caBIG_logo_w_tm

**caGRID**

**Identifier Framework**

***Design***

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

# 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 single data-object. Furthermore, identifier can be (globally) resolved to 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”.

### The Data Owner

This is the system or domain where the target data objects reside. These are the objects being identified. *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.

### 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.

### The Naming Authority

The *Naming Authority (NA)* is the entity that issues and manages identifiers and their metadata.

### The Resolution Process

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

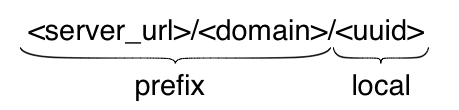
### The Data Retrieval Process

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.

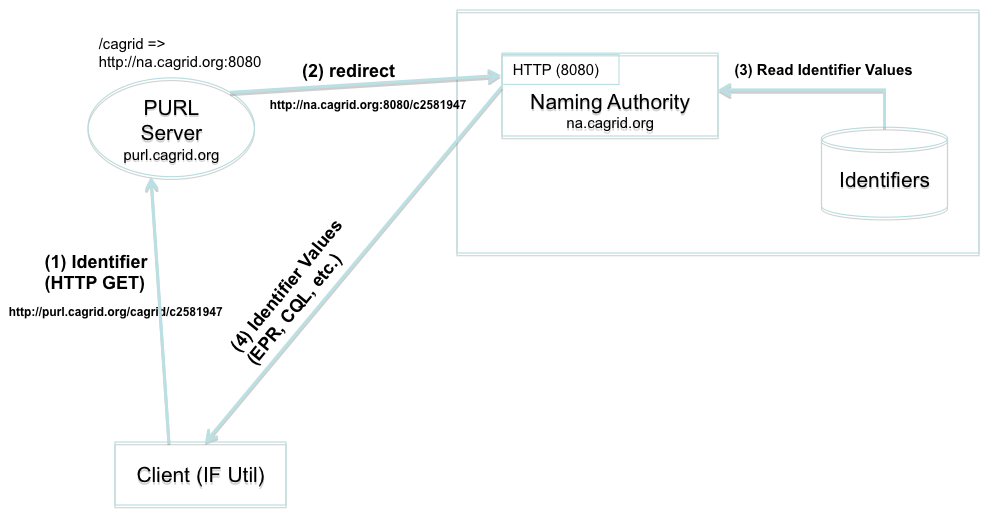


Figure HTTP Resolution

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 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.

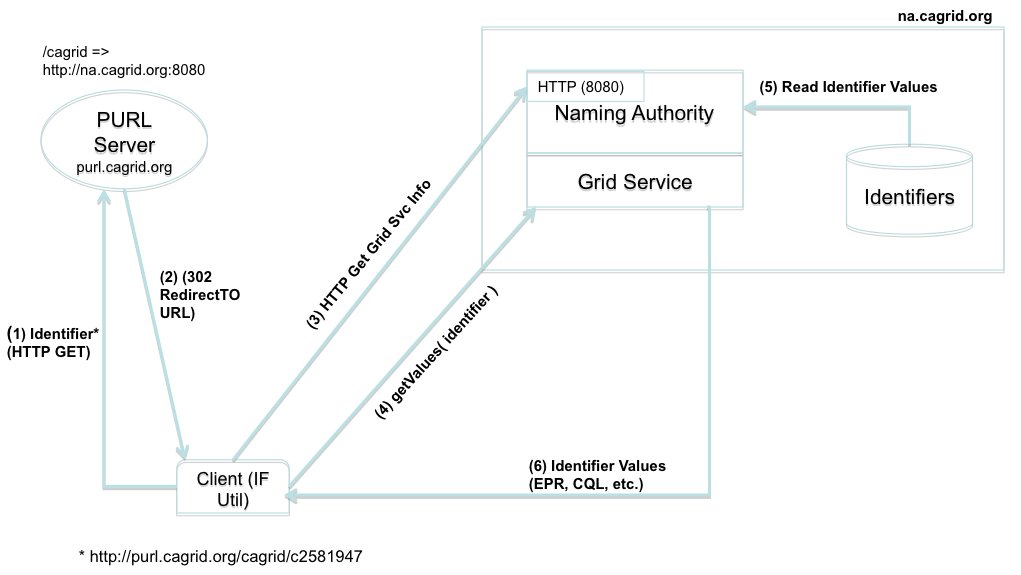


Figure Grid Resolution

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.

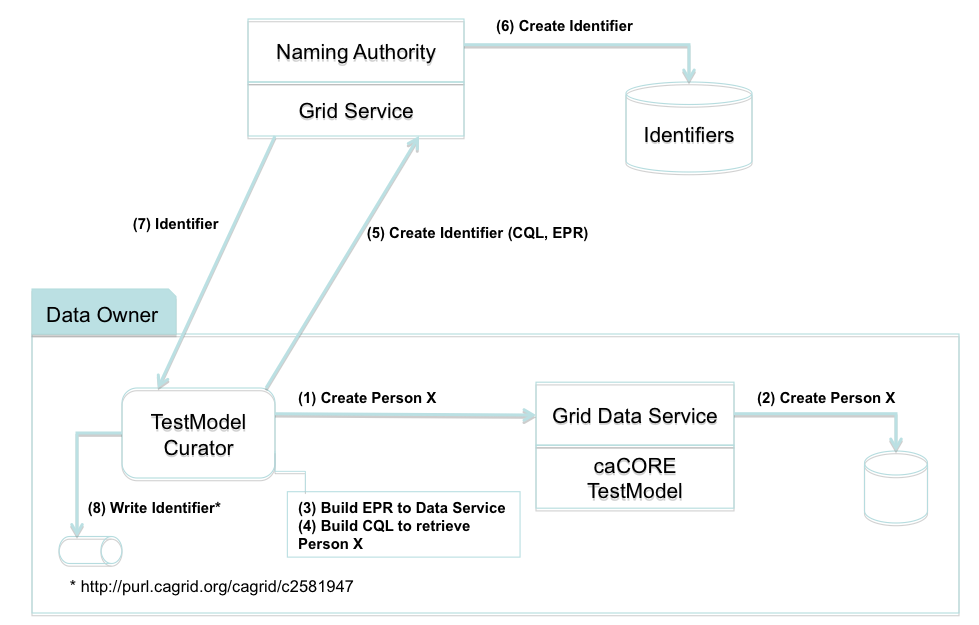


Figure Use Case: Creating Identifier

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.

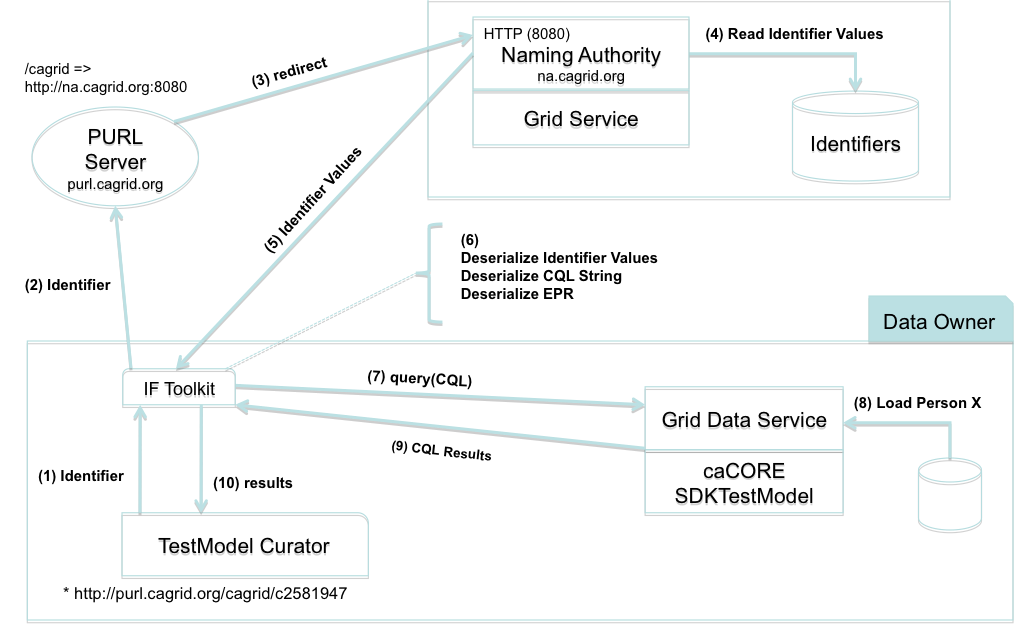


Figure Use Case: Data Retrieval

# 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.

### org.cagrid.identifiers.core

TBD

TBD

TBD

## Framework-NamingAuthority

This is the naming authority source code and runtime components.

### 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.

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.

### org.cagrid.identifiers.namingauthority

TBD

TBD

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

|  |
| --- |
| <!-- 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> |

Table framework-namingauthority-context.xml

TBD

## Framework-Resolver

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

|  |
| --- |
| // 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 *ResolverUtil* class provides utility methods for doing HTTP-GET resolution (*resolveHttp*), or Grid-based resolution (*resolveGrid*).

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.

TBD

|  |
| --- |
| <!-- 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> |

TBD

TBD

## Framework-NamingAuthority-GridSvc

TBD