XML Entity and URI Resolvers

Version 1.5 of this document

Norman Walsh

Staff Engineer
Sun Microsystems, XML Technology Center

20 Nov 2006

Table of Contents

Finding Resources on the Net

It's very common for web resources to be related to other resources: documents rely on DTDs and schemas, schemas are derived from other schemas, stylesheets are often customizations of other stylesheets, documents refer to the schemas and stylesheets with which the expect to be processed, etc. These relationships are expressed using URIs, most often URLs.

Relying on URLs to directly identify resources to be retrieved often causes problems for end users:

1. If they're absolute URLs, they only work when you can reach them[1]. Relying on remote resources makes XML processing susceptible to both planned and unplanned network downtime.

   The URL “http://www.oasis-open.org/docbook/xml/4.1.2/docbookx.dtd” isn't very useful if I'm on an airplane at 35,000 feet.

2. If they're relative URLs, they're only useful in the context where the were initially created.

   The URL “../../xml/dtd/docbookx.xml” isn't useful *anywhere* on my system. Neither, for that matter, is “/export/home/fred/docbook412/docbookx.xml”.

One way to avoid these problems is to use an entity resolver (a standard part of SAX) or a URI Resolver (a standard part of JAXP). A resolver can examine the URIs of the resources being requested and determine how best to satisfy those requests.

The best way to make this function in an interoperable way is to define a standard format for mapping system identifiers and URIs. The OASIS Entity Resolution Technical Committee is defining an XML representation for just such a mapping. These “catalog files” can be used to map public and system identifiers and other URIs to local files (or just other URIs).

Resolver Classes Version 1.2

The Resolver classes that are described in this article greatly simplify the task of using Catalog files to perform entity resolution. Many users will want to simply use these classes directly “out of the box” with their applications (such as Xalan and Saxon), but developers may also be interested in the JavaDoc API Documentation. The full documentation, current source code, and discussion mailing list are available from the Apache XML Commons project.

Changes from Version 1.1

See the release notes.

The most important change in this release is the availability of both source and binary forms under a generous license agreement.

Other than that, there have been a number of minor bug fixes and the introduction of system properties in addition to the `CatalogManager.properties` file to control the resolver.

What's Wrong with System Identifiers?

The problems associated with system identifiers (and URIs in general) arise in several ways:

1. I have an XML document that I want to publish on the web or include in the distribution of some piece of software. On my system, I keep the doctype of the document in some local directory, so my doctype declaration reads:

   ```
   <!DOCTYPE article PUBLIC "-//OASIS//DTD DocBook XML V4.1.2//EN"
                   "file:///n:/share/doctypes/docbook/xml/docbookx.dtd">
   ```

   As soon as I distribute this document, I immediately begin getting error reports from customers who can't read the document because they don't have DocBook installed at the location identified by the URI in my document.

2. Or I remember to change the URI before I publish the document:

   ```
   <!DOCTYPE article PUBLIC "-//OASIS//DTD DocBook XML V4.1.2//EN"
                   "http://www.oasis-open.org/docbook/xml/4.1.2/docbookx.dtd">
   ```

   And the next time I try to edit the document, *I get errors* because I happen to be working on my laptop on a plane somewhere and can't get to the net.

3. Just as often, I get tripped up this way: I'm working collaboratively with a colleague. She's created initial drafts of some documents that I'm supposed to review and edit. So I grab them and find that I can't open or publish them because I don't have the same network connections she has or I don't have my applications installed in the same place. And if I change the system identifiers so they work on my system, she has the same problems when I send them back to her.

4. These problems aren't limited to editing applications. If I write a special stylesheet for formatting our collaborative document, it will include some reference to the “main” stylesheet:

   ```
   <xsl:import href="/path/to/real/stylesheet.xsl"/>
   ```

   But this won't work on my colleague's machine because she has the main stylesheet installed somewhere else.

Public identifiers offer an effective solution to this problem, at least for documents. They provide global, unique names for entities independent of their storage location. Unfortunately, public identifiers aren't used very often because many users find that they cannot rely on applications resolving them in an interoperable manner.

For XSLT, XML Schemas, and other applications that rely on URIs without providing a mechanism for associating public identifiers with them, the situation is a little more irksome, but it can still be addressed using a URI Resolver.

Naming Resources

In some contexts, it's more useful to refer to a resource by name than by address. If I want the version 3.1 of the DocBook DTD, or the 1911 edition of Webster's dictionary, or *The Declaration of Independence*, that's what I want, irrespective of its location on the net (or even if it's available on the net). While it is possible to view a URL as an address, I don't think that's the natural interpretation.

There are currently two ways that I might reasonably assign an address-independent name to an object: public identifiers or Uniform Resource Names (URNs)[2].

Public Identifiers

Public identifiers are part of XML 1.0. They can occur in any form of external entity declaration. They allow you to give a globally unique name to any entity. For example, the XML version of DocBook V4.1.2 is identified with the following public identifier:

```
-//OASIS//DTD DocBook XML V4.1.2//EN
```

You'll see this identifier in the two doctype declarations I used earlier. This identifier gives no indication of where the resource (the DTD) may be found, but it does uniquely name the resource. That public identifier, now and forever refers to the XML version of DocBook V4.1.2.

Uniform Resource Names

URNs are a form of URI. Like public identifiers, they give a location-neutral, globally unique name to an entity. For example, OASIS might choose to identify the XML version of DocBook V4.1.2 with the following URN:

```
urn:oasis:names:specification:docbook:dtd:xml:4.1.2
```

Like a public identifier, a URN can now and forever refer to a specific entity in a location-independent manner.

The publicid URN Namespace

Public identifiers don't fit very well into the web architecture (they are not, for example, always valid URIs). This problem can be addressed by the `publicid` URN namespace defined by RFC 3151.

This namespace allows public identifiers to be easily represented as URNs. The OASIS XML Catalog specification accords special status to URNs of this form so that catalog resolution occurs in the expected way.

Resolving Names

Having extolled the virtues of location-independent names, it must be said that a name isn't very useful if you can't find the thing it refers to. In order to do that, you must have a name resolution mechanism that allows you to determine what resource is referred to by a given name.

One important feature of this mechanism is that it can allow resources to be distributed, so you don't have to go to http://www.oasis-open.org/docbook/xml/4.1.2/docbookx.dtd to get the XML version of DocBook V4.1.2, if you have a local copy.

There are a few possible resolution mechanisms:

- The application just “knows”. Sure, it sounds a little silly, but this is currently the mechanism being used for namespaces. Applications know what the semantics of namespaced elements are because they recognize the namespace URI.

- OASIS Catalog files provide a mechanism for mapping public and system identifiers, allowing resolution to both local and distributed resources. This is the resolution scheme we're going to consider for the balance of this column.

- Many other mechanisms are possible. There are already a few for URNs, including at least one built on top of DNS, but they aren't widely deployed.

Catalog Files

Catalog files are straightforward text files that describe a mapping from names to addresses. Here's a simple one:

Example 1. An Example Catalog File

```
<catalog xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog">

<public publicId="-//OASIS//DTD XML DocBook V4.1.2//EN"
        uri="docbook/xml/docbookx.dtd"/>

<system systemId="urn:x-oasis:docbook-xml-v4.1.2"
        uri="docbook/xml/docbookx.dtd"/>

<delegatePublic publicIdStartString="-//Example//"
          catalog="http://www.example.com/catalog"/>
</catalog>
```

This file maps both the public identifier and the URN I mentioned earlier to a local copy of DocBook on my system. If the doctype declaration uses the public identifier for DocBook, *I'll get DocBook* regardless of the (possibly bogus) system identifier! Likewise, my local copy of DocBook will be used if the system identifier contains the DocBook URN.

The delegate entry instructs the resolver to use the catalog “`http://www.example.com/catalog`” for any public identifier that begins with “-//Example//”. The advantage of delegate in this case is that I don't have to parse that catalog file unless I encounter a public identifier that I reasonably expect to find there.

Understanding Catalog Files

The OASIS Entity Resolution Technical Committee is actively defining the next generation XML-based catalog file format. When this work is finished, it is expected to become the official XML Catalog format. In the meantime, the existing OASIS Technical Resolution TR9401 format is the standard.

OASIS XML Catalogs

OASIS XML Catalogs are being defined by the Entity Resolution Technical Committee. This article describes the 01 Aug 2001 draft. Note that this draft is labelled to reflect that it is “not an official committee work product and may not reflect the consensus opinion of the committee.”

The document element for OASIS XML Catalogs is `catalog`. The official namespace name for OASIS XML Catalogs is “`urn:oasis:names:tc:entity:xmlns:xml:catalog`”.

There are eight elements that can occur in an XML Catalog: `group`, `public`, `system`, `uri`, `delegatePublic`, `delegateSystem`, `delegateURI`, and `nextCatalog`:

`<catalog` *`prefer`*`="`*`public|system`*`"` *`xml:base`*`="`*`uri-reference`*`">`

> The `catalog` element is the root of an XML Catalog.
>
> The `prefer` setting determines whether or not public identifiers specified in the catalog are to be used in favor of system identifiers supplied in the document. Suppose you have an entity in your document for which both a public identifier and a system identifier has been specified, and the catalog only contains a mapping for the public identifier (e.g., a matching `public` catalog entry). If the current value of `prefer` is “public”, the URI supplied in the matching `public` catalog entry will be used. If it is “system”, the system identifier in the document will be used. (If the catalog contained a matching `system` catalog entry giving a mapping for the system identifier, that mapping would have been used, the public identifier would never have been considered, and the setting of override would have been irrelevant.)
>
> Generally, the purpose of catalogs is to override the system identifiers in XML documents, so `prefer` should usually be “public” in your catalogs.
>
> The `xml:base` URI is used to resolve relative URIs in the catalog as described in the XML Base specification.

`<group` *`prefer`*`="`*`public|system`*`"` *`xml:base`*`="`*`uri-reference`*`">`

> The `group` element serves merely as a wrapper around one or more other entries for the purpose of establishing the preference and base URI settings for those entries.

`<public publicId="`*`pubid`*`" uri="`*`systemuri`*`"/>`

> Maps the public identifier *`pubid`* to the system identifier *`systemuri`*.

`<system systemId="`*`sysid`*`" uri="`*`systemuri`*`"/>`

> Maps the system identifier *`sysid`* to the alternate system identifier *`systemuri`*.

`<uri name="`*`uri`*`" uri="`*`alternateuri`*`"/>`

> The `uri` entry maps a *`uri`* to an *`alternateuri`*. This mapping, as might be performed by a JAXP URIResolver, for example, is independent of system and public identifier resolution.

`<delegatePublic publicIdStartString="`*`pubid-prefix`*`" catalog="`*`cataloguri`*`"/>`, `<delegateSystem systemIdStartString="`*`sysid-prefix`*`" catalog="`*`cataloguri`*`"/>`, `<delegateURI uriStartString="`*`uri-prefix`*`" catalog="`*`cataloguri`*`"/>`

> The delegate entries specify that identifiers beginning with the matching prefix should be resolved using the catalog specified by the *`cataloguri`*. If multiple delegate entries of the same kind match, they will each be searched, starting with the longest prefix and continuing with the next longest to the shortest.

The delegate entries differs from the `nextCatalog` entry in the following way: alternate catalogs referenced with a `nextCatalog` entry are parsed and included in the current catalog. Delegated catalogs are only considered, and consequently only loaded and parsed, if necessary. Delegated catalogs are also used *instead of* the current catalog, not as part of the current catalog.

`<rewriteSystem systemIdStartString="`*`sysid-prefix`*`" rewritePrefix="`*`new-prefix`*`"/>`, `<rewriteURI uriStartString="`*`uri-prefix`*`" rewritePrefix="`*`new-prefix`*`"/>`

Supports generalized rewriting of system identifiers and URIs. This allows all of the URI references to a particular document (which might include many different fragment identifiers) to be remapped to a different resource).

`<nextCatalog catalog="`*`cataloguri`*`"/>`

Adds the catalog file specified by the *`cataloguri`* to the end of the current catalog. This allows one catalog to refer to another.

OASIS TR9401 Catalogs

These catalogs are officially defined by OASIS Technical Resolution TR9401.

A Catalog is a text file that contains a sequence of entries. Of the 13 types of entries that are possible, only six are commonly applicable in XML systems: BASE, CATALOG, OVERRIDE, DELEGATE, PUBLIC, and SYSTEM:

BASE *`uri`*

Catalog entries can contain relative URIs. The BASE entry changes the base URI for subsequent relative URIs. The initial base URI is the URI of the *catalog* file.

In XML Catalogs, this functionality is provided by the closest applicable `xml:base` attribute, usually on the surrounding `catalog` or `group` element.

CATALOG *`cataloguri`*

This entry serves the same purpose as the `nextCatalog` entry in XML Catalogs.

OVERRIDE *`YES|NO`*

This entry enables or disables overriding of system identifiers for subsequent entries in the catalog file.

In XML Catalogs, this functionality is provided by the closest applicable `prefer` attribute on the surrounding `catalog` or `group` element.

An override value of “yes” is equivalent to “prefer="public"”.

DELEGATE *`pubid-prefix cataloguri`*

This entry serves the same purpose as the `delegate` entry in XML Catalogs.

PUBLIC *`pubid systemuri`*

This entry serves the same purpose as the `public` entry in XML Catalogs.

SYSTEM *`sysid systemuri`*

This entry serves the same purpose as the `system` entry in XML Catalogs.

XCatalogs

The Resolver classes also understand the XCatalog format supported by Apache.

Resolution Semantics

Resolution is performed in roughly the following way:

1. If a system entry matches the specified system identifier, it is used.
2. If no system entry matches the specified system identifier, but a rewrite entry matches, it is used.
3. If a public entry matches the specified public identifier and either `prefer` is public or no system identifier is provided, it is used.
4. If no exact match was found, but it matches one or more of the partial identifiers specified in delegate entries, the delegated catalogs are searched for a matching identifier.

For a more detailed description of resolution semantics, including the treatment of multiple catalog files and the complete rules for delegation, consult the XML Catalog standard.

Controlling the Catalog Resolver

The Resolver classes uses either Java system properties or a standard Java properties file to establish an initial environment. The property file, if it is used, must be called `CatalogManager.properties` and must be somewhere on your `CLASSPATH`. The following properties are supported:

System property `xml.catalog.files`; CatalogManager property `catalogs`

A semicolon-delimited list of catalog files. These are the catalog files that are initially consulted for resolution.

Unless you are incorporating the resolver classes into your own applications, and subsequently establishing an initial set of catalog files through some other means, at least one file must be specified, or all resolution will fail.

System property `xml.catalog.prefer`; CatalogManager property `prefer`

The initial prefer setting, either `public` or `system`.

System property `xml.catalog.verbosity`; CatalogManager property `verbosity`

An indication of how much status/debugging information you want to receive. The value is a number; the larger the number, the more information you will receive. A setting of 0 turns off all status information.

System property `xml.catalog.staticCatalog`; CatalogManager property `static-catalog`

> In the course of processing, an application may parse several XML documents. If you are using the built-in `CatalogResolver`, this option controls whether or not a new instance of the resolver is constructed for each parse. For performance reasons, using a value of `yes`, indicating that a static catalog should be used for all parsing, is probably best.

System property `xml.catalog.allowPI`; CatalogManager property `allow-oasis-xml-catalog-pi`

> This setting allows you to toggle whether or not the resolver classes obey the `<?oasis-xml-catalog?>` processing instruction.

System property `xml.catalog.className`; CatalogManager property `catalog-class-name`

> If you're using the convenience classes `org.apache.xml.resolver.tools.*`), this setting allows you to specify an alternate class name to use for the underlying catalog.

CatalogManager property `relative-catalogs`

> If `relative-catalogs` is `yes`, relative catalogs in the `catalogs` property will be left relative; otherwise they will be made absolute with respect to the base URI of the `CatalogManager.properties` file. This setting has no effect on catalogs loaded from the `xml.catalogs.files` system property (which are always returned unchanged).

System property `xml.catalog.ignoreMissing`

> By default, the resolver will issue warning messages if it cannot find a `CatalogManager.properties` file, or if resources are missing in that file. However if *either* `xml.catalog.ignoreMissing` is `yes`, or catalog files are specified with the `xml.catalog.catalogs` system property, this warning will be suppressed.

My `CatalogManager.properties` file looks like this:

Example 2. Example CatalogManager.properties File

```
#CatalogManager.properties

verbosity=1

relative-catalogs=yes

# Always use semicolons in this list
catalogs=./xcatalog;/share/doctypes/catalog;/share/doctypes/xcatalog

prefer=public

static-catalog=yes

allow-oasis-xml-catalog-pi=yes

catalog-class-name=org.apache.xml.resolver.Resolver
```

Using Catalogs with Popular Applications

A number of popular applications provide easy access to catalog resolution:

Xalan

Recent development versions of Xalan include new command-line switches for setting the resolvers. You can use them directly with the `org.apache.xml.resolver.tools` classes:

```
-URIRESOLVER org.apache.xml.resolver.tools.CatalogResolver
-ENTITYRESOLVER org.apache.xml.resolver.tools.CatalogResolver
```

Saxon

Similarly, Saxon supports command-line access to the resolvers:

```
-x org.apache.xml.resolver.tools.ResolvingXMLReader
-y org.apache.xml.resolver.tools.ResolvingXMLReader
-r org.apache.xml.resolver.tools.CatalogResolver
```

The *`-x`* class is used to read source documents, the *`-y`* class is used to read stylesheets.

XP

To use XP, simply use the included `org.apache.xml.xp.xml.sax.Driver` class instead of the default XP driver.

XT

Similarly, for XT, use the `org.apache.xml.xt.xsl.sax.Driver` class.

Adding Catalog Support to Your Applications

If you work with Java applications using a parser that supports the SAX1 `Parser` interface or the SAX2 `XMLReader` interface, adding Catalog support to your applications is a snap. The SAX interfaces include an `entityResolver` hook designed to provide an application with an opportunity to do this sort of indirection. The Resolver classes implements the full OASIS Catalog semantics and provide an appropriate class that implements the SAX `entityResolver` interface.

All you have to do is setup a `org.apache.xml.resolver.tools.CatalogResolver` on your parser's `entityResolver` hook. The code listing in Example 3, “Adding a CatalogResolver to Your Parser” demonstrates how straightforward this is:

Example 3. Adding a CatalogResolver to Your Parser

```
import org.apache.xml.resolver.tools.CatalogResolver;
...
    CatalogResolver cr = new CatalogResolver();
...
    yourParser.setEntityResolver(cr)
```

The system catalogs are loaded from the `CatalogManager.properties` file on your `CLASSPATH`. (For all the gory details about these classes, consult the API documentation.) You can explicitly parse your own catalogs (perhaps taken from command line arguments or a Preferences dialog) instead of or in addition to the system catalogs.

Catalogs In Action

The Resolver distribution includes a couple of test programs, **resolver** and **xparse**, that you can use to see how this all works.

Using resolver

The **resolver** application simply performs a catalog lookup and returns the result. Given the following catalog:

Example 4. An Example XML Catalog File

```
<catalog xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog">

<public publicId="-//Example//DTD Example V1.0//EN"
        uri="example.dtd"/>

</catalog>
```

A demonstration of public identifier resolution can be achieved like this:

Example 5. Resolving Identifiers

```
$ java org.apache.xml.resolver.apps.resolver -d 2 -c example/catalog.xml \
  -p "-//Example//DTD Example V1.0//EN" public
Loading catalog: ./catalog
Loading catalog: /share/doctypes/catalog
Resolve PUBLIC (publicid, systemid):
  public id: -//Example//DTD Example V1.0//EN
Loading catalog: file:/share/doctypes/entities.cat
Loading catalog: /share/doctypes/xcatalog
Loading catalog: example/catalog.xml
Result: file:/share/documents/articles/sun/2001/01-resolver/example/example.dtd
```

Using xparse

The **xparse** command simply sets up a catalog resolver and then parses a document. Any external entities encountered during the parse are resolved appropriately using the catalogs provided.

In order to use the program, you must have the `resolver.jar` file on your `CLASSPATH` and you must be using JAXP. In the examples that follow, I've already got these files on my `CLASSPATH`.

The file we'll be parsing is shown in Example 6, “An xparse Example File”.

Example 6. An xparse Example File

```
<!DOCTYPE example PUBLIC "-//Example//DTD Example V1.0//EN"
                  "file:///dev/this/does/not/exist/example.dtd">
<example>
<p>This is just a trivial example.</p>
</example>
```

First let's look at what happens if you try to parse this document without any catalogs. For this example, I deleted the `catalogs` entry on my `CatalogManager.properties` file. As expected, the parse fails:

Example 7. Parsing Without a Catalog

```
$ java org.apache.xml.resolver.apps.xparse -d 2 example.xml
Attempting validating, namespace-aware parse
Fatal error:example.xml:2:External entity not found:
   "file:///dev/this/does/not/exist/example.dtd".
Parse failed with 1 error and no warnings.
```

With an appropriate catalog file, we can map the public identifier to a local copy of the DTD. We could have mapped the system identifier instead (or as well), but the public identifier is probably more stable.

Using a command-line option to specify the catalog, I can now successfully parse the document:

Example 8. Parsing With a Catalog

```
$ java org.apache.xml.resolver.apps.xparse -d 2 -c catalog.xml example.xml
Loading catalog: catalog.xml
Attempting validating, namespace-aware parse
Resolved public: -//Example//DTD Example V1.0//EN
        file:/share/documents/articles/sun/2001/01-resolver/example/example.dtd
Parse succeeded (0.32) with no errors and no warnings.
```

The additional messages in each of these examples arise as a consequence of the debugging option, *`-d 2`*. In practice, you can make resolution silent.

May All Your Names Resolve Successfully!

We hope that these classes become a standard part of your toolkit. Incorporating this code allows you to utilize public identifiers in XML documents with the confidence that you will be able to move those documents from one system to another and around the Web.

---

[1] It is technically possible to use a proxy to transparently cache remote resources, thus making the cached resources available even when the real hosts are unreachable. In practice, this requires more technical skill

(and system administration access) than many users have available. And I don't know of any such proxies that can be configured to provide preferential caching to the specific resources that are needed. Without such preferential treatment, its difficult to be sure that the resources you need are actually in the cache.

[2] URIs that rely on the domain name system to identify objects (in other words, all URLs) are addresses, not names, even though the domain name provides a level of indirection and the illusion of a stable name.