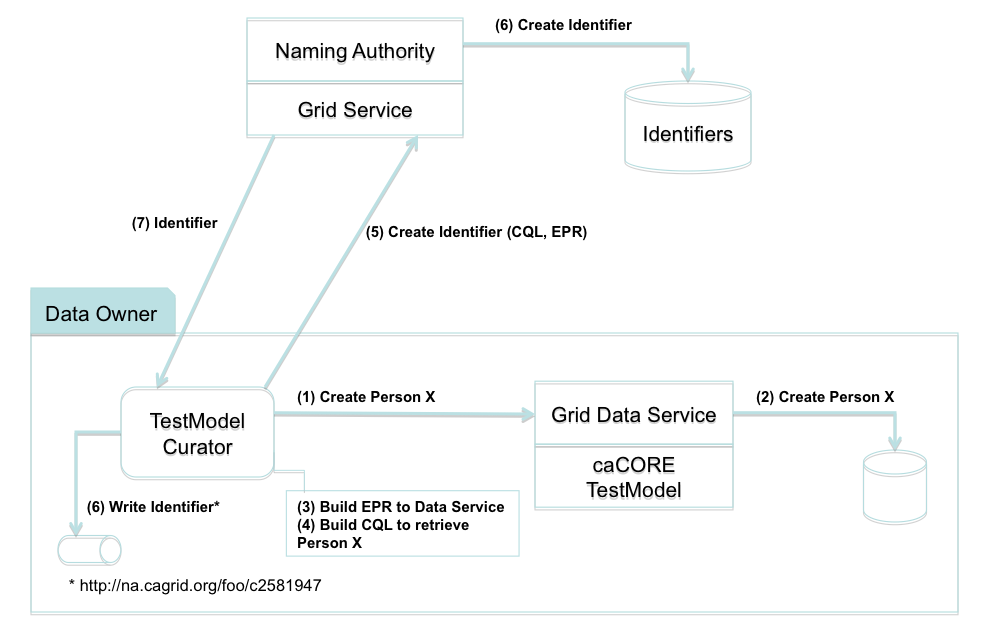


Figure Use Case: Creating Identifier

Figure 7 shows how the identifier is used to retrieve a person object. Steps 1 through 5 correspond to the *Resolution* process as described earlier. In step 6, the retriever class (*CQLRetriever.java*) de-serializes the CQL and EPR strings, and use the resulting java objects to make the call to the grid data service. The retriever interface returns the CQL result set to the *TestModel Curator*, where it can be further processed by “casting it” to the expected Person object.

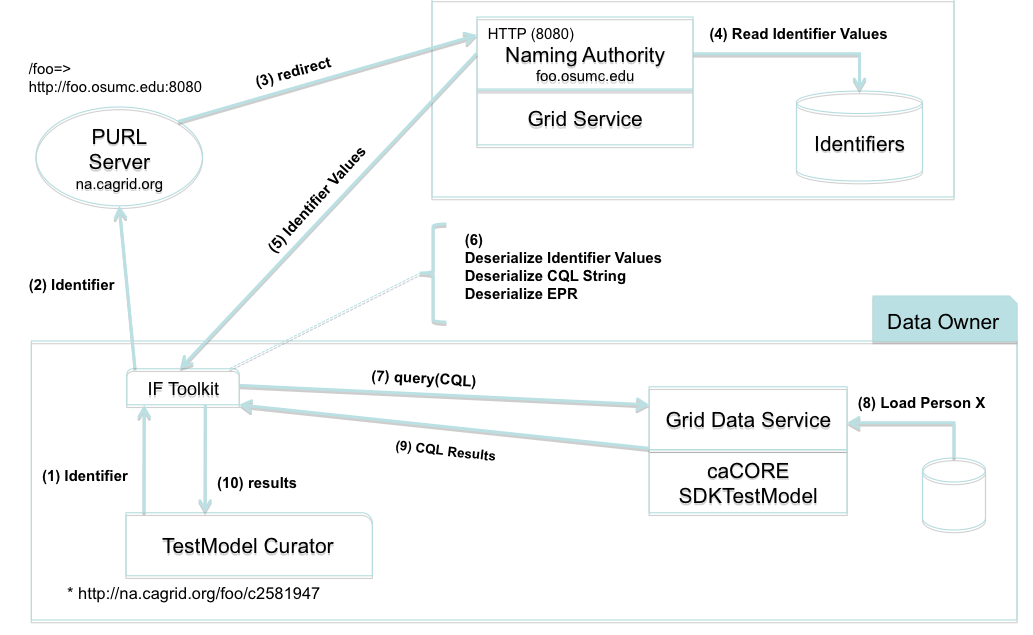


Figure Use Case: Data Retrieval

# Toolkit

The identifier framework is composed of 3 sub-projects: *identifiers-namingauthority*, *identifiers-namingauthority-gridsvc*, and *identifiers-client*.

## Identifiers-NamingAuthority

This is the naming authority source code and run-time components.

### Package org.cagrid.identifiers.core

#### IdentifierMaintainer Interface

|  |
| --- |
| public interface **IdentifierMaintainer** {  String *create*( IdentifierValues values );  } |

#### IdentifierUser Interface

|  |
| --- |
| public interface **IdentifierUser** {  IdentifierValues *getValues*( String identifier );  } |

#### IdentifierValues Class

|  |
| --- |
| public class **IdentifierValues** {  private HashMap<String, ArrayList<String>> values;  HashMap<String, ArrayList<String>> getValues();  void *setValues*( HashMap<String, ArrayList<String>> values );    String[] *getValues*( String type );    String[] *getTypes*();  void *add*(String type, String data);  } |

*IdentifierValues* maintains a hash map keyed by the data types associated with the identifier. For example, if a given identifier is associated with two end point references (EPR type), a map entry would look like:

“EPR” -> “EPR1”, “EPR2”

*getValues()* returns the entire map collection.

*getValues( type )* returns the list (map entry value) associated with the input type (map entry key).

*getTypes()* returns all data types associated with the identifier (map keys).

*add( type, data )* adds a new element (data) to the list pointed to by type in the map.

### Package org.cagrid.identifiers.namingauthority.http

#### HttpProcessor Class

This class handles HTTP requests from clients; including identifier resolution, and configuration.

##### Resolution Request

Resolution is requested by clients by using a URL of the form:

<*na\_server\_url*>/<*local\_identifier*>

For example, assuming the NA runs at *http://foo.osumc.edu:8080*, a local identifier *c893454* can be resolved by navigating to *http://foo.osumc.edu:8080/c893454*

##### Resolution Response

The resolution response contains the set of identifier values (metadata) associated with the identifier. The response format can be either HTML, or XML, which is chosen by examining the ACCEPT HTTP header in the request. The ACCEPT header typically contains a list of response formats that are acceptable by the client.

While testing identifier resolution using web browsers, it was found that they would set the ACCEPT header differently. For example, IE would indicate “\*/\*”, which means, all formats are acceptable; Firefox would list HTML first, then XML; Safari would list XML first, then HTML.

Therefore, in order to request XML, the ACCEPT header must contain *application/xml*, and must not include *text/html*. In other words, the presence of *text/html* or *\*/\** anywhere in the list would result in HTML.

HTML is also the default format. Therefore, an empty ACCEPT header would also result in HTML being returned.

Figure 4 shows a web browser view of a resolved identifier.

The XML response is a serialized view of the *IdentifierValues* object, which can be easily de-serialized as follows:

|  |
| --- |
| String response = …; // XML response from naming authority  XMLDecoder decoder = new XMLDecoder(new StringBufferInputStream(response));    IdentifierValues ivs = (IdentifierValues)decoder.readObject();  decoder.close(); |

*HttpProcessor* also supports a way to force XML response, which could be leveraged by web browser users for debugging purposes. This is accomplished by adding a *?xml* parameter to the resolution query string. For example:

*http://foo.osumc.edu:8080/c893454****?xml***

Web browsers may display XML responses differently. For example, in Safari, you may have to use the menu option *View->View Source* to be able to inspect the full XML response.

##### Retrieving Naming Authority Configuration

The naming authority public configuration object can be retrieved from the server via HTTP. This is accomplished by adding a *?config* parameter to the query string. For example:

*http://foo.osumc.edu:8080****?config***

The response is an XML document that can be de-serialized as follows:

|  |
| --- |
| import org.cagrid.identifiers.namingauthority.http.NamingAuthorityConfig;  String response = …; // XML response from naming authority  XMLDecoder decoder = new XMLDecoder(new StringBufferInputStream(response));    NamingAuthorityConfig ivs = (NamingAuthorityConfig)decoder.readObject();  decoder.close(); |

#### NamingAuthorityConfig Class

This class maintains public naming authority configuration. This can be retrieved from the naming authority via HTTP as seen in the previous section.

|  |
| --- |
| public class **NamingAuthorityConfig** {  String gridSvcUrl;  NamingAuthorityConfig();    NamingAuthorityConfig( org.cagrid.identifiers.namingauthority.NamingAuthorityConfig);    void setGridSvcUrl( String gridSvcUrl );  String getGridSvcUrl();  } |

Currently, the only available public configuration setting is the URL of the naming authority grid service (if deployed).

#### HttpServer Class

This class runs a Jetty HTTP server to offer resolution services and configuration information for debugging purposes and development environments, in cases where the naming authority can’t be deployed to servlet container, such as Tomcat.

### Package org.cagrid.identifiers.namingauthority

#### NamingAuthorityConfig Interface

|  |
| --- |
| public interface **NamingAuthorityConfig** {  String *getPrefix*();  String *getGridSvcUrl*();  } |

Naming authorities must have a URI prefix configured. The grid service is optional. Implementations must return null or the empty string if no grid service is deployed.

#### IdentifierGenerator Interface

|  |
| --- |
| public interface **IdentifierGenerator** {  String *generate( NamingAuthorityConfig )*;  } |

#### NamingAuthority Class

|  |
| --- |
| public **abstract** class **NamingAuthority** {  NamingAuthorityConfig configuration;  IdentifierGenerator identifierGenerator;  NamingAuthorityConfig *getConfiguration*();  void *setConfiguration*( NamingAuthorityConfig );  IdentifierGenerator *getIdentifierGenerator*();  void *setIdentifierGenerator*( IdentifierGenerator );  String *generateIdentifier*();  **abstract** void *initialize*();  } |

#### NamingAuthorityLoader Class

|  |
| --- |
| public class **NamingAuthorityLoader** {  ApplicationContext appCtx;  NamingAuthority namingAuthority;  *NamingAuthorityLoader*() {  init( new String[] {  "/resources/spring/identifiers-namingauthority-context.xml"},  "NamingAuthority");  }  *NamingAuthorityLoader*( String[] contextList, String authorityName ) {  init( contextList, authorityName );  }  void *init*( String[] contextList, String authorityName ) {  appCtx = new ClassPathXmlApplicationContext( contextList );  namingAuthority = (NamingAuthority) appCtx.getBean( authorityName );  }  NamingAuthority *getNamingAuthority*();  } |

### Package org.cagrid.identifiers.namingauthority.impl

#### NamingAuthorityConfigImpl Class

|  |
| --- |
| public interface **NamingAuthorityConfigImpl** implements *NamingAuthorityConfig* {  String prefix;  Integer httpServerPort;  String gridSvcUrl;    void *setPrefix*( String prefix );  String *getPrefix*();  void *setHttpServerPort*( Integer port );  Integer *getHttpServerPort*();  void *setGridSvcUrl*(String gridSvcUrl);  String *getGridSvcUrl*();  } |

#### IdentifierGeneratorImpl Class

|  |
| --- |
| public class **IdentifierGeneratorImpl** implements *IdentifierGenerator* {  String *generate*(NamingAuthorityConfig config) {  return java.util.UUID.*randomUUID*().toString();  }  } |

#### NamingAuthorityImpl Class

|  |
| --- |
| import org.cagrid.identifiers.namingauthority.http.HttpServer;  import org.cagrid.identifiers.namingauthority.util.Database;  import org.cagrid.identifiers.namingauthority.datatype.DataTypeService;  public class **NamingAuthorityImpl** extends **NamingAuthority**  implements IdentifierMaintainer, IdentifierUser {  HttpServer httpServer;  Database db;    void *initialize*();  Database *getDatabase*();  void *setDatabase*( Database );  String *create*( IdentifierValues );  IdentifierValues *getValues*( String identifier );  void startHttpServer();  } |

#### NamingAuthorityService Class

This is the servlet class used to stand up the naming authority. It creates a naming authority instance and uses it to serve HTTP requests as previously described.

### Package org.cagrid.identifiers.namingauthority.util

#### Database Class

The naming authority uses hibernate to manage the identifiers database. A single table *identifier\_values* is used. This class implements load and save operations.

|  |
| --- |
| class **Database** {  String dbUrl, dbUser, dbPassword;  String *getDbUrl*();  void *setDbUrl*( String );  String *getDbUser*();  void *setDbUser*( String );  String *getDbPassword*();  void *setDbPassword*( String );  void *initialize*();  void *save*( String identifier, IdentifierValues values );  IdentifierValues *getValues*( String identifier );  }; |

#### HibernateUtil Class

Use to manage Hibernate session factory.

|  |
| --- |
| class **HibernateUtil** {  HibernateUtil();    SessionFactory *initFactory*( String dbUrl, String dbUser, String dbPassword );  SessionFactory *getSessionFactory*();  } |

#### IdentifierUtil Class

This class provides utility functions that operate on identifiers.

|  |
| --- |
| class **IdentifierUtil** {  // builds the complete identifier given the prefix and the local name  String *build*( String prefix, String localName );  // extracts the local name from the full identifier string  String *getLocalName*( String prefix, String identifier );  } |

### Package org.cagrid.identifiers.namingauthority.hibernate

This packages contains the hibernate bean *IdentifierValue* and related configuration files.

#### IdentifierValue Class

|  |
| --- |
| class **IdentifierValue** {  *IdentifierValue*();  Long *getId*();  void *setId*(Long id);  String *getName*();  void *setName*(String name);  String *getType*();  void *setType*(String type);  String *getData*();  void *setData*(String data);  } |

#### IdentifierValue.hbm.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <!DOCTYPE hibernate-mapping PUBLIC  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">  <hibernate-mapping>  <class  name="org.cagrid.identifiers.namingauthority.hibernate.IdentifierValue"  table="IDENTIFIER\_VALUES">  <id name="id" column="ID">  <generator class="native"/>  </id>  <property name="name" not-null="true"/>  <property name="type" not-null="true"/>  <property name="data" not-null="true"/>  </class>  </hibernate-mapping> |

#### Identifiers.hibernate.cfg.xml

|  |
| --- |
| <?xml version='1.0' encoding='utf-8'?>  <!DOCTYPE hibernate-configuration PUBLIC  "-//Hibernate/Hibernate Configuration DTD 3.0//EN"  "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">  <hibernate-configuration>  <session-factory>  <!-- Database connection settings -->  <property name="connection.driver\_class">com.mysql.jdbc.Driver</property>  <property name="connection.url">jdbc:mysql://localhost/identifiers</property>  <property name="connection.username">root</property>  <property name="connection.password">rtwertwert</property>  <!-- JDBC connection pool (use the built-in) -->  <property name="connection.pool\_size">1</property>  <!-- SQL dialect -->  <property name="dialect">org.hibernate.dialect.MySQLDialect</property>  <!-- Enable Hibernate's automatic session context management -->  <property name="current\_session\_context\_class">thread</property>  <!-- Disable the second-level cache -->  <property name="cache.provider\_class">org.hibernate.cache.NoCacheProvider</property>  <!-- Echo all executed SQL to stdout -->  <property name="show\_sql">true</property>  <mapping resource="org/cagrid/identifiers/namingauthority/hibernate/IdentifierValue.hbm.xml"/>  </session-factory>  </hibernate-configuration> |

### Deploying the Naming Authority

1. Configure the naming authority by editing default settings in *<project\_home>/resources/spring/identifiers-namingauthority-context.xml*
2. *$ cd <project\_home>*
3. *$ ant deployTomcat*

## Identifiers-Client

This project provides a client toolkit that aids in identifier resolution and data object retrieval.

### Package org.cagrid.identifiers.resolver

#### ResolverUtil Class

This class provides utility methods to resolve an identifier. That is, given an identifier, retrieve the values (metadata) associated with it. Utility methods are available for doing HTTP-GET resolution (*resolveHttp*), or Grid-based resolution (*resolveGrid*).

|  |
| --- |
| class **ResolverUtil** {  IdentifierValues *resolveGrid*( String identifier );  IdentifierValues *resolveHttp*( String identifier );  NamingAuthorityConfig *getNamingAuthorityConfig*( String url );  } |

*resolveGrid* resolves the identifier using a grid client. This requires the framework’s grid service to be deployed. The grid client generated by the naming authority grid service is used to make the *getValues()* call exposed by that service.

*getNamingAuthorityConfig* retrieves the configuration object from the naming authority, which is necessary to determine the grid service URL. *getNamingAuthorityConfig* simply adds *?config* to the identifier and “follows” it.

### Package org.cagrid.identifiers.retriever

#### Retriever Class

This is an abstract class that defines the *retrieve* operation, whose purpose is to retrieve a data object from the owner’s space. A retriever object also has a list of data types which are required to be associated with the identifier in order to execute the *retrieve* operation successfully.

|  |
| --- |
| **abstract** class **Retriever** {  String[] requiredTypes;  **abstract** Object *retrieve*( IdentifierValues ivs );  String[] *getRequiredTypes*();  void *setRequiredTypes*( String[] types );  void *validateTypes*( IdentifierValues );  } |

#### RetrieverFactory Interface

Retriever factories must implement this interface in order to be used by the framework. One method allows the factory to pick the retriever that best matches the input *IdentifierValues*. The other method requires the name of the retriever as input.

|  |
| --- |
| interface **RetrieverFactory** {  Retriever *getRetriever*( IdentifierValues );  Retriever *getRetriever*( String retrieverName );  } |

### Package org.cagrid.identifiers.retriever.impl

#### DefaultRetrieverFactory Class

This is the default retriever factory implemented by the toolkit. It maintains a map of *Retriever* objects keyed by retriever name.

|  |
| --- |
| class **DefaultRetrieverFactory** implements **RetrieverFactory** {  Map<String, Retriever> retrievers;  *DefaultRetrieverFactory*(Map<String, Retriever> retrievers);  Retriever *getRetriever*( IdentifierValues ivs ) throws Exception {  throw new Exception("Not implemented yet");  }  Retriever *getRetriever*( String name ) throws Exception {  Retriever retriever = retrievers.get(name);  if (retriever == null)  throw new Exception("No retriever defined for [" + name + "]");  return retriever;  }  } |

#### RetrieverService Class

This class loads a *RetrieverFactory* from spring framework configuration file(s). The default constructor loads default retriever factory name and configuration files. The specialized constructor can be used to specify a different factory name and/or configuration files.

|  |
| --- |
| class **RetrieverService** {  ApplicationContext appCtx;  RetrieverFactory factory;    *RetrieverService*() {  init( new String[] {  **"/resources/spring/framework-resolver-context.xml"**,  **"/resources/spring/framework-namingauthority-context.xml**"},  "**RetrieverFactory**");  }    *RetrieverService*( String[] contextList, String factoryName ) {  init( contextList, factoryName );  }    void *init*( String[] contextList, String factoryName ) {  appCtx = new ClassPathXmlApplicationContext( contextList );  factory = (RetrieverFactory) appCtx.getBean( factoryName );  }    RetrieverFactory *getFactory*() {  return factory;  }  Object *retrieve*( String retrieverName, IdentifierValues ivs ) {  Retriever retriever = factory.getRetriever( retrieverName );  return retriever.retrieve(ivs);  }  Object *retrieve*( IdentifierValues ivs ) {  Retriever retriever = factory.getRetriever(ivs);  return retriever.retrieve(ivs);  }  } |

#### CQLRetriever Class

This is currently the only retriever built-in with the framework. It allows to query a grid data service and return a *CQLResultSet*.

|  |
| --- |
| class **CQLRetriever** extends **Retriever** {  Object retrieve( IdentifierValues ivs );  gov.nih.nci.cagrid.cqlresultset.CQLQueryResults  query(gov.nih.nci.cagrid.cqlquery.CQLQuery cqlQuery, String url,  String portName);  } |

### Using Identifiers-Client to Resolve and Retrieve a Data Object

|  |
| --- |
| // Resolution  IdentifierValues ivs = ResolverUtil.resolveHttp( identifierStr );  // Data Retrieval  RetrieverFactory factory = new RetrieverService().getFactory();  Retriever retriever = factory.getRetriever( “CQLRetriever” );  CQLQueryResults results = (CQLQueryResults) retriever.retrieve( ivs ); |

Or, a simplified way:

|  |
| --- |
| // Resolution  IdentifierValues ivs = ResolverUtil.resolveHttp( identifierStr );  // Data Retrieval  CQLQueryResults results =  (CQLQueryResults) new RetrieverService().retrieve( “CQLRetriever”, ivs ); |

In both cases, the first step is to resolve the identifier. That is, retrieve the identifier values (metadata).

The second overall step is to instantiate a *Retriever* object from the *RetrieverFactory*. The *RetrieverService* class loads a factory using the default spring configuration files. Other spring files can be used by using the specialized *RetrieverService* constructor.

Currently, a retriever name has to provided to the *getRetriever* method. The example requests *CQLRetriever*. There are plans to implement a *getRetriever* interface that can use a default algorithm to choose the “most appropriate” retriever based on the available identifier values.

The simplified code snippet shown above makes use of convenience method available in *RetrieverService* class that basically combines the three retrieval steps into one.

### Identifiers-Resolver-Context.xml

This spring framework file defines retrievers and factory. A retriever bean includes the implementation class as well as the data types that are required to be present with the identifier.

The *DefaultRetrieverFactory* is initialized with a map of retrievers it supports. Currently, it only has an entry for the CQLRetriever retriever.

|  |
| --- |
| <!-- CQLRetriever Retriever Profile -->  <bean id="**CQLRetriever**"  class="**org.cagrid.identifiers.retriever.impl.CQLRetriever**">  <property name="**requiredTypes**">  <util:list>  <value>**CQL**</value>  <value>**EPR**</value>  </util:list>  </property>  </bean>  <!-- End of Profiles -->    <bean id="**RetrieverFactory**"  class="**org.cagrid.identifiers.retriever.impl.DefaultRetrieverFactory**">  <constructor-arg>  <util:map>  <entry key="**CQLRetriever**">  <ref local="**CQLRetriever**"/>  </entry>  </util:map>  </constructor-arg>  </bean> |

## Identifiers-NamingAuthority-GridSvc

The framework implements a standard analytical grid service that runs the naming authority implementation described above. Even though deployment of this grid service is not required by the framework, it adds value to the naming authority web application:

* It provides the “write” interface necessary to manage/administer identifiers.
* It provides a fined-grained read interface (TBD).
* It implements security/authorization requirements (TBD).

### Deployment

1. Configure the naming authority by editing *caGrid/projects/identifiers-namingauthority/resources/spring/identifiers-namingauthoritydentifiers-namingauthority-context.xml.*

1. Re-build *caGrid* so that project dependencies are updated (*cd caGrid; ant all*).
2. cd caGrid/projects/identifiers-namingauthority-gridsvc
3. ant deployTomcat

### Schema

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <schema  targetNamespace="http://types.identifers-na.cagrid.org/IdentifiersNAService"  xmlns:tns="http://types.identifers-na.cagrid.org/IdentifiersNAService"  xmlns="http://www.w3.org/2001/XMLSchema">  <complexType name="Values">  <sequence>  <element name="Value" type="string" minOccurs="1"  maxOccurs="unbounded" />  </sequence>  </complexType>  <element name="Values" type="tns:Values" />  <complexType name="TypeValues">  <sequence>  <element name=”type” type=”string” />  <element ref="tns:Values" />  </sequence>  </complexType>  <element name="TypeValues" type="tns:TypeValues" />  <complexType name="TypeValuesMap">  a<sequence>  <element ref="tns:TypeValues" minOccurs="0" maxOccurs="unbounded" />  </sequence>  </complexType>  <element name="TypeValuesMap" type="tns:TypeValuesMap" />  <complexType name="Identifier">  <sequence>  <element type="string" name="Name"/>  <element ref="tns:TypeValuesMap"/>  </sequence>  </complexType>  <element name="Identifier" type="tns:Identifier"/>  </schema> |

### API

The grid service currently supports two operations:

|  |
| --- |
| String createIdentifier(gov.nih.nci.cagrid.identifiers.TypeValuesMap);  gov.nih.nci.cagrid.identifiers.TypeValuesMap getTypeValues(String identifier); |

1. The NA does not store the full identifier name; only the local name. [↑](#footnote-ref-3)