caBIG_logo_w_tm

**caGRID**

**Identifier Framework**

***Design***

|  |  |
| --- | --- |
| ***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 |

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

Protecting the Naming Authority 9

1. Persistent Uniform Resource Locator (PURL) 10

Partial-redirect PURL 10

1. PURL-based Identifiers 10

Running the Naming Authority 11

The Resolution Process 11

The Data Retrieval Process 15

Use Case 15

Chapter 3 Toolkit 17

Framework-Core 17

Package org.cagrid.identifiers.core 17

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

Framework-NamingAuthority 18

Package org.cagrid.identifiers.namingauthority.http 18

1. HttpServer Class 18

Resolution Request 18

Resolution Response 19

1. NamingAuthorityConfig Class 20

Package org.cagrid.identifiers.namingauthority 20

1. NamingAuthorityConfig Interface 20
2. NamingAuthority Class 21

Package org.cagrid.identifiers.namingauthority.impl 21

1. NamingAuthorityConfigImpl Class 21
2. NamingAuthorityImpl Class 21
3. NamingAuthorityService Class 22

Package org.cagrid.identifiers.namingauthority.datatype 22

1. DataType Class 22
2. DataTypeFactory Class 23
3. DataTypeService Class 23
4. Data Types and Factory Configuration 23

Package org.cagrid.identifiers.namingauthority.util 24

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

Package org.cagrid.identifiers.namingauthority.hibernate 25

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

Creating a Naming Authority Instance 27

Deploying the Naming Authority as a Web Application 27

Framework-Resolver 27

Package org.cagrid.identifiers.resolver 27

1. ResolverUtil Class 27

Package org.cagrid.identifiers.retriever 28

1. Retriever Class 28
2. RetrieverFactory Interface 28

Package org.cagrid.identifiers.retriever.impl 29

1. DefaultRetrieverFactory Class 29
2. RetrieverService Class 29
3. CQLRetriever Class 30

Using Framework-Resolver to Resolve and Retrieve a Data Object 31

Framework-resolver-context.xml 31

Framework-NamingAuthority-GridSvc 32

Deployment 32

Schema 33

API 34

# 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

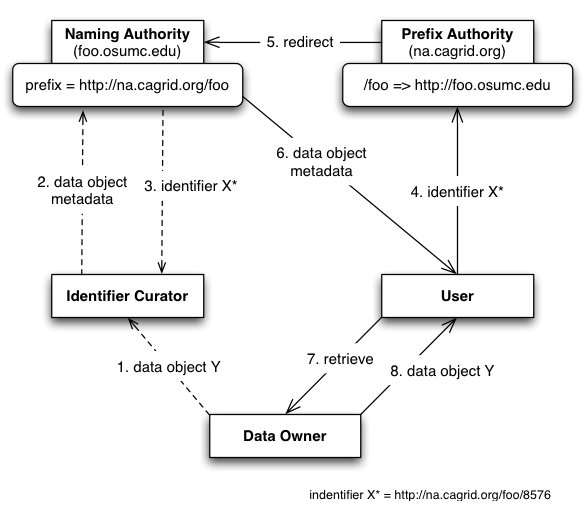


Figure Conceptual Model of Identifier Framework

The conceptual model of the identifier framework is depicted in .

#### 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. shows a model with no prefix authority. Notice in that case that the prefix reveals the naming authority location.

#### Putting it all together

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.

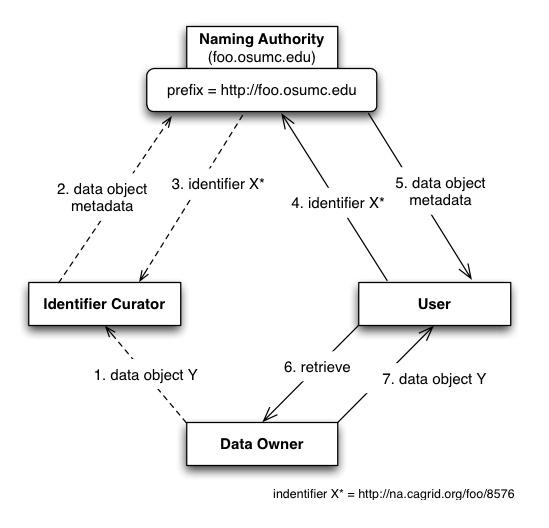


Figure Conceptual Model of Identifier Framework (No Prefix Authority)

### The Resolution Process

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

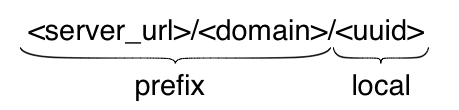
### The Data Retrieval Process

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** |
| <identifier1> | EPR | <ns1:EndpointRerefence…> |
| <identifier1> | CQL | <CQLQuery…> |

The table entries represent the metadata or identifier values associated with identifier <identifier1>. 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. The NA is configured to support a specific set of data types.

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”.

### Protecting the Naming 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 the 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. The recommended approach to address this is the use of PURL.

#### Persistent Uniform Resource Locator (PURL)

“*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://purl.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://purl.cagrid.org/illness/cancer.html>, the document <http://www.osumc.edu/illness/cancer.html> is automatically 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, then using the above approach, a million definitions would have to be defined in the PURL server. 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://purl.cagrid.org/illness/cancer.html>, the document <http://www.osumc.edu/illness/cancer.html> is retrieved.
* When a client browses to <http://purl.cagrid.org/illness/swine-flu.html>, the document http://www.osumc.edu/illness/swine-flu.html is retrieved.

The partial redirect we’ve defined have allowed us to define the location of a million illnesses using a single mapping. Therefore, should they all 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. The idea is to have the identifiers point to a PURL server, as opposed to pointing to the naming authority directly. For example:

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

Using the setup above, the NA’s prefix would be configured as <http://purl.cagrid.org/osumc>

Therefore, identifiers created by the NA would look like:

<http://purl.cagrid.org/osumc/c2581947-7c80-4330-9dd0-2761f6efdd41>

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

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

The naming authority already knows its prefix; with that and the local part name provided in the URL query string, it can lookup the correct identifier and return the corresponding metadata.

Should the NA move to a different URL, say <http://new.na.cagrid.org>, only the partial redirect URL has to be updated:

/osumc => <http://new.na.cagrid.org>

### Running the Naming Authority

The identifier framework provides two ways to stand up the naming authority.

The NA can be deployed as a *webapp* to a *servlet* container, such as *Tomcat*. A NA servlet bootstraps the naming authority and forwards all HTTP requests to the NA’s built in HTTP port.

Alternatively, the framework also includes a standard analytical grid service that starts the naming authority and exposes a grid interface to access naming authority operations.

## 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.

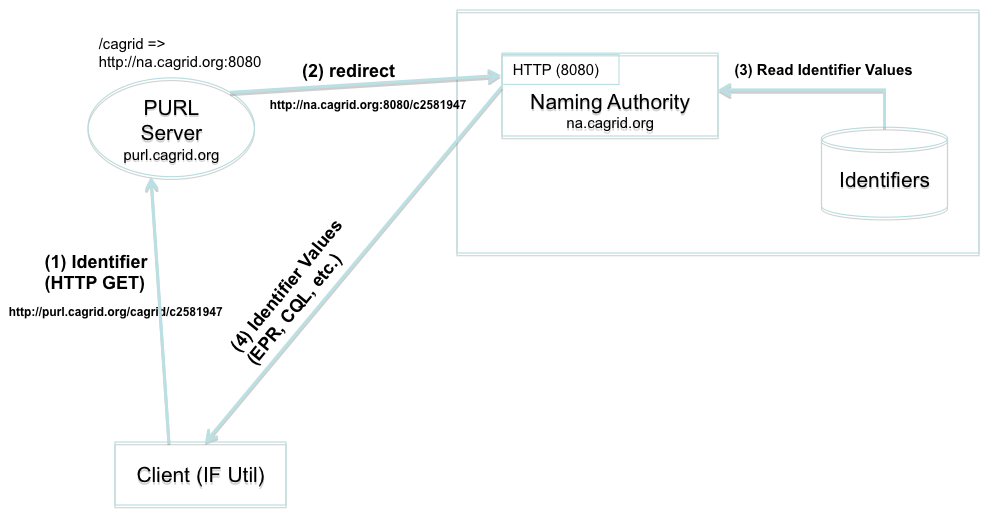


Figure HTTP Resolution

shows a resolution scenario using the naming authority’s built in HTTP port. When the identifier (URL) is followed (client simply issues a HTTP-GET), the PURL server redirects the request to the location (NA) that has been mapped. The NA builds the full identifier by appending the local name (*c2581947*) provided to its well known prefix (*http://purl.cagrid.org/cagrid*), 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. shows a sample response as displayed by web browser.



Figure 4 HTTP Resolution (Web Browser)

shows a NA running under the framework’s grid service and a client using the grid service to resolve an identifier.

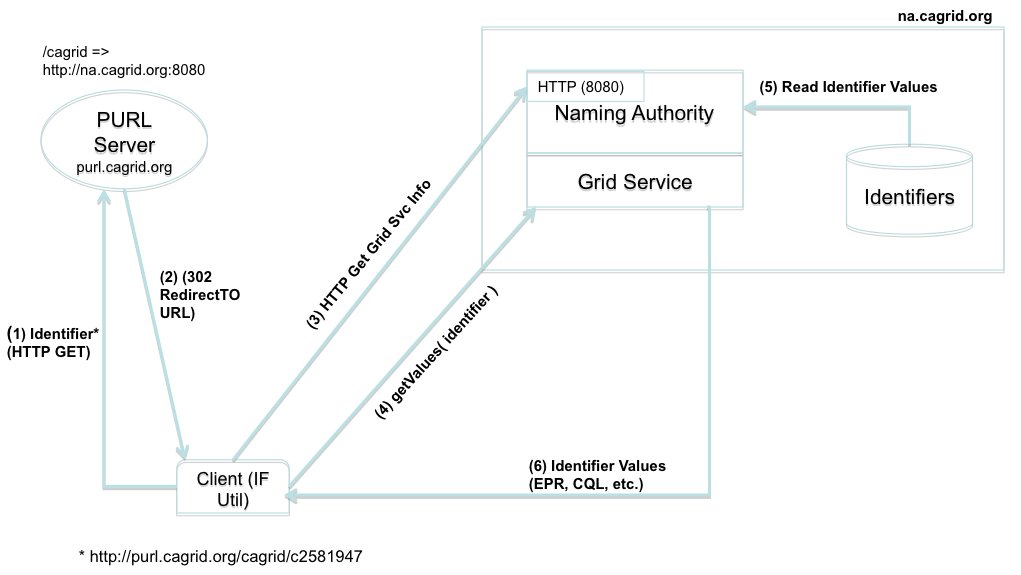


Figure Grid Resolution

The framework’s http client is configured to handle HTTP redirects itself, otherwise the underlying protocol library would follow the redirect automatically and issue an HTTP GET to the naming authority, which is not desired in this case. Instead, the client uses the target location returned by PURL to request the naming authority’s configuration from the naming authority. For example:

* The client starts by issuing an HTTP GET on <http://purl.cagrid.org/cagrid/c2581947>
* A redirect response is received, with target location [http://na.cagrid.org:8080/cagrid/c258194](http://na.cagrid.org:8080/cagrid/c2581949)7
* Client appends an extra parameter (?config) to query string, and issue an HTTP GET on <http://na.cagrid.org:8080/cagrid/c2581947?config>
* The 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://na.cagrid.org/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 as they depend on the mechanisms made available by the data owners to retrieve data from their space.

The identifier 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.
* An specific java implementation that knows how to use the required metadata to retrieve the data object.

For example, consider a naming authority that supports 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*.

*CQLRetriever.java* must implement 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

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.

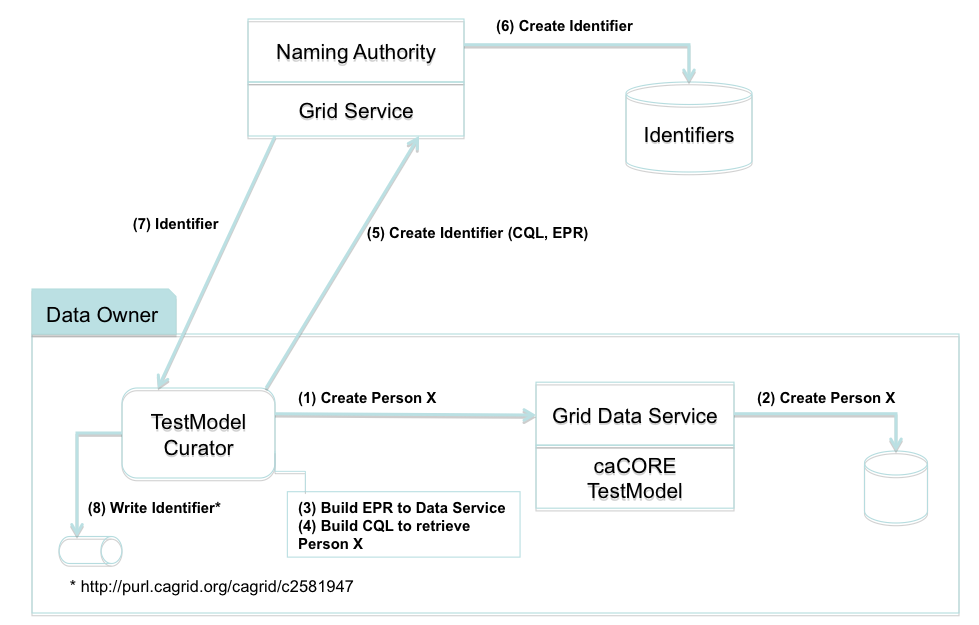


Figure Use Case: Creating Identifier

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.

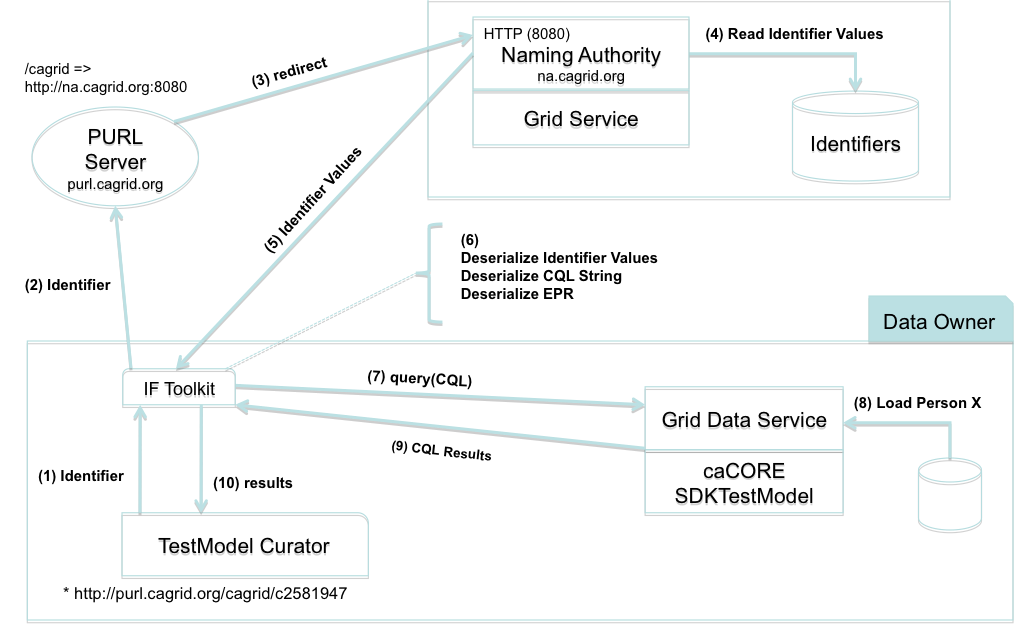


Figure Use Case: Data Retrieval

# Toolkit

The identifier framework is composed of 4 sub-projects: *framework-core*, *framework-namingauthority*, *framework-namingauthority-gridsvc*, and *framework-resolver*.

## Framework-Core

This project contains a set of core classes that all other projects depend on.

### 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.

## Framework-NamingAuthority

This is the naming authority source code and runtime components.

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

#### HttpServer Class

The naming authority runs a Jetty http server to offer resolution services and configuration information.

##### 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://na.cagrid.org:8080>, a local identifier *c893454* can be resolved by navigating to <http://na.cagrid.org:8080/c893454>

##### Resolution Response

The response contains the set of identifier values (metadata) associated with the identifier. The response format can be either HTML, or XML.

The response format 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, we 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 2 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(); |

We’ve seen that clients must set header to application/xml in order to request XML-serialized identifier values. *HttpServer* 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://na.cagrid.org: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 accomplish by adding a *config* parameter to the query string. For example:

<http://na.cagrid.org: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 HTTP server 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).

### Package org.cagrid.identifiers.namingauthority

#### NamingAuthorityConfig Interface

|  |
| --- |
| public interface **NamingAuthorityConfig** {  String getPrefix();  Integer getHttpServerPort();  String getGridSvcUrl();  String getDbUserName();  String getDbPassword();  String getDbUrl();  } |

#### NamingAuthority Class

|  |
| --- |
| public **abstract** class **NamingAuthority** {  NamingAuthorityConfig config;  NamingAuthority( NamingAuthorityConfig config );  NamingAuthorityConfig getConfig();  } |

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

#### NamingAuthorityConfigImpl Class

|  |
| --- |
| public interface **NamingAuthorityConfigImpl** implements NamingAuthorityConfig {  String prefix;  Integer httpServerPort;  String gridSvcUrl;  String dbUrl;  String dbUser;  String dbPassword;    void setPrefix( String prefix );  String getPrefix();  void setHttpServerPort( Integer port );  Integer getHttpServerPort();  void setGridSvcUrl(String gridSvcUrl);  String getGridSvcUrl();  void setDbUrl( String dbUrl );  String getDbUrl();  void setDbUser( String user );  String getDbUserName();  void setDbPassword( String pwd );  String getDbPassword();  } |

#### 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;  DataTypeService dataTypeSvc;    public NamingAuthorityImpl(NamingAuthorityConfig config) {  super(config);    // Initialize Database factory  db = new Database(config.getDbUrl(), config.getDbUserName(),  config.getDbPassword());    // Initialize data types factory  dataTypeSvc = new DataTypeService();  }  String create(IdentifierValues values);  IdentifierValues getValues( String identifier );  void startHttpServer();  } |

#### NamingAuthorityService Class

This is the naming authority servlet. This is used when the naming authority is deployed as a web application in a container such as Tomcat.

The servlet starts the naming authority using a configuration created from servlet initialization parameters read from *<project\_home>/WebContent/WEB-INF/web.xml*.

This servlet redirects all HTTP GET requests to the naming authorities, built-in HTTP port.

### Package org.cagrid.identifiers.namingauthority.datatype

The naming authority is configured to support a specific set of data types in the identifiers table.

#### DataType Class

|  |
| --- |
| String name;  String description;  String getName();  void setName( String name );  String getDescription();  void setDescription( String description ); |

#### DataTypeFactory Class

This class maintains the list of supported data type names.

|  |
| --- |
| private List<String> types;  DataTypeFactory( List<String> types );  List<String> getTypes();  boolean containsType( String type ); |

#### DataTypeService Class

This class loads a *DataTypeFactory* from spring framework configuration file(s). The default constructor loads default configuration files. The specialized constructor can be used to specify different configuration files.

|  |
| --- |
| ApplicationContext appCtx;  DataTypeFactory factory;  DataTypeService();  DataTypeService( String[] contextList, String factoryName );  DataTypeFactory getFactory(); |

#### Data Types and Factory Configuration

The configuration is driven by a spring framework context file as shown in Figure 6. Each supported data type is defined as a bean of type *DataType*. The default factory name *DataTypeFactory* is also defined here. It is initialized with the data type names previously defined in the same file.

Therefore, adding support for an additional data type is fairly simple. A new bean id would be entered in the file, and then referenced from the factory.

|  |
| --- |
| <!-- Data Types -->  <bean id="**EPRType**"  class="org.cagrid.identifiers.namingauthority.datatype.DataType">  <property name="name" value="**EPR**" />  <property name="description" value="A serialized org.apache.axis.message.addressing.EndpointReferenceType" />  </bean>  <bean id="**CQLType**"  class="org.cagrid.identifiers.namingauthority.datatype.DataType">  <property name="name" value="**CQL**" />  <property name="description" value="A serialized gov.nih.nci.cagrid.cqlquery.CQLQuery" />  </bean>  <!-- End of Data Types -->    <bean id="**DataTypeFactory**"  class="org.cagrid.identifiers.namingauthority.datatype.DataTypeFactory">  <constructor-arg>  <util:list>  <bean id="**CQLType.name**"  class="org.springframework.beans.factory.config.PropertyPathFactoryBean"/>  <bean id="**EPRType.name**"  class="org.springframework.beans.factory.config.PropertyPathFactoryBean"/>  </util:list>  </constructor-arg>  </bean> |

Figure 8 framework-namingauthority-context.xml

### 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.

|  |
| --- |
| Database( String dbUrl, String dbUser, String dbPassword );  void save( String identifier, IdentifierValues values );  IdentifierValues getValues( String identifier ); |

#### HibernateUtil Class

Use to manage Hibernate session factory.

|  |
| --- |
| HibernateUtil()    SessionFactory initFactory( String dbUrl, String dbUser, String dbPassword );  SessionFactory getSessionFactory(); |

#### IdentifierUtil Class

This class generates identifier strings. Local identifiers are generated using *java.util.UUID.randomUUID()*.

|  |
| --- |
| String generate( String prefix );  String generate( String prefix, String localId ); |

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

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

#### IdentifierValue Class

|  |
| --- |
| 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> |

### Creating a Naming Authority Instance

|  |
| --- |
| // Create NA configuration object  NamingAuthorityConfigImpl naConfig = new NamingAuthorityConfigImpl();  naConfig.setPrefix( “http://purl.cagrid.org/cagrid” );  naConfig.setHttpServerPort( 8080 );  naConfig.setDbUrl( “jdbc:mysql://localhost/identifiers” );  naConfig.setDbUser( “root” );  naConfig.setDbPassword( “soxpride” );  naConfig.setGridSvcUrl(  “http://localhost:8080/wsrf/services/cagrid/IdentifiersNAService” );  // Create NA  NamingAuthorityImpl na = new NamingAuthorityImpl(naConfig);  // Start NA’s HTTP server (optional)  na.startHttpServer(); |

### Deploying the Naming Authority as a Web Application

1. Configure the naming authority by editing default settings in *<project\_home>/WebContent/WEB-INF/web.xml*
2. *$ cd <project\_home>*
3. *$ ant deployTomcat*

## Framework-Resolver

This project provides classes related to resolving identifiers and retrieving data objects.

### 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*).

|  |
| --- |
| IdentifierValues resolveGrid( String identifier );  IdentifierValues resolveHttp( String identifier ); |

*resolveGrid* can be used when the naming authority is being run by the grid service. In this case, the grid client generated by the naming authority grid service is used to make the *getValues()* call exposed by that service.

There are prior two steps that *resolveGrid* has to complete in order to determine the grid service URL:

1. **Determine the naming authority URL**. The identifier URL may not be necessarily point to the naming authority; it could be pointing to a redirect service, such as the PURL server described earlier. In this case, the naming authority location is determined by “following” the identifier (via HTTP GET) and capturing the “*Location*” that gets returned along with the 302 HTTP redirect code. If no redirect is received, then the identifier URL is assumed to be pointing to the naming authority server directly. *ResolverUtil* has a utility method called *getNamingAuthorityURL* that implements the functionality described here.
2. **Retrieve configuration object from naming authority**. Once the NA’s URL is known, the NA’s configuration object can be retrieved by adding *“?config”* to the URL and “following” it (via HTTP GET). The configuration object contains the desired grid service URL (given there is one). ResolverUtil has a utility method called *getNamingAuthorityConfig* that implements the functionality described here.

### 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 indentifier in order to execute the *retrieve* operation successfully.

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

#### 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.

|  |
| --- |
| public interface RetrieverFactory {  Retriever getRetriever( IdentifierValues ivs );  Retriever getRetriever( String name );  } |

### 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.

|  |
| --- |
| public class **DefaultRetrieverFactory** implements **RetrieverFactory** {  private Map<String, Retriever> retrievers;  public DefaultRetrieverFactory(Map<String, Retriever> retrievers);  public Retriever **getRetriever**( IdentifierValues ivs ) throws Exception {  throw new Exception("Not implemented yet");  }  public 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.

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

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

### Using Framework-Resolver 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.

### Framework-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 data types are references to the data type beans defined by the naming authority (*framework-namingauthority-context.xml*).

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>  <bean id="**CQLType.name**" class="org.springframework.beans.factory.config.PropertyPathFactoryBean"/>  <bean id="**EPRType.name**" class="org.springframework.beans.factory.config.PropertyPathFactoryBean"/>  </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> |

## Framework-NamingAuthority-GridSvc

The framework implements a standard analytical grid service that runs the naming authority implementation described above. This is an alternative to running the NA as a standalone web application. This also provides a “write” interface to create identifiers. The web application deployment is read-only currently.

### Deployment

1. Configure the naming authority by editing *service.properties*

|  |
| --- |
| #service deployment properties  #Tue Jul 14 11:15:30 EDT 2009  **identifiersNaDbUrl**=jdbc\:mysql\://localhost/identifiers  **identifiersNaHttpServerPort**=8081  **identifiersNaDbUser**=root  **identifiersNaGridSvcUrl**=http\://localhost\:8080/wsrf/services/cagrid/IdentifiersNAService  **identifiersNaDbPassword**=cagrid  **identifiersNaPrefix**=http\://purl.cagrid.org\:8090/cagrid |

1. cd <project\_home>
2. 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">  <simpleType name="Type">  <restriction base="string">  <enumeration value="URL"/>  <enumeration value="DOI"/>  <enumeration value="EPR"/>  <enumeration value="CQL"/>  </restriction>  </simpleType>  <element name="Type" type="tns:Type"/>  <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 ref="tns:Type" />  <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); |