caBIG_logo_w_tm

***Design v. 0.4***

**caGRID**

**Identifier Framework**

|  |  |
| --- | --- |
| ***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** |
| 0.2 | 2009-10-13 | Calixto Melean | Added Chapter 4 Extending the Framework |
| 0.3 | 2009-10-14 | Calixto Melean | Added Chapter 5 Resolution over SSL |
| 0.4 | 2009-12-10 | Calixto Melean | Major updates to chapters 3 & 4 due to code reviews and refactors |

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

The Data Owner 6

The Naming Authority 6

The Identifier Curator 7

The User 7

The Prefix Authority 7

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

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

Partial-redirect PURL 10

PURL-based Identifiers 11

The Resolution Process 12

The Data Retrieval Process 15

Use Case 15

Chapter 3 Toolkit 17

Identifiers-NamingAuthority 17

HTTP Protocol 18

Resolution Response 18

HTML Response 18

XML Response 19

Exposing Naming Authority Configuration 19

Naming Authority Configuration Schema 20

Default Naming Authority Implementation 20

Identifiers-Client 21

The Resolver Class 21

The Retriever Interface 22

The RetrieverFactory Interface 22

The RetrieverService Class 23

Using Identifiers-Client to Resolve and Retrieve a Data Object 23

Identifiers-NamingAuthority-GridSvc 23

Deployment 24

API 24

Chapter 4 Extending the Framework 24

Other Naming Authority Implementations 24

Use Case 25

Naming Authority Implementation 25

applicationContext-na.xml 26

Extending the Client Toolkit by Adding Retrieval Profiles 26

Chapter 5 Resolution over SSL 27

Securing the Naming Authority 27

Securing the Prefix Authority 29

Client Configuration 29

Performance Considerations 30

Chapter 6 Requirements to Design Mapping 30

# 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 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 (including semantic web technologies) 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

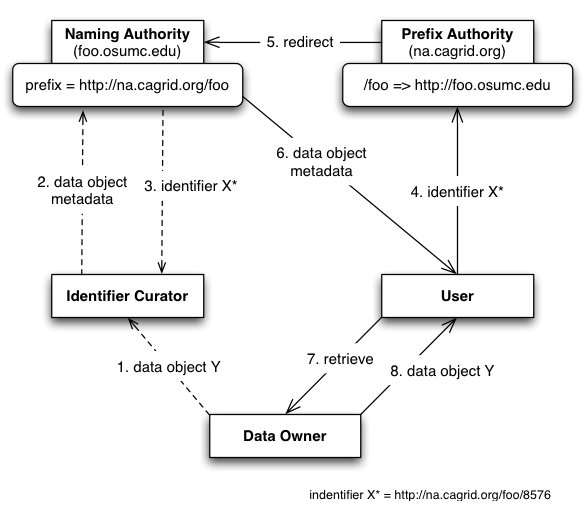


Figure 1 Conceptual Model of Identifier Framework

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. 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. It is the authority for identifiers created within its prefix.

#### The Identifier Curator

This *Identifier Curator* 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 *Data Owner*. This information is sent to the *Naming Authority* represented as metadata. In this way, the *Identifier Curator* is responsible for creating the binding between data object and identifier, using a *Naming Authority* to store the binding.

#### 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 an identifier 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 identifier 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 *Identifier Curator* wants to globally identify a new data object Y contained within the *Data Owner*. The curator builds the metadata required to help retrieve data object Y later, via the mechanisms supported by the *Data Owner*, and gives it to the *Naming Authority* as part of the “create identifier” request. The *Naming Authority* generates an identifier, stores the metadata associated with it, and returns the identifier to the *Identifier 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” (via http) to retrieve the associated metadata. The *Prefix Authority* notices the URL specifies the *foo* domain, and redirects the *U*ser to the correct *Naming Authority*. The *Naming Authority* responds to the request with the identifier’s metadata. At this point, the metadata, can be used to retrieve the data object from the *Data Owner*.

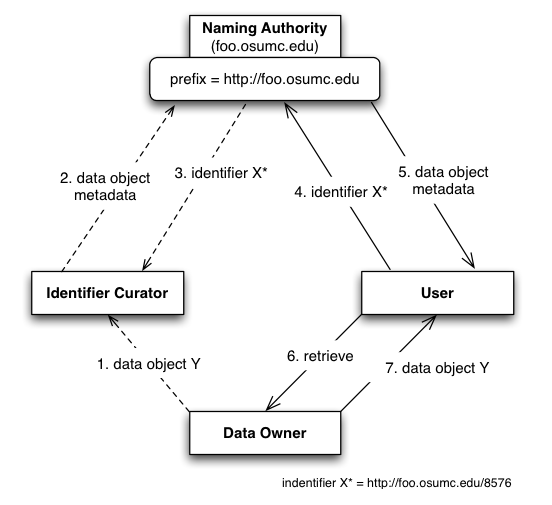


Figure 2 Conceptual Model of Identifier Framework (No Prefix Authority)

### The Resolution Process

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

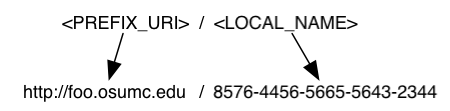
### The Data Retrieval Process

The framework defines *Data Retrieval* as the process of retrieving the data 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) or something as simple as a sequence number local to the naming authority.

## The Naming Authority (NA)

The NA maintains a database table of identifiers and their metadata. The conceptual data model can be described as a hash of arrays, where each hash entry key is a “data type”, and the entry value is a list of data values of that particular data type. For example:

“URL” => [“http://osumc.edu/flu.html”, “http://osu.edu/illness.html#flu”]

The example above defines a *URL* data type that includes two possible values (URLs) where the target resource can be retrieved from.

Multiple data types, and multiple values within a data type, could be associated with an identifier. The table below shows a flat view of this model.

|  |  |  |
| --- | --- | --- |
| **Identifier** | **Data Type** | **Value** |
| ABC | EPR | <ns1:EndpointRerefence…> |
| ABC | CQL | <CQLQuery…> |
| ABC | URL | http://osumc.edu/flu.html |

The table entries represent the metadata or identifier values associated with local identifier *ABC*[[1]](#footnote-2). 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. That is, it is expected many NAs will be deployed (even co-located with data resources), which can’t provide the permanence required by framework. 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*.” For more information, see <http://purl.org/docs/long_intro.html>

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, 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:

*/foo* => *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.

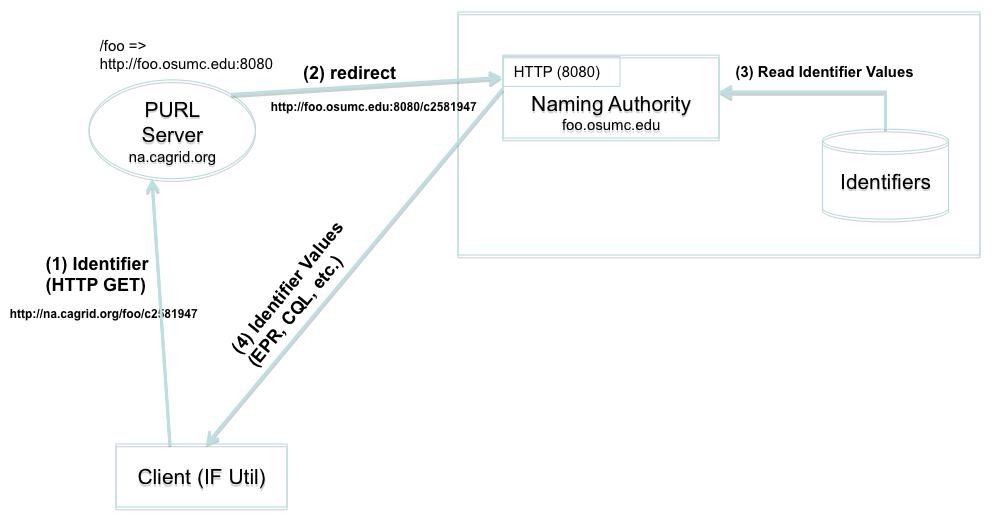


Figure 3 HTTP Resolution

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.

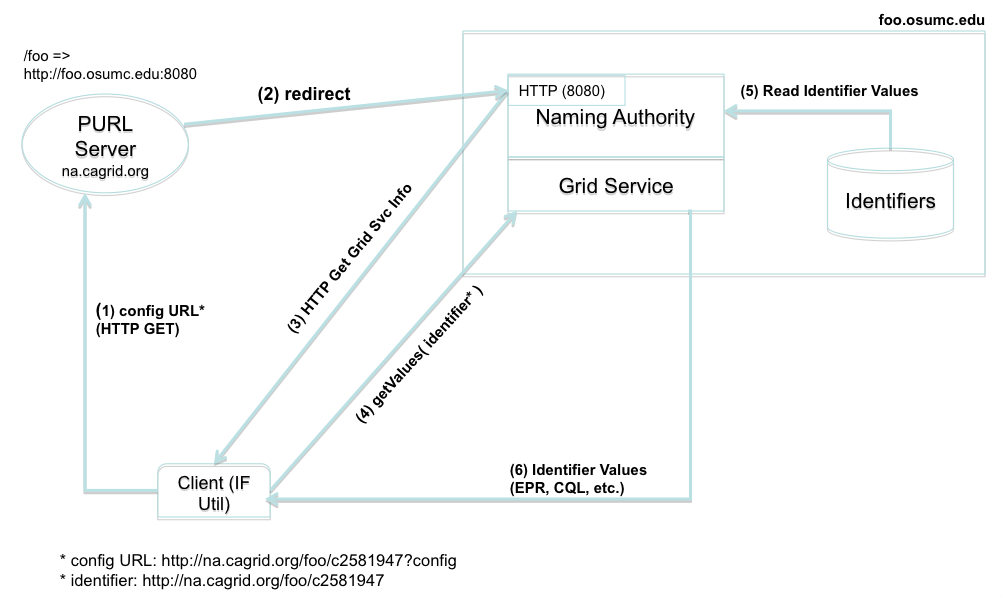


Figure 5 Grid Resolution

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.

In this way, the web (HTTP) is used resolve the identifier to its naming authority, and after the naming authority provides the information about its grid service, the grid is used to actually retrieve the identifier’s metadata. One motivating reason for using the grid service is if the identifier values are private information, grid security can be used to authorize the client prior to returning the data (the web resolution would always result in an access error in this case, such as HTTP 403 error code).

## 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 XML string that represents a ws-addressing End Point Reference. This includes the service address and port type of a deployed grid data server.
* ***CQL***: An XML string that represents 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 (also acting as identifier curator) 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.

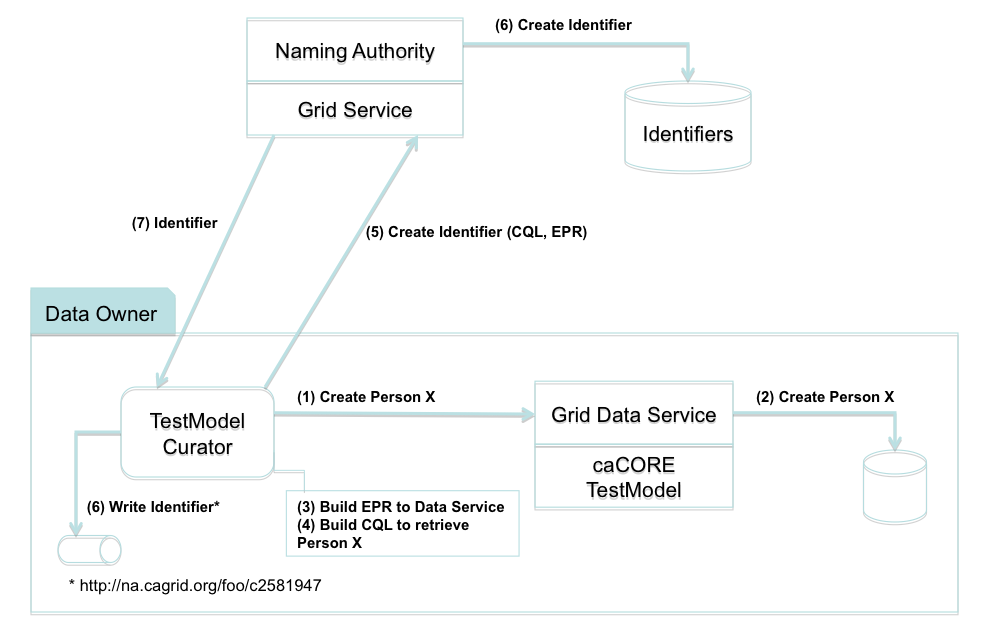


Figure 6 Use Case: Creating Identifier

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.

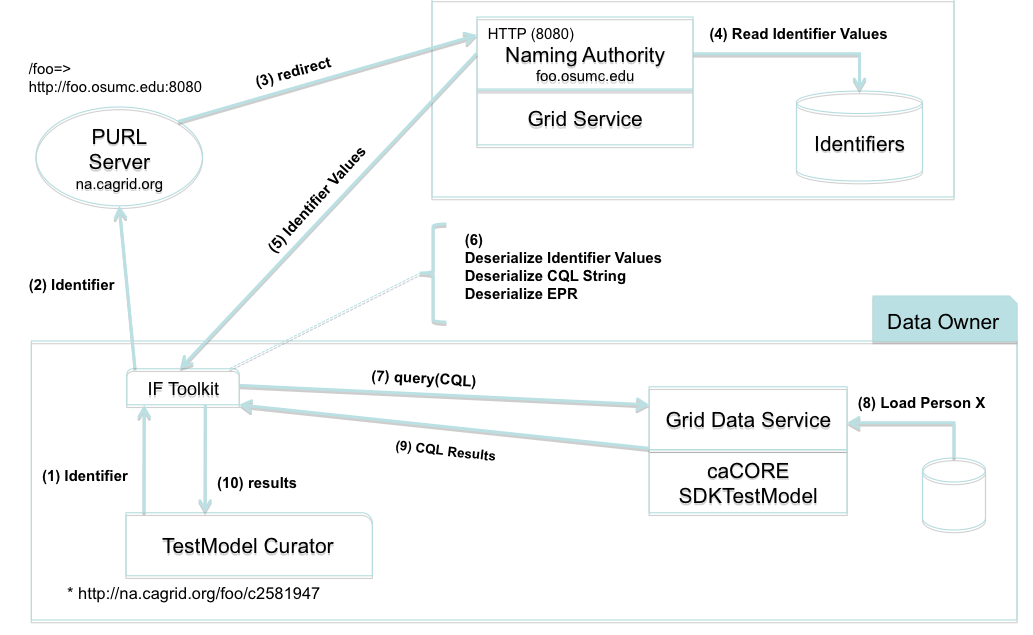


Figure 7 Use Case: Data Retrieval

# Toolkit

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

## Identifiers-NamingAuthority

This project establishes the interface and protocols that must be followed by all naming authority deployments/implementations. It also provides a default java implementation.

At minimum, a naming authority is capable of *resolving* identifiers. This is, given an identifier (URI), return the metadata (*IdentifierValues*) associated with it.

|  |
| --- |
| public interface **NamingAuthority** {  …  *IdentifierValues* ***resolveIdentifier***(URI identifier);  …  } |

A naming authority could also provide identifier creation and maintenance capabilities.

|  |
| --- |
| public interfaces **MaintainerNamingAuthority** extends **NamingAuthority** {  …  *URI* ***createIdentifier***(IdentifierValues values);  …  } |

The *createIdentifier* interface is expected to accept identifier values (metadata), generate a new identifier URI and return it to the client.

### HTTP Protocol

A naming authority responds to identifier resolution requests via HTTP GET. Identifiers are functional HTTP URIs that lead a client to the naming authority that owns it.

For example, assuming a naming authority with a prefix URI of *http://foo.osumc.edu:8080/*; a local identifier *ABC* can be resolved by navigating to *http://foo.osumc.edu:8080/ABC*

#### Resolution Response

A naming authority responds to a HTTP resolution request (such as GET *http://foo.osumc.edu:8080/ABC)* by returning the set of identifier values associated with the identifier URI.

The format of the response is either HTML or XML. A client specifies the type of response desired by setting the HTTP ACCEPT request header appropriately.

The ACCEPT header typically contains a list of response formats that are acceptable by the client.

For a naming authority, HTML is the dominant format. That is, in order to request XML, the ACCEPT header must contain *application/xml*, and must not include *text/html*. 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.

##### HTML Response

This response is intended to be consumed by humans, such as those using a web browser for resolution purposes. Therefore, this specification does not require any particular styling or presentation layout. The response should simply present identifier values in a valid HTML document.

Figure 4 shows an example web browser (HTML) view of a resolved identifier.

##### XML Response

The XML response is a serialized view of the *IdentifierValues* object. It’s a XML document that conforms to a well-known XML schema. As example:

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <na:IdentifierValues  xmlns:na="http://na.cagrid.org/1.0/NamingAuthority"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://na.cagrid.org/1.0/NamingAuthority http://foo.osumc.edu:8080/namingauthority/org.cagrid.identifiers.namingauthority.xsd">  <na:KeyValues>  <na:key>EPR</na:key>  <na:value>end point reference 1</na:value>  <na:value>end point reference 2</na:value>  </na:KeyValues>  <na:KeyValues>  <na:key>URL</na:key>  <na:value>http://www.google.com</na:value>  </na:KeyValues>  </na:IdentifierValues> |

###### IdentifierValues XML Schema

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <xs:schema  targetNamespace="http://na.cagrid.org/1.0/NamingAuthority"  xmlns="http://na.cagrid.org/1.0/NamingAuthority"  xmlns:xs="http://www.w3.org/2001/XMLSchema"  elementFormDefault="qualified" attributeFormDefault="unqualified">  <xs:element name="IdentifierValues" type="IdentifierValues" />  <xs:complexType name="IdentifierValues">  <xs:sequence>  <xs:element ref="KeyValues" minOccurs="0" maxOccurs="unbounded"/>  </xs:sequence>  </xs:complexType>    <xs:element name="KeyValues" type="KeyValues"/>  <xs:complexType name="KeyValues">  <xs:sequence>  <xs:element name="key" type="xs:string"/>  <xs:element name="value" type="xs:string" minOccurs="1" maxOccurs="unbounded"/>  </xs:sequence>  </xs:complexType>  </xs:schema> |

#### Exposing Naming Authority Configuration

The naming authority must also make available its configuration object via HTTP. A client can retrieve this object’s data by adding a *?config* parameter to the prefix URI or any identifier URI.. As example:

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

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

When the *?config* parameter is present, the naming authority ignores the rest of the data in the URI (if any) and returns an XML document such as:

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <na:Configuration  xmlns:na="http://na.cagrid.org/1.0/NamingAuthority"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://na.cagrid.org/1.0/NamingAuthority http://foo.osumc.edu:8080/namingauthority/org.cagrid.identifiers.namingauthority.xsd">  <na:gridSvcUrl>  http://foo.osumc.edu:8081/wsrf/services/cagrid/IdentifiersNAService  </na:gridSvcUrl>  </na:Configuration> |

##### Naming Authority Configuration Schema

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <xs:schema  targetNamespace="http://na.cagrid.org/1.0/NamingAuthority"  xmlns="http://na.cagrid.org/1.0/NamingAuthority"  xmlns:xs="http://www.w3.org/2001/XMLSchema"  elementFormDefault="qualified" attributeFormDefault="unqualified">    <xs:element name="Configuration" type="Configuration"/>  <xs:complexType name="Configuration">  <xs:sequence>  <xs:element name="gridSvcUrl" type="xs:string" minOccurs="1" maxOccurs="1"/>  </xs:sequence>  </xs:complexType>  </xs:schema> |

å

**HTTP Status Codes**

In general, the Naming and Prefix Authority follow the standard semantics of the HTTP specification for status codes (see <http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>). The PURL documentation, notes some specific interpretations where there were potential options (such as how to respond to a “tombstoned” PURL: (see <http://purl.org/docs/help.html#table1>). Of particular note to the Naming Authority are the following codes:

|  |  |
| --- | --- |
| **HTTP Status Code** | **Meaning** |
| 200 | **OK** (standard meaning); used for all successful resolution requests |
| 403 | **Forbidden** (standard meaning); used if the user is not authorized to view the metadata, or hasn’t provided authentication and authorization is required on the requested identifier |
| 404 | **Not Found** (as in PURL specification); used if the NA knows nothing of the requested identifier |
| 410 | **Gone** (as in PURL specification); used if the NA deleted the requested identifier |
| 500 | Server Error (standard meaning); return if there is a configuration or implementation issue with the NA |

**HTTP Protocol Examples**

TODO: Show some sample GETs (with headers), and the responses Use format something like this <http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol#Example_session> .

### Default Naming Authority Implementation

The framework’s default implementation has the following characteristics:

* Implements the *MaintainerNamingAuthority* interface. Therefore, it can create and resolve identifiers.
* Persists identifiers and *IdentifierValues* in a *MySQL* database using hibernate.
* Runs as web application in a Tomcat container.

Typical configuration parameters can be adjusted by editing the properties file *[PROJECT\_HOME]/WebContent/WEB-INF/na.properties*

|  |
| --- |
| **cagrid.na.prefix**=http://foo.osumc.edu:8080/  **cagrid.na.schemauri**=http://foo.osumc.edu:8080/namingauthority/org.cagrid.identifiers.namingauthority.xsd  **cagrid.na.grid.url**=http://foo.osumc.edu:8081/wsrf/services/cagrid/IdentifiersNAService  **cagrid.na.db.dialect**=org.hibernate.dialect.MySQL5InnoDBDialect  **cagrid.na.db.driver**=com.mysql.jdbc.Driver  **cagrid.na.db.name**=nadb  **cagrid.na.db.url**=jdbc:mysql://localhost:3306/${cagrid.na.db.name}  **cagrid.na.db.username**=root  **cagrid.na.db.password**= |

The naming authority is bootstrapped via *springframework* configuration: *[PROJECT\_HOME]/WebContent/WEB-INF/applicationContext-na.xml*

To deploy the naming authority to Tomcat:

1. Set CATALINA\_HOME environment appropriately
2. $ cd [PROJECT\_HOME]
3. $ ant deployTomcat

## Identifiers-Client

This project provides a toolkit for clients to easily interface with a naming authority for resolution services and with data owners for object retrieval. A client would typically use the *Resolver* class to obtain the identifier metadata (*IdentifierValues*) and then use a built-in or custom *Retriever* object to retrieve the data object from the owner’s space.

### The Resolver Class

This class provides two methods for identifier resolution:

|  |
| --- |
| IdentifierValues *resolveGrid*( URI identifier )  IdentifierValues *resolveHttp*( URI identifier ) |

***resolveGrid*** resolves the identifier using the naming authority’s grid service (if available). The grid service end point URL is discovered by querying the configuration object from the naming authority.

***resolveHTTP*** resolves the identifier by using the naming authority’s HTTP services as previously described. The identifier is simply followed using HTTP GET. *IdentifierValues* are received from the naming authority as an XML document which is then automatically de-serialized into a java object (*IdentifierValues*).

|  |
| --- |
| URI identifier = new URI("http://foo.osumc.edu:8080/ABC");  IdentifierValues ivs = new Resolver().resolveHttp(identifier);  System.out.println(ivs.toString()); |

By default, *Resolver* loads spring configuration for *identifier-client-context.xml*. A different resource can be provided by using the *Resolver(String[])* constructor. Currently, the only configuration required from this resource is the castor XML context necessary to de-serialize responses from the naming authority.

### The Retriever Interface

This interface declares a single operation, *retrieve*; whose purpose is to retrieve a data object from the owner’s space.

|  |
| --- |
| public interface **Retriever** {  public Object *retrieve*(IdentifierValues ivs) throws Exception;  } |

The framework has a built-in retriever (*CQLRetriever*) that allows a client to query a grid data service.

|  |
| --- |
| public class **CQLRetriever** extends **RetrieverImpl** {  public Object *retrieve*( IdentifierValues ivs ){  CQLQueryResults results = new CQLQueryResults();  …  …  …  return results;  } |

### The RetrieverFactory Interface

A *RetrieverFactory* “owns” a collection of *Retriever* objects uniquely identified by their name. A retriever instance can be calling the *getRetriever* method, which takes either a name, that uniquely identifies the retriever class; or the identifier’s metadata.

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

The framework provides a default factory, *DefaultRetrieverFactory*. It maintains a map of *RetrieverImpl* objects keyed by retriever name.

The *getRetriever* by *IdentifierValues* chooses the retriever instance whose ALL required keys (*RetrieverImpl.getRequiredKeys()*) exist in the identifier’s metadata (*IdentifierValues*). If multiple retrievers meet this criteria, the one with the largest number of keys is chosen. If multiple retrievers have the same number of keys, an exception is thrown.

Identifier adopters are free to provide different factory implementations with different retriever selection criteria. All factories must implement the *RetrieverFactory* interface.

### The RetrieverService Class

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

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

|  |
| --- |
| // Resolution  IdentifierValues ivs = new Resolver().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 = new Resolver().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 file *identifiers-client-context.xml*. Other spring files can be used by using the specialized *RetrieverService* constructor.

## 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 “update” 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-gridsvc/etc/na.properties*

1. cd [PROJECT\_HOME]/projects/identifiers-namingauthority-gridsvc
2. ant deployTomcat

### API

The grid service currently supports two operations:

|  |
| --- |
| URI createIdentifier(IdentifierValues);  IdentifierValues getTypeValues(URI identifier); |

# Extending the Framework

## Other Naming Authority Implementations

The default naming authority implementation provided by the framework (*org.cagrid.identifiers.namingauthority.impl.NamingAuthority*) may not exactly match all use cases and deployment scenarios required by identifiers adopters. Therefore, the framework could be configured to use a different naming authority implementation.

This springframework resource *applicationContext-na.xml*[[2]](#footnote-3) sets up the desired naming authority implementation. The default configuration is shown below.

|  |
| --- |
| <bean id="**NamingAuthority**"  class="org.cagrid.identifiers.namingauthority.impl.**NamingAuthorityImpl**">  <property name="configuration">  <bean  class="org.cagrid.identifiers.namingauthority.impl.**NamingAuthorityConfigImpl**">  <property name="prefix" value="${cagrid.na.prefix}" />  <property name="gridSvcUrl" value="${cagrid.na.grid.url}" />  </bean>  </property>  <property name="identifierGenerator">  <bean  class="org.cagrid.identifiers.namingauthority.impl.IdentifierGeneratorImpl" />  </property>  <property name="identifierDao" ref="identifierDao" />  </bean> |

Providing a different naming authority implementation involves the following general steps:

1. Implement either the *NamingAuthority* or the *MaintainerNamingAuthority* interface.
2. Configure *applicationContext-na.xml* to point to the newly created beans and initialize them accordingly.

### Use Case

An organization wishes to deploy an identifiers framework instance with the following characteristics:

* Naming authority does not generate identifiers
* Naming authority “forwards” identifier resolution requests to a remote system
* Remote system uses the identifier to lookup its associated metadata and returns it to the naming authority
* Naming authority stores neither identifiers nor metadata

#### Naming Authority Implementation

|  |
| --- |
| import org.cagrid.identifiers.namingauthority.NamingAuthority;  import org.cagrid.identifiers.namingauthority.IdentifierValues;  public class **CustomNamingAuthority** implements **NamingAuthority** {  private String remoteSystemURL;  String *getRemoteSystemURL*(){…};  void *setRemoteSystemURL*( String url ) {…};    public IdentifierValues ***resolveIdentifier***(URI identifier) {    IdentifierValuesImpl ivs = new IdentifierValuesImpl();  String identifierStr = (String)identifier;  //  // Insert code here to obtain metadata from remoteSystemURL  //  // Populate ivs with relevant metadata, for example:  // ivs.add(“URL”, “http://lexevs.nci.org/C009822”);  // ivs.add(“CODE”, “C009822”);  //  return ivs;  }  } |

#### applicationContext-na.xml

|  |
| --- |
| <bean id="**NamingAuthority**" class="**CustomNamingAuthorityImpl**">  <property name="configuration">  <bean  class="org.cagrid.identifiers.namingauthority.impl.**NamingAuthorityConfigImpl**">  <property name="prefix" value="${cagrid.na.prefix}" />  <property name="gridSvcUrl" value="${cagrid.na.grid.url}" />  </bean>  </property>  </bean> |

## Extending the Client Toolkit by Adding Retrieval Profiles

The framework’s client toolkit (*identifiers-client* project) includes a retrieval profile (*CQLRetriever*) that shows how identifier metadata would be used to retrieve the referenced data object from a caGrid data service using CQL.

In order to enable processing of different types of metadata and enable data retrieval from the corresponding data sources, new profiles can be defined. This involves the following general steps:

1. Implement the new data retrieval class by implementing the *Retriever* interface or extending (sub-classing) the *RetrieverImpl* abstract class.
2. Add the new implementation to *identifiers-client-context.xml*[[3]](#footnote-4)*.* For example:

|  |
| --- |
| <bean id="**LexEVSRetriever** " class="**org.nci.LexEVSRetriever**">  <property name="requiredKeys">  <util:list>  <value>**CODE**</value>  <value>**URL**</value>  </util:list>  </property>  </bean>  <bean id="RetrieverFactory"  class="org.cagrid.identifiers.retriever.impl.DefaultRetrieverFactory">  <constructor-arg>  <util:map>  <entry key="**LexEVSRetriever**">  <ref local=" **LexEVSRetriever** "/>  </entry>    <entry key="CQLRetriever">  <ref local="CQLRetriever"/>  </entry>  </util:map>  </constructor-arg>  </bean> |

Notice that a list of required metadata keys can be configured by using the *requiredKeys* property. Even though the toolkit does not enforce this, the implementation class (e.g. *LexEVSRetriever*) can make use of it by calling the parent method *validateTypes*() from the implemented *retrieve*() method[[4]](#footnote-5). *ValidateTypes*() throws an exception if the metadata does not have at least one value for all of the required keys.

# Resolution over SSL

The identifiers framework supports resolution over an encrypted channel. That is, resolution of identifiers that use https (e.g. *https://namingauthority.cagrid.org/8586-3434-3444*). The client toolkit (*identifiers-client*) uses [Apache HttpClient](http://hc.apache.org/httpcomponents-client/index.html)[[5]](#footnote-6), which fully leverages the [Java Secure Socket Extension (JSSE)](http://java.sun.com/javase/technologies/security/)[[6]](#footnote-7). Hence, the only requirement to enable SSL is to configure JSSE properly.

## Securing the Naming Authority

This involves securing the corresponding application container (i.e. Tomcat). This document will not cover how to install certificates and configure Tomcat to use SSL. This is information is readily available from Apache [documentation](http://tomcat.apache.org/tomcat-5.5-doc/ssl-howto.html)[[7]](#footnote-8) and the web. The [caGrid installer](http://gforge.nci.nih.gov/frs/download.php/6860/caGrid-installer-1.3.0.1.zip)[[8]](#footnote-9) is also capable of installing and [configuring](https://cagrid.org/display/caGrid13/Install+caGrid+and+Configure+a+Secure+Container+Using+the+caGrid+1.3+Installer)[[9]](#footnote-10) a secure container by requesting host certificates from [Dorian](https://cagrid.org/display/dorian/Home)[[10]](#footnote-11).

Optionally[[11]](#footnote-12), in order to force the container to use SSL, uncomment the following block in *<PROJECT\_HOME>/WebContent/WEB-INF/web.xml*, and re-deploy to Tomcat.

|  |
| --- |
| <!-- Uncomment this to force the container to SSL  <security-constraint>  <web-resource-collection>  <web-resource-name>HTTPS Only Naming Authority</web-resource-name>  <url-pattern>/NamingAuthorityService/\*</url-pattern>  <http-method>GET</http-method>  <http-method>POST</http-method>  </web-resource-collection>  <user-data-constraint>  <transport-guarantee>CONFIDENTIAL</transport-guarantee>  </user-data-constraint>  </security-constraint>  --> |

It is also worth mentioning in this section that it is possible to secure the naming authority with SSL and still be able to resolve identifiers that start with “http://”. This requires un-commenting the section above to require SSL, as well as configuring Tomcat (*server.xml*) to redirect the non-SSL port the SSL port.

The Non-SSL port (redirects to SSL):

|  |
| --- |
| <Connector port="8080" redirectPort="8443"/> |

The globus SSL connector (Note we need to add *secure=”true”*):

|  |
| --- |
| <Connector **secure="true"** acceptCount="10" autoFlush="true" cert="host1-cert.pem"  className="org.globus.tomcat.coyote.net.HTTPSConnector" debug="0"  disableUploadTimeout="true" enableLookups="true" key="host1-key.pem"  maxSpareThreads="75" maxThreads="150" minSpareThreads="25" port="8443"  protocolHandlerClassName="org.apache.coyote.http11.Http11Protocol"  socketFactory="org.globus.tomcat.catalina.net.BaseHTTPSServerSocketFactory"  scheme="https"/> |

Or a simpler (non-globus) SSL connector:

|  |
| --- |
| <Connector port="8443" scheme="https" secure="true"  keystoreFile=".keystore" keystorePass="changeit"/> |

## Securing the Prefix Authority

The following procedure can be used to enable SSL in a local deployment of [PURLZ](http://www.purlz.org/)[[12]](#footnote-13).

1. Create a java *keystore* if none exists:

|  |
| --- |
| $ keytool –keystore /home/purlz/keystore -genkey -storepass cagrid -keyalg DSA -alias jetty –dname "CN=cagrid.org, OU=Software Research Institute, O=Biomedical Informatics, L=Columbus, ST=Ohio, C=US" -validity 999 |

1. Obtain an officially signed certificate from a known certificate authority and add it to the *keystore* created above. **Note** certificate alias must be “*jetty*”. Alternatively, use *keytool* to create a create a self-signed certificate:

|  |
| --- |
| $ keytool –keystore /home/purlz/keystore -selfcert -storepass cagrid -alias jetty -validity 999 |

1. Export certificate to a file for later use:

|  |
| --- |
| $ keytool –exportcert -storepass cagrid -alias jetty -rfc –file jetty.cer |

1. Edit *$PURLZ\_INSTALL\_DIR/modules/mod-fulcrum-frontend/etc/TransportJettyConfig.xml* as follows:
   1. Remove the *<DISABLE></DISABLE>* tags from around the Jetty SSL HTTP Server section.
   2. Set the *Port*, *Keystore*, *Password*, and *KeyPassword* accordingly (example below). Set *NeedClientAuth* to false.

|  |
| --- |
| <Set name="Port">8443</Set>  <Set name="Keystore">/home/purlz/keystore</Set>  <Set name="Password">changeit</Set>  <Set name="KeyPassword">changeit</Set>  <Set name="NeedClientAuth">false</Set> |

## Client Configuration

The certificates from the prefix authority and naming authority must be added to the *keystore* used by the JVM running the client. For example, the following command imports the PURLZ certificate that was exported earlier for later use:

|  |
| --- |
| $ keytool –keystore /home/client/keystore –import –alias jetty –file jetty.cer |

Similarly, import the naming authority certificate.

The JVM that runs the client program using the resolution toolkit (identifiers-client) must be passed they *keystore* being used (e.g. –Djavax.net.ssl.trustStore=/home/client/keystore)

## Performance Considerations

Encrypting communications adds overhead to the resolution processes, as expected. The following test scenario has been run for comparison purposes:

* PURLZ 1.6 running in a dedicated machine.
* Naming Authority running in a dedicated machine.
* Client running in a dedicated machine.

|  |
| --- |
| Scenario 1:   * One client * Resolution of 35,000 identifiers   Using HTTP: 62 minutes  Using HTTPS (PA & NA): 98 minutes |

|  |
| --- |
| Scenario 2:   * Five clients concurrently * ~7,000 identifiers/client (total 35,000 identifiers)   Using HTTP: 48 minutes  Using HTTPS (Both PA & NA): 56 minutes |

# Requirements to Design Mapping

|  |  |
| --- | --- |
| **Requirement** | **Design** |
| **CAGRID-IDS-030** | The Prefix Authority (7), The Prefix Authority (10) |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

1. References
2. *caGrid Identifiers Framework Requirements*, <https://cagrid.org/display/identifiers/Requirements>

1. The naming authority does not store the full identifier name; only the local name. [↑](#footnote-ref-2)
2. caGrid/projects/identifiers-namingauthority/WebContent/WEB-INF/applicationContext-na.xml [↑](#footnote-ref-3)
3. caGrid/projects/identifiers-client/etc/org/cagrid/identifiers/client/identifiers-client-context.xml [↑](#footnote-ref-4)
4. This assumes LevEVSRetriever extends RetrieverImpl [↑](#footnote-ref-5)
5. http://hc.apache.org/httpcomponents-client/index.html [↑](#footnote-ref-6)
6. http://java.sun.com/javase/technologies/security [↑](#footnote-ref-7)
7. http://tomcat.apache.org/tomcat-5.5-doc/ssl-howto.html [↑](#footnote-ref-8)
8. http://gforge.nci.nih.gov/frs/download.php/6860/caGrid-installer-1.3.0.1.zip [↑](#footnote-ref-9)
9. https://cagrid.org/display/caGrid13/Install+caGrid+and+Configure+a+Secure+Container+Using+the+caGrid+1.3+Installer [↑](#footnote-ref-10)
10. https://cagrid.org/display/dorian/Home [↑](#footnote-ref-11)
11. This is a “best practice”. It’s not absolutely required to deploy to a secure container. It’s simply a way to force the container to have SSL enabled. [↑](#footnote-ref-12)
12. http://www.purlz.org [↑](#footnote-ref-13)