Entity management in XML applications

Control how your XML app discovers and accesses entities

Skill Level: Introductory

Leigh Dodds (leigh@xmlhack.com)
Developer and editor
Ingenta, Ltd.

30 Sep 2003

Entity management is the term used to describe the process for controlling how an XML application discovers and accesses external resources known as entities. Entity management is an often overlooked aspect of XML application development. However, the technique offers a number of advantages. This tutorial presents the basic principles of entity management through the concept of an XML catalog -- an address book that defines mappings from resources referenced in XML documents (such as a stylesheet or schema) to URI references (such as file system paths or URLs).

Section 1. Introduction

What is this tutorial about?

Entity management is the term used to describe the process for controlling how an XML application discovers and accesses external resources known as **entities**. Entity management is an often overlooked aspect of XML application development; however, the technique offers a number of advantages, including:

- **Increased robustness**, by removing sensitivity to network failures
- **Better security**, by reducing the opportunity for external resources to be

© Copyright IBM Corporation 2003. All rights reserved.

compromised

- **Improved performance**, by reducing sensitivity to network bottlenecks
- **Greater interoperability**, by granting local control over how resources are located

This tutorial presents the basic principles of entity management through the concept of an **XML catalog** -- an address book that defines mappings from resources referenced in XML documents (such as a stylesheet or schema) to URI references (such as file system paths or URLs). A catalog is therefore used to control the external resources that an application accesses. The XML Catalog specification, which provides a standard means of describing these catalogs using a simple XML format, is also introduced using working examples.

Tutorial examples use the open source Resolver API, provided as part of the Apache XML Commons project, to demonstrate entity management in practice as well as to illustrate how to implement an XML application with XML catalog support through the Java API for XML Parsing (JAXP).

Should I take this tutorial?

This tutorial is aimed at Java developers who are already familiar with XML application development, particularly JAXP. While some examples illustrate how entity management works in conjunction with XSL transformations, no detailed knowledge of XSLT is required. For pointers to other tutorials covering JAXP and XSLT, see Resources.

This tutorial will also be of interest to application developers who want to learn more about XML catalogs in general so that they may get the most flexibility from applications that already provide support for entity management using the OASIS XML Catalog format. A number of open source XML frameworks and parsers, including Apache Cocoon, already include this support.

Tools

To compile and run the Java examples and applications introduced later in the tutorial, the reader will need a Java 2 Development Kit installed. The Java 2 SDK 1.4.2 can be downloaded from http://java.sun.com/j2se/.

Other than a text or XML editor for viewing the example documents, no other tools are required.

See Installing the XML Commons Resolver classes for instructions on how to download and configure the Apache XML Resolver classes used throughout the

© Copyright IBM Corporation 2003. All rights reserved.

examples.

The tutorial examples are available from x-entmngexamples.zip. To install the examples, simply extract the files from the archive (preserving the directory structure) into a newly created directory. This directory, referred to as `$EXAMPLES_HOME` throughout the rest of the tutorial, contains the following sub-directories:

- `bin` -- shell scripts for running the resolver command-line tool
- `basics` -- the example catalogs referenced in Exploring XML catalogs
- `jaxp` -- Java source code and example files referenced in Adding catalog support to XML applications

---

Section 2. Entity management

What is entity management?

By design, XML applications are very tightly integrated with the Internet. XML and its related specifications make heavy use of URIs for referencing resources such as DTDs, schemas, namespaces, and stylesheets. A typical XML parser is capable of parsing XML documents and loading schemas directly from the Internet, while XSLT stylesheets can access any Web-accessible XML resource through the XSLT `document()` function. With the rise of Web service-based architectures, this reliance on remote resources will continue to grow.

While it is very easy to focus on higher-level functionality in an XML application, developers should consider what this fundamental layer of Web integration means for their applications, particularly regarding issues of performance, stability, and security.

Consider a simple XML application that processes XML documents, such as purchase orders, conforming to an industry standard DTD. It is not unusual for such applications to validate documents as they arrive to confirm that the data conforms to the standard. A validating parser loads this standard DTD before processing each document. If the DTD is hosted on a remote server (for example, by an industry body), then parsing each individual XML document results in an HTTP request from the XML parser to fetch the DTD, which is then parsed and applied to the document.

Here are a few important questions you should ask about this kind of architecture:

© Copyright IBM Corporation 2003. All rights reserved.

- What will happen if a network interruption occurs and the server (and hence the DTD) is not available? Will the application fail?
- If network traffic is heavy between the application and server, will the application slowly grind to a halt?
- If the DTD is updated to a new version, subtly changing the class of valid documents, will the application fail?
- What if a client sends in a purchase order document with a reference to another DTD entirely?

The most visible side effects -- performance and stability problems -- are arguably the least worrisome; the invisible changes are more of a concern as they may not be spotted for some time.

On a smaller but no less frustrating scale, you can even encounter these same kinds of issues when moving XML documents and applications between machines -- for example, between development and production environments. While you might quickly solve this by manually editing a number of configuration files for small applications, scale this up to the level of a content management system -- which may contain many thousands of documents -- and it's obvious that the problem can't be solved with a quick series of hand edits.

Resources loaded by XML applications -- DTDs, schemas, and so forth -- are generally known as entities. Controlling how an XML application, usually a parser, discovers and loads these entities is known as entity management. This tutorial discusses how to introduce entity management functionality into XML applications using a technology known as XML catalogs.

System and public identifiers

The XML specification introduces the concept of an **external identifier**, which is used to locate resources referenced from XML documents, typically references to DTDs (in a DOCTYPE declaration), entity files (within ENTITY declarations in a DTD), or other forms of XML schema. See Resources for pointers to XML tutorials to review the basic DTD syntax.

An external identifier consists of one or two components:

- A mandatory **system identifier**
- An optional **public identifier**

System identifiers are quite straightforward and are URI references:

© Copyright IBM Corporation 2003. All rights reserved.

```
<!DOCTYPE example SYSTEM "http://www.examples.com/example.dtd">
<!DOCTYPE example SYSTEM "file:///c:/dtds/example.dtd">
<!ENTITY example-entity SYSTEM "example-entity.xml">
```

Public identifiers are a concept that XML has inherited from SGML and have been retained for backwards compatibility. An XML entity or DTD reference can use a public identifier, but it must also provide a system identifier as a fall-back option:

```
<!DOCTYPE example PUBLIC "-//Example Inc.//Example DTD//EN"
                         "http://www.examples.com/example.dtd">
```

The XML specification does not require XML processors to understand public identifiers and therefore does not specify them further. However, the intention is that they should provide some globally unique identifier for the resource, whereas the system identifier indicates just one possible location. Public identifiers are therefore best used in conjunction with an entity management system.

More on public identifiers

While XML does not require a particular format for public identifiers, SGML does define a syntax for describing what is known as a **Formal Public Identifier (FPI)**. Some additional examples of FPIs include:

```
-//OASIS//DTD XML catalogs V1.0//EN
+//IDN example.com//Another Example DTD//EN
```

Construct an FPI as follows, with each section separated from the one preceeding it by a double-slash (`//`):

1. The initial character indicates whether the identifier is officially registered. A `+` sign indicates a registered public identifier, a `-` for all others.
2. The second portion of the identifier is the name of the institution, organization, or group defining the identifier.
3. The third portion of the identifier is the name of the entity.
4. The last portion of the identifier is an ISO language code indicating the language of the resource.

While not strictly required, it is worth adopting the FPI syntax for creating public identifiers. All publicly-defined XML specifications, including those from the W3C and OASIS, use FPIs to construct public identifiers for their DTDs and entities. The

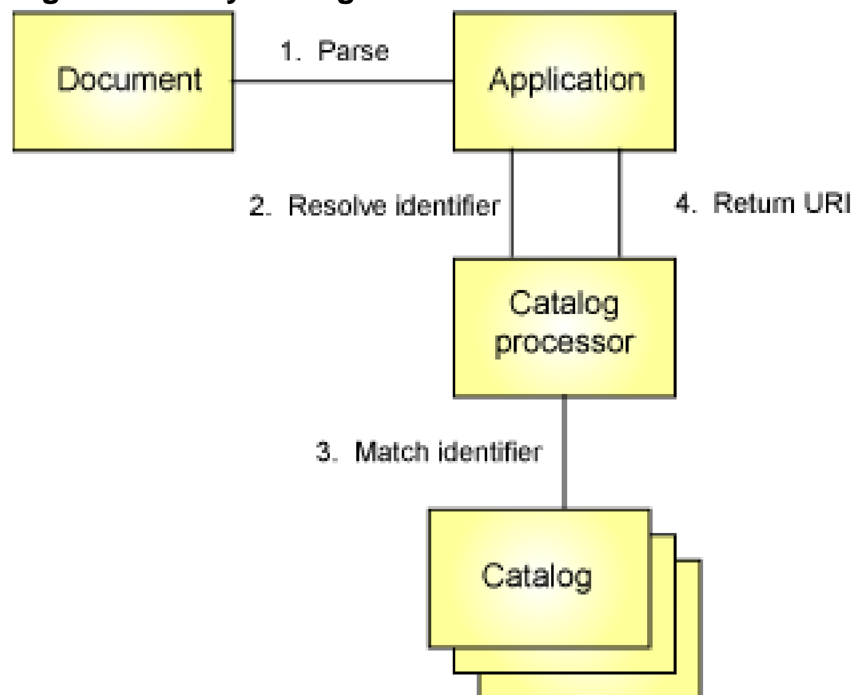
© Copyright IBM Corporation 2003. All rights reserved.

practice is therefore in widespread use.

Entity management basics

The XML Catalog specification, introduced later in this tutorial (see Exploring XML catalogs), describes a basic model that usefully illustrates the relevant roles and responsibilities in entity management. This is described below and summarized in the following diagram:

Figure 1. Entity management model



The process begins when an application needs to access an external resource. These applications are typically an XML parser or an XSLT stylesheet processor, but the model can be generalized to *any* application.

Rather than access the external resource directly -- for example, through a URL provided in a system identifier -- the application uses another component known as a **catalog processor** to discover the correct location of the resource -- for example, a local copy of a remote DTD or stylesheet.

To achieve this, the application provides one or more identifiers to the catalog processor, which uses them to determine the correct resource. A catalog processor manages a **catalog**, which contains a **mapping** of identifiers to URIs (a URL, for instance). Catalog-aware applications typically allow the user to configure which catalog the catalog processor will use.

A catalog may actually be made up of one or more *catalog files* that conform to the

© Copyright IBM Corporation 2003. All rights reserved.

XML Catalog specification. However, a catalog processor may support other formats for describing a catalog, including reading the mappings from a database table. This tutorial concentrates solely on catalog processors that process XML catalog files.

Using the mapping data contained in the catalog, the catalog processor is said to *resolve* the identifiers provided by the application into a URI that the application can then access to obtain the resource.

Benefits of XML catalogs

It should be obvious from the model described in Entity management basics that a catalog is essentially just a means of indirection -- substituting a resource reference obtained from an XML document or stylesheet with an alternative preferred location configured by the application author or user. This indirection brings a number of advantages:

- **Performance** -- By allowing local resources, such as those on the same machine or local area network, to be substituted for resources otherwise only available remotely, catalogs can increase application performance by reducing (or removing) network latency incurred when accessing those resources.
- **Stability** -- By allowing an application to rely on local resources, the application is less likely to be affected by remote server failures or other problems that may make remote resources unavailable.
- **Interoperability** -- XML documents become more portable when the processing system can substitute hard-coded resource references with alternatives. The fallback involves updating the XML documents every time a resource changes location. Catalogs can therefore be particularly advantageous in systems (such as a content management system) that involve archiving and re-processing of XML documents
- **Security** -- By ensuring that an application only processes locally controlled resources, extra guarantees can be made about security -- specifically that the application will not access remote resources that may be compromised, either accidentally or maliciously.

While the advantages of adding catalog support to an application shouldn't be overplayed -- after all, fetching and parsing of resources are likely to be only small factors in application performance -- it is low-level optimization that is often missed by XML application developers. While the core APIs, including SAX and JAXP, offer support for entity management, the fact that it is not mandated in the XML specification means that it is often overlooked.

The details of making an application catalog aware are covered in Adding catalog

© Copyright IBM Corporation 2003. All rights reserved.

support to XML applications .

Section recap

This section introduced the concept of entity management and highlighted some potential low-level issues in XML applications that you can solve by adding support for XML catalogs. These issues include performance, stability, security, and interoperability of data.

The section reviewed the different kinds of identifiers used in XML applications including the notion of an external identifier, which is composed of a system and an optional public identifier. The Formal Public Identifier (FPI) syntax for public identifiers was also introduced as a best practice means of creating these types of identifiers.

The basic model of entity management was introduced, including the notion of a catalog processor, which is responsible for resolving identifiers supplied by an XML application into URI references. A catalog contains mappings from known public and system identifiers, and URIs, to alternatives. These mappings may be spread across one or more catalog files.

The following section introduces the XML Catalog specification, which defines an XML syntax for describing mappings between identifiers. Following this, the tutorial moves on to describe how to adapt an application to become catalog aware.

Section 3. Getting started with the Resolver API

Section overview

This section reviews how to install the Apache XML Commons Resolver API, which is used throughout the rest of this tutorial. Also introduced are the command-line applications provided with the API, which are useful for exploring how a catalog processor will respond to requests to resolve various combinations of URIs and system and public identifiers.

The Resolver API is provided as part of the Apache XML Commons project, which bundles together a number of standard XML APIs and related utilities shared by a number of different Apache XML projects.

While available as part of the main XML Commons distribution, the Resolver API is

© Copyright IBM Corporation 2003. All rights reserved.

also available as a separate bundle for applications that only need to use this facility. The installation of this distribution is discussed next.

Installing the XML Commons Resolver classes

All of the the Apache XML Commons distributions are available from: http://xml.apache.org/dist/commons/.

The Resolver API 1.0 release can be downloaded from: http://apache.us.lucid.dk/xml/commons/.

Installation is simple: Download the above file and unzip the archive into a local directory. This directory will be referred to as $RESOLVER_HOME from this point forward. After you unzip the distribution, the directory contains the following structure:

- `build` -- build directory into which the compiled classes are placed
- `docs` -- contains the Javadoc-generated API documentation
- `src` -- the source code for the Resolver API
- `resolver.xml` -- an Ant build script for compiling the API from the provided source
- `resolver.jar` -- a pre-built version of the API

Building the Resolver API directly from source is outside the scope of this tutorial, but is quite straightforward using the provided Ant script. (See Resources for more information about installing and using the Ant build tool.)

Installation of the API is completed by adding `resolver.jar` to the Java CLASSPATH as follows:

```
set CLASSPATH=$CLASSPATH:$RESOLVER_HOME/resolver.jar
export CLASSPATH
```

The command-line tools

The Resolver API distribution includes several command-line applications that are useful tools for exploring how a catalog processor processes XML catalogs. The first of these is a simple command-line wrapper around the catalog processor component that allows the user to specify a catalog file and various combinations of identifiers to see how the catalog processor will respond. This application called **resolver** will be used extensively in the next section (Exploring XML catalogs).

© Copyright IBM Corporation 2003. All rights reserved.

The full command-line for invoking this application is as follows:

```
java org.apache.xml.resolver.apps.resolver [options] [keyword]
```

The examples (available for download from Tools) include both a Unix shell script and a Windows batch file for invoking this application as follows:

```
$EXAMPLES_HOME/bin/resolver.sh [options] [keyword]
```

Before using these scripts, edit them to ensure that the value of $RESOLVER_HOME conforms to the local installation of the Resolver API. Adding the `$EXAMPLES_HOME/bin` directory to the PATH further simplifies the command-line, allowing `resolver.sh` to be invoked directly.

The command-line options

For the full set of command-line options for this class, consult the Resolver API Javadoc for the `org.apache.xml.resolver.apps.resolve` class. The options useful for this tutorial are summarized here:

- `-c` -- specifies the catalog file to load
- `-p` -- specifies a public identifier to be resolved
- `-s` -- specifies a system identifier to be resolved
- `-u` -- specifies a URI to be resolved

The command-line ends with a keyword indicating the type of resolution that the catalog processor should perform. Again, a number of alternatives cover different types of XML entities (such as DTD, entity, and notation); however, the only two used in this tutorial are:

- `doctype` -- simulates resolution of identifiers supplied in a DOCTYPE declaration
- `uri` -- simulates resolution of a URI, for example as specified in the XSLT `document` function

Some examples illustrate how these options are used:

```
resolver.sh -c catalog.xml -s http://www.example.com/dtd/example.dtd doctype
```


© Copyright IBM Corporation 2003. All rights reserved.

The above instructs the resolver application to load the catalog `catalog.xml` and attempt to resolve the system identifier `http://www.example.com/dtd/example.dtd`. Resolving a public identifier works similarly, substituting the `-p` parameter and public identifier in place of the `-s` parameter.

Both a public and a system identifier can be specified as follows:

```
resolver.sh -c catalog.xml \
            -s http://www.example.com/dtd/example.dtd \
            -p "-//Example Inc.//Example DTD//EN" doctype
```

The above, which should be entered as a single line, requests that the processor try to resolve both identifiers.

Simulating the resolution of a URL involves using the `-u` option and the `uri` keyword as follows:

```
resolver.sh -c catalog.xml -u http://www.remote-site.com/stylesheet.xsl uri
```

The other useful application is **xread**, a simple command-line XML parser that incorporates XML catalog support. This application, while useful for testing whether a given document will parse using a catalog, is not covered further in this tutorial. For further information on using it from the command line, see the Resolver API Javadoc for the `org.apache.xml.resolver.apps.xread` class. It should be obvious how to customize the shell scripts included in the examples to run this application instead.

Section recap

This section described the installation of the Apache XML Resolver API, which is available as a standalone distribution from the Apache XML Commons project.

This API provides two useful command-line tools, one of which is used extensively in the next section to explore the XML Catalog specification. This application enables the simulation of catalog resolution for different catalogs and combinations of XML identifiers.

Section 4. Exploring XML catalogs

© Copyright IBM Corporation 2003. All rights reserved.

Section overview

This section introduces the catalog format described in the XML Catalog specification. The specification, produced by the OASIS Entity Resolution Technical Committee, defines an XML syntax for mapping identifiers to URI references. The examples in this tutorial are based on the XML catalog 1.0 document published on 3rd June 2003.

The XML catalog vocabulary is very straightforward, consisting of 11 elements that can be divided up into the following broad groups:

- **Document structure** -- `catalog` is the root element of an XML catalog, while `group` can be used to group together catalog entries.
- **Identifier mapping elements** -- The `system`, `public`, and `uri` elements each map a specific type of identifier to a URI.
- **Identifier rewriting elements** -- The `rewriteSystem` and `rewriteURI` elements allow system identifiers and URIs to be rewritten by substituting their existing prefixes for alternatives.
- **Catalog organization** -- The `nextCatalog`, `delegatePublic`, `delegateURI`, and `delegateSystem` elements allow a catalog to be split up into several catalog entry files, making them easier to maintain.

In the information that follows, I will introduce each group of elements in turn, illustrating their operation using the `resolve` command-line utility. See Getting started with the Resolver API for instructions on how to install the XML Resolver API and use this tool.

The example files for this section of the tutorial are available in the `$EXAMPLES_HOME/basics` directory. See Tools for instructions on how to obtain the examples.

Basic document structure

The root element of an XML catalog document must be a `catalog` element that's associated with the following namespace URI:

```
urn:oasis:names:tc:entity:xmlns:xml:catalog
```

Therefore, the following is the simplest possible XML catalog document:

© Copyright IBM Corporation 2003. All rights reserved.

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

<!-- other elements here -->

</catalog>
```

The `catalog` element may contain any of the other elements defined in the specification. It has one optional attribute, `prefer`, which is discussed further in The prefer attribute.

The specification defines a `group` element, similar to the `catalog` element, which has the same content model and also supports the `prefer` attribute. The `group` element has no other special meaning and is included merely as a means to apply the `prefer` attribute to a group of other elements.

All of the elements in the XML Catalog specification support an `id` attribute which must contain a unique identifier for that element in the current document. It is not used in catalog resolution and is provided for user convenience so that elements can be easily labeled and identified.

The xml:base attribute

All of the elements in the XML Catalog specification also support the `xml:base` attribute. This attribute, defined in the W3C XML Base specification, is used to specify a base URI that you can use to turn relative URI references into absolute URIs. The value of the `xml:base` attribute specifies an alternative base URI for the current element and all of its children. If this is not specified, an XML application assumes that the base URI for all references is the same as that of the current document -- meaning all references are resolved relative to the current document. The following example illustrates this.

Assume the following XML catalog document has been published at this URL: `http://www.example.com/catalog/catalog.xml`.

```
<catalog
  xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog">
 <system systemId="..." uri="another.dtd"/>
 <public publicId="..." uri="another.dtd"
  xml:base="http://www.somewhere-else.com/catalogs"/>
</catalog>
```

The relative URI in the `uri` attribute on the system element is resolved as `http://www.example.com/catalog/catalog.xml` (relative to the current document). However, the `public` element has an `xml:base` attribute that substitutes an alternate base URI. This means that the absolute URI value for the

© Copyright IBM Corporation 2003. All rights reserved.

uri attribute on the public element is actually
http://www.somewhere-else.com/catalogs/another.dtd.

For more discussion of the xml:base attribute, see the XML Base specification in Resources.

Identifier mapping

The system, public, and uri elements all allow a single identifier to be mapped to a URI. Each element specifies a mapping for a particular type of identifier. For example, the public element defines mapping from public identifiers to URIs.

The elements define this mapping using a common pattern. First, each has a **matching attribute** whose value is the original identifier it matches; the name of this attribute varies with each element. Second, each element has a uri attribute that specifies the replacement URI for that identifier.

For example, the system element's matching attribute is the systemId attribute. Therefore, to map the system identifier
http://www.example.com/dtds/example1.dtd to
http://www.somewhereelse.com/xml/example1.dtd, an XML catalog should contain the following markup:

```
<system
  systemId="http://www.example.com/dtds/example1.dtd"
  uri="http://www.somewhereelse.com/xml/example1.dtd"/>
```

To see this in action, run the command-line resolver (see The command-line tools) as follows:

```
resolver.sh -c example1.cat
            -s http://www.example.com/dtds/example1.dtd
            doctype
```

The file example1.cat is available in the basics directory of the examples download (see Tools). The tool will indicate that it successfully matches the provided system identifier, mapping it to the alternative specified in the catalog.

If the system identifier provided on the command-line (or directly to the catalog processor when embedded in an application) does not match any system identifier in the catalog, then the resolver does not return anything.

You can apply the same process to test mapping public identifiers and URIs. The matching attribute for the public element is the publicId attribute; for the uri element, the matching attribute is called name. See example2.cat and

© Copyright IBM Corporation 2003. All rights reserved.

`example3.cat` for catalogs that demonstrate this syntax.

To try matching a public identifer on the command-line, issue the following command:

```
resolver.sh -c example2.cat
            -p "-//Example Inc.//Example DTD 1//EN"
            doctype
```

Once a catalog processor has successfully matched an identifier using a mapping provided by one of these elements, it will stop processing and return the specified URI. The processor will not continue to look for matches, therefore if a catalog contains multiple mappings for a single identifier, then only the first will ever be used. To see this in action try the following:

```
resolver.sh -c example4.cat
            -s http://www.example.com/dtds/example1.dtd
            doctype
```

The catalog `example4.cat` contains two matches for the same URI. Only the URI specified in the first `system` element is ever returned by the catalog processor. The XML Catalog specification notes that a processor must process each mapping element in document order.

Mappings made using the `system`, `public`, and `uri` elements also take precedence over the identifer rewriting mechanism described next. Therefore, the catalog processor always first attempts a specific match, and then examines the catalog to see whether the identifier should be rewritten only if no match is found.

Identifier rewriting

If the only means of mapping an identifier to a URI were by explicitly listing each one, then creating and maintaining an XML catalog would be very tedious for systems that use many different entities.

The XML Catalog specification therefore provides a simple rewriting mechanism that allows a large set of identifiers to be collectively mapped. This mechanism is especially useful when creating a local mirror of a number of entities that are available from the same location.

The rewriting algorithm involves identifying the set of identifiers that are suitable for mapping through a string prefix. This prefix is then substituted for an alternative.

As an example, assume that several stylesheets are available from `http://www.example.com/xsl/` -- for instance,

© Copyright IBM Corporation 2003. All rights reserved.

`http://www.example.com/xsl/stylesheet1.xsl` and `http://www.example.com/xsl/stylesheet2.xsl`. To create a local mirror of these files, you could download and store them under a local directory, such as `/var/xsl/mirrors/example.com`. To rewrite the original URIs so that a system would instead use the local versions, an XML catalog would contain the following markup:

```
<rewriteURI
  uriStartString="http://www.example.com/xsl/"
  rewritePrefix="/var/xsl/mirrors/example.com/"/>
```

The `uriStartString` mapping attribute specifies to the processor that any URI beginning with the specified prefix should be mapped to an alternative by substituting that string for the alternative value specified in the `rewritePrefix` attribute.

To see this in action, use the following command-line. Note the use of the `uri` keyword to request that the processor perform a URI-based mapping:

```
resolve.sh -c example5.cat
           -u http://www.example.com/xsl/stylesheet1.xsl
           uri
```

The result of the mapping is the following URI:

```
/var/xsl/mirrors/example.com/stylesheet1.xsl
```

Try altering the value of the URI supplied on the command-line to confirm that the processor will match *any* URI with the prefix `http://www.example.com/xsl/`.

Rewriting system identifiers

System identifiers can be mapped in a similar way. The `rewriteSystem` element is used to declare this mapping, with the match prefix being specified by the `systemIdStartString` attribute. The `rewritePrefix` attribute is used as above to declare the alternative prefix. See `example6.cat` for an example of this usage. This catalog also contains a specific mapping for `http://www.example.com/dtds/example2.dtd`. As noted previously, this specific match takes precedence over any applicable rewrites also specified in the catalog. To confirm this, use the following command-line:

```
resolver.sh -c example6.cat
            -s http://www.example.com/dtds/example1.dtd
```


© Copyright IBM Corporation 2003. All rights reserved.

```
            doctype
```

As in example1.cat, the catalog processor maps this to http://www.somewhereelse.com/xml/example1.dtd. However, any other file under http://www.example.com/dtds/ is mapped to the locally mirrored copy. This example also illustrates that catalogs can be used to flexibly map identifiers between remote locations as easily as they can be used to map to local copies.

Public identifiers cannot be rewritten. Public identifiers must always be specifically mapped to a URI using the public element.

The prefer attribute

In addition to an order of precedence between explicit mappings and rewrite-based mappings, the XML Catalog specification also defines an order of precedence for mapping system and public identifiers. It is possible for an XML application to supply both a system and public identifier as input to the catalog resolver, for example from a DOCTYPE of the form:

```
<!DOCTYPE example PUBLIC
          "-//Example Inc.//Example DTD 1//EN"
          "http://www.example.com/dtds/example1.dtd">
```

The specification notes that a catalog processor must always attempt to map system identifiers first. This means that in the scenario where both identifiers are available, mappings specified by the system, rewriteSystem, and delegateSystem (see Catalog organization) elements take precedence.

If the catalog processor is unable to make a match based on the provided system identifier, then it has two options available:

- It may return nothing, in which case the calling XML application uses the original system identifier to obtain the resource
- The processor may continue to search for matches using the public identifier

This behavior is controlled by the prefer attribute, which can have one of two values: "system" or "public". If the value is "system" then the application prefers to use the original system identifier. If the value is "public" then the application prefers that the catalog processor continue searching for matches using the public identifier before using the original system identifier as a fallback.

The group element is provided to allow the setting of the prefer attribute to be

© Copyright IBM Corporation 2003. All rights reserved.

altered for a set of catalog entries, rather than for the catalog as a whole. The following catalog extract, taken from `example7.cat`, illustrates this:

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

  <public publicId="public-id1" uri="..."/>

  <group prefer="public">
     <public publicId="public-id2" uri="..."/>
     <public publicId="public-id3" uri="..."/>
  </group>

</catalog>
```

The catalog entry for `public-id1` is *always* ignored if the application provides a system identifier, whether or not a match is made. This is because the default catalog processor behavior has been set to "prefer system" by the `prefer` attribute on the root element. However, the `public-id2` and `public-id3` identifiers are always checked if no system identifier-based match is made, as the enclosing `group` element alters the processor behaviour to "prefer public".

In general, as catalogs are used as a means of overriding the system identifiers in documents, "prefer public" should always be used. This is the default mode of operation for the XML Resolver API -- meaning if the `prefer` attribute is missing, then the API assumes that it was specified with a value of `public`. You can alter this by explicitly setting the `prefer` attribute in the catalog or by altering a system property (see Configuring the CatalogResolver).

The `prefer` attribute may seem slightly confusing: From the naming, you might expect it to set a preference of whether public or system identifiers should be checked first. But this is not the case -- system identifiers are *always* checked first. This attribute merely controls whether public identifiers will be checked *at all*.

Catalog organization

XML catalogs can become quite large, especially if one set of catalog files is shared between several applications. To make catalogs more manageable, the XML Catalog specification specifies several elements that allow a catalog to be modularized. First among these are the delegation elements, which indicate to a catalog processor that mappings for a specific set of identifiers are contained in an alternative catalog file.

One delegation element exists for each kind of identifier: `delegateSystem`, `delegatePublic`, and `delegateURI`. Similar to how the rewriting elements operate, the set of identifiers whose resolution is to be delegated is defined using an identifier prefix. If the identifier has that string as a prefix, and no other match is

© Copyright IBM Corporation 2003. All rights reserved.

available, then the catalog processor delegates processing to the catalog file specified in the `catalog` attribute.

For example, to ensure that the mappings for all system identifiers beginning with `http://www.example.com/` are specified in a catalog file called `example-sys.cat`, use the `delegateSystem` element as follows:

```
<delegateSystem
   systemIdStartString="http://www.example.com/"
   catalog="example-sys.cat"/>
```

Note the use of the `systemIdStartString` attribute to define the match prefix. To delegate matching of public identifiers, use the `delegatePublic` element and the `publicIdStartString` attribute to specify the match prefix. The `uriStartString` attribute on the `delegateURI` element performs a similar role when delegating matching of URIs.

The delegation elements are only checked if no other matches are available in the catalog. As system identifier-based matches are always preferred over public identifier matches, the `delegateSystem` element takes precedence over any `public` and `delegatePublic` elements.

More on catalog organization

The final element available for organizing catalogs is the `nextCatalog` element. This indicates to the catalog processor that after it has attempted all matches (explicit, rewrites, and those available through delegation), it should load another catalog file and continue processing with that file. The `nextCatalog` element is used as follows:

```
<nextCatalog catalog="another-catalog.cat"/>
```

A catalog file can contain any number of `nextCatalog` elements, thereby indicating that multiple additional catalog files should be loaded and processed.

The delegation elements and `nextCatalog` have two key differences:

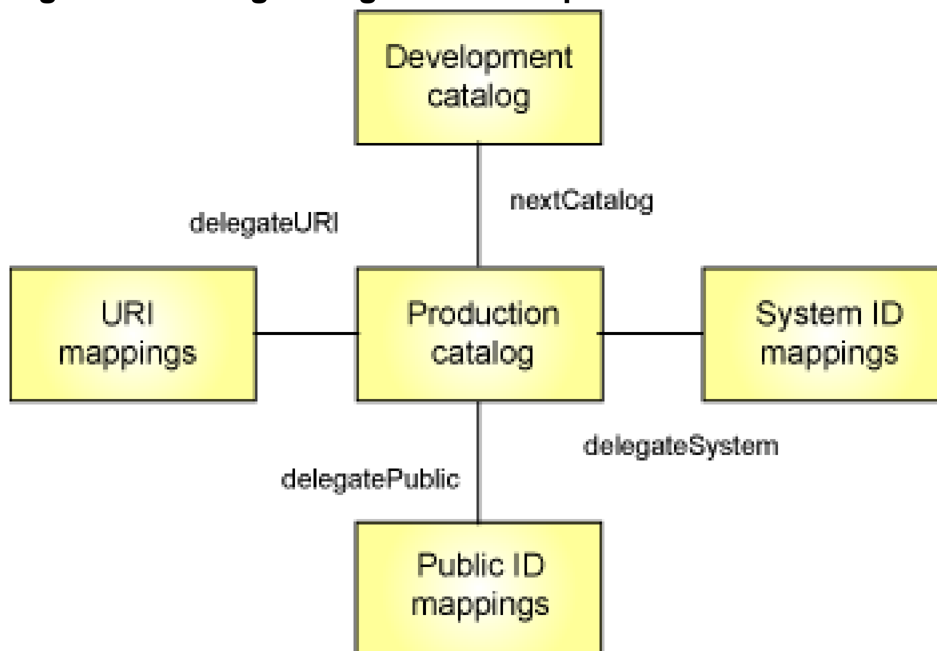
- The catalog file referenced from a `nextCatalog` element is always loaded if a match is not found in the current catalog file, whereas catalog files referenced from the delegation elements are only loaded if the identifier prefix matches.
- The delegation elements allow a single logical catalog to be assembled out of a number of physical catalog entry files. Specifically, these

© Copyright IBM Corporation 2003. All rights reserved.

elements allow a catalog to be broken up into separate catalog files each containing matches for a particular type of identifier, with a specified prefix. This gives very fine-grained control over the catalog structure, whereas the `nextCatalog` element allows entry files to be chained together regardless of their contents.

Controlling the structure of a catalog is very useful when working with development and production versions of an application. The production catalog can be modularized using the delegation elements, with a single catalog file operating as the main entry point. The development version of the application can have a custom catalog that substitutes alternative versions of various resources (such as stylesheets) for those used in production. This development catalog can use the `nextCatalog` element to refer to the main production catalog so that any mappings not overridden in the development catalog will default to their production values. This *star* structure is illustrated in the following diagram:

Figure 2. Catalog arrangement example



You can use similar mechanisms to chain a local catalog to a remote catalog, for example one managed by a standards body or working group. This allows you to selectively cache some resources locally while allowing the default values to be managed remotely.

Section recap

This section provided a detailed walk-through of the XML catalog specification, describing each of the 11 elements and how they operate.

© Copyright IBM Corporation 2003. All rights reserved.

The specification provides the ability to map system and public identifiers as well as URIs to alternative URI references. Identifiers can be explicitly matched or rewritten based on a matching string prefix. Only system and URI identifiers can be rewritten.

Catalogs can also be modularized in several ways using the delegation elements. These elements allow individual catalog files to contain mappings for specific types of identifiers. Like rewriting, delegation is also based on string prefixes. The `nextCatalog` element provides the facility to chain together catalog files. As a whole, these elements allow a single logical catalog to be created out of a number of individual catalog entry files that are either hosted locally or available over a network.

The Catalog specification also defines an order of precedence that describes the order in which a catalog processor should examine each of the above elements. If both public and system identifiers are provided, then system identifier-based matching is always attempted first. The public identifier is only used as a fall-back if the `prefer` setting is `"public"`. URIs are never provided in conjuction with system or public identifiers, so they are considered entirely separately.

Based on these rules, the loose order of precedence when attempting system- and public identifier-based matching is:

1. `system`
2. `rewriteSystem`
3. `delegateSystem`
4. `public`
5. `delegatePublic`

The next section describes how to add catalog support to an XML application by embedding the catalog processor that's bundled in the XML Resolver API.

Section 5. Adding catalog support to XML applications

Section overview

This section provides examples that illustrate how to embed catalog support in XML

© Copyright IBM Corporation 2003. All rights reserved.

applications through Java API for XML Parsing (JAXP). Specifically, this section introduces the the `org.xml.sax.EntityResolver` and `javax.xml.transform.URIResolver` interfaces and an implementation of these interfaces provided by the XML Resolver API.

For pointers to resources on the JAXP, SAX, and DOM APIs, see Resources. For instructions on how to download and install the XML Resolver API, see Getting started with the Resolver API . The source code for the example applications referenced in this section is available from the `$EXAMPLES_HOME/jaxp` directory. See Tools for details on how to download and install the examples.

The EntityResolver interface

The SAX API is used as the foundation for many XML applications, both directly through the use of SAX parsers and indirectly as many DOM implementations are layered upon the SAX API. As an interface designed for low-level manipulation of XML data, SAX provides hooks for entity management as part of its core functionality. This support is provided through the `org.xml.sax.EntityResolver` interface.

You can instruct a SAX `XMLReader` to use an `EntityResolver` through its `setEntityResolver` method. Prior to accessing any external entity referenced in an XML document, the parser will invoke the `EntityResolver`'s `resolveEntity` method. This method has the following signature:

```
public InputSource resolveEntity (String
publicId, String systemId)
  throws SAXException, IOException
```

When the parser encounters a reference to an external entity, it invokes this method on the available entity resolver, passing in the available public and system identifiers as parameters. The resolver is then responsible for constructing a SAX `InputSource` object that the parser can use to access the entity. An `InputSource` is essentially a wrapper around an IO stream that has been constructed by the resolver. The `XMLReader` reads from this stream, closing it after it has finished processing the contents.

A resolver may use any means to determine the location of the entity and may even construct it dynamically based on data in the provided identifiers. If a resolver is unable to determine the location of the entity, then it simply returns `null` which indicates to the parser that it should fetch the external entity itself using the original system identifier.

Adding catalog support to SAX applications

© Copyright IBM Corporation 2003. All rights reserved.

The XML Resolver API provides an implementation of the SAX `EntityResolver` interface which reads XML catalog files to determine how to map external identifiers into URIs. Therefore, adding catalog support to a SAX application involves simply setting an instance of the `org.apache.xml.resolver.tools.CatalogResolver` class as the entity resolver for the `XMLReader` parser.

`SAXCatalogParser.java` , available in the example download (see Tools), demonstrates the use of the `CatalogResolver` class. The following code extract, which omits exception handling, shows the relevant code:

```
SAXParserFactory factory = SAXParserFactory.newInstance();
//create validating parsers for this example
factory.setValidating();
SAXParser parser = factory.newSAXParser();
//get the XMLReader wrapped inside the SAXParser
XMLReader reader = parser.getXMLReader();
//create the EntityResolver
EntityResolver resolver = new CatalogResolver();
//set the EntityResolver
reader.setEntityResolver(resolver);
//tell the parser to parse ...
parser.parse(...);
```

Notice that because the JAXP API wraps a SAX parser (XMLReader) in a `SAXParser` object, but does not provide direct access to all of the underlying methods, the code must first obtain a reference to the `XMLReader` object so that it can call the `setEntityResolver` method.

SAX applications that do not use the JAXP API typically create an `XMLReader` object directly using the `org.xml.sax.helpers.XMLReaderFactory`. These can instead create an instance of the `org.apache.xml.resolver.tools.ResolvingXMLReader` class. This parser automatically creates an underlying parser object using the `XMLReaderFactory` and sets its entity resolver.

Now that I've shown you how to create a catalog-aware application, the next step is to configure the application to use specific XML catalog files.

Configuring the CatalogResolver

The XML Resolver API provides several means of configuring the list of catalog files that its resolver implementation will process. The simplest of these involves setting a Java System property, `xml.catalog.files`. The value of this property should be a semi-colon delimited list of catalog file names. The Resolver classes automatically read the property and process the list of catalog files using them to resolve identifiers. You can set this property from the command-line as follows:

© Copyright IBM Corporation 2003. All rights reserved.

```
java -Dxml.catalog.files=catalog.cat YourApplicationClass
```

The advantage of this method is that the user does not have to use any additional configuration files, and existing applications can be transparently adapted to add catalog support by adding an appropriate `setEntityResolver` method call and altering the command-line.

If the above system property is not set, then the Resolver API classes fall back to attempting to load a Java properties file, `CatalogManager.properties`, which must be available somewhere in the Java `CLASSPATH`. If preferred over a system property, this file should contain a `catalogs` property whose value is a semi-colon separated list of catalog file names. For example:

```
catalogs=catalog1.cat;catalog2.cat
```

If the Resolver classes cannot determine the list of catalog files using either method, then error messages ("Cannot load CatalogManager.properties") are written to the console.

The Resolver classes also support a number of other properties that you can use to control their behavior, such as the default value of the `prefer` attribute. Consult the Javadoc for the `org.apache.xml.resolver.CatalogManager` class for more details.

The Resolver classes also support reading of catalog files programmatically by obtaining a reference to the `Catalog` object that's managed by the `CatalogResolver`:

```
CatalogResolver resolver = new CatalogResolver();
Catalog catalog = resolver.getCatalog();
catalog.parseCatalog("catalog.cat");
...
```

This method is useful if an application already has a configuration file or command-line parameters that are used to indicate the appropriate catalog file(s) for the application to use. This has the advantage of keeping all application configuration in a single place.

Running the example SAX application

The example SAX application, `SAXCatalogParser`, is very straightforward: It accepts a single command-line argument -- the name of an XML file to parse. It then parses and validates this document using the referenced DTD. XML catalogs

© Copyright IBM Corporation 2003. All rights reserved.

supplied using the `xml.catalog.files` system property (or through `CatalogManager.properties`) are used to resolve any external identifiers.

The `jaxp` directory in the examples (see Tools for a download link) contains a simple XML document, `example.xml`, that you can use to test the application. This document consists of a `DOCTYPE` declaration and a single element:

```
<!DOCTYPE example PUBLIC "-//Example Inc.//Example DTD//EN" "loose.dtd">
<example>This is an example document</example>
```

The `DOCTYPE` declaration provides both a public and system identifier, the latter defaulting to a DTD, `loose.dtd`, which is available from the same directory and is reproduced here:

```
<!ELEMENT example (#PCDATA)>
```

As shown, the loose DTD simply declares an XML element called `example` and indicates that it contains text content.

A *strict* version of this DTD is also available -- see `strict.dtd`. This indicates that the `example` element should also have a required `id` attribute:

```
<!ATTLIST example id CDATA #REQUIRED>
```

An XML parser that does not process XML catalogs will only validate the example document using the loose DTD, which is specified in the `DOCTYPE` declaration of the document. However, by creating an appropriate XML catalog and using a catalog-aware parser (the `SAXCatalogParser` application), it is possible to instead use the strict form of the DTD.

The `jaxp` examples directory contains two catalogs:

- **`development.cat`** , which maps the public identifier used in the example document, `-//Example Inc.//Example DTD//EN`, to the loose form of the DTD
- **`production.cat`** , which maps the same public identifier to the strict form of the DTD

These files simulate the use of a loose form of validation for a development environment, but stricter validation checks in a production application.

To parse the example document using the strict DTD use the following command-line:

© Copyright IBM Corporation 2003. All rights reserved.

```
java -Dxml.catalog.files=production.cat SAXCatalogParser example.xml
```

This will generate an error message of the form:

```
Attribute value for "id" is #REQUIRED.
```

This indicates that the parser is using the strict DTD. If you alter the command-line so that it refers to `development.cat`, the document will be successfully parsed.

Adding catalog support to DOM applications

As with SAX-based applications, adding catalog support to a DOM application involves setting an appropriate `EntityResolver` instance for the DOM parser to use.

The code for this differs slightly from the SAX version in that, while many DOM implementations are layered on top of a SAX parser, the JAXP API does not provide direct access to an `XMLReader` object through the JAXP `DocumentBuilder` class. Instead, the `DocumentBuilder` object itself has a `setEntityResolver` method that can be used to configure the entity resolver.

The following example code, which is extracted from `DOMCatalogParser.java` and is available from the `jaxp` directory in the examples (see Tools), demonstrates this:

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
//create validating parsers for this example
factory.setValidating(true);
//create a parser
DocumentBuilder parser = factory.newDocumentBuilder();
//create and set the entity resolver
parser.setEntityResolver( new CatalogResolver() );
//parse ...
parser.parse(...);
```

`DOMCatalogParser.java` differs from `SAXCatalogParser` only in its use of the DOM API rather than SAX. To execute the application, simply change the class names from the examples given in Running the example SAX application:

```
java -Dxml.catalog.files=production.cat DOMCatalogParser example.xml
```

The URI resolver interface

© Copyright IBM Corporation 2003. All rights reserved.

The `URIResolver` instance is part of the `javax.xml.transform` package and fulfills a role similar to the `EntityResolver` interface, except for mapping URIs. The interface has a single method with the following signature:

```
public Source resolve(String href, String base)
   throws TransformerException;
```

The `resolve` method is invoked with two parameters: the first is the URI reference (which may be relative or absolute) and the second is the current base URI (which allows a relative URI to be made absolute). The method returns `null` if the `URIResolver` cannot determine a suitable mapping, and the application should try to resolve the URI itself. If the `URIResolver` can resolve the URI, then it should return an instance of the `javax.xml.transform.Source` interface which provides access to the resource.

The `URIResolver` interface is primarily designed to resolve URIs from XSLT transformations. Specifically, it is used in two separate contexts:

- by `TransformerFactory` instances to resolve URIs specified in the `xsl:import` and `xsl:include` elements
- by `Transformer` objects to resolve URIs that are passed as parameters to the XSLT `document()` function

Both of the above classes have a `setURIResolver` method that can be used to configure the URI resolver.

The `CatalogResolver` class from the Resolver API also implements the `URIResolver` interface. Therefore, you can use this same class to add catalog support to XSLT applications. The following code snippet demonstrates this:

```
//create transformer factory
TransformerFactory factory = TransformerFactory.newInstance();
//create catalog resolver
CatalogResolver resolver = new CatalogResolver();
//set URIResolver for xsl:include and xsl:import
factory.setURIResolver(resolver);
//create transformer from source ...
Transformer transformer = factory.newTransformer(...);
//set URIResolver for the document() function
transfomer.setURIResolver(resolver);
//perform the transform
transformer.transform(..., ...);
```

Running the XSLT Example Application

The use of a `URIResolver` in XSL transformations is demonstrated by

© Copyright IBM Corporation 2003. All rights reserved.

CatalogTransformer.java available in the jaxp directory of the tutorial examples (see Tools for a download link).

This command-line application accepts two parameters. The first is the name of an XML document to transform; the second is the name of the XSLT stylesheet used to perform the transformation. The results of the transformation are written out to the console. The application registers an instance of the CatalogResolver class as the URI resolver for the JAXP Transformer.

An example XSLT stylesheet, example.xsl is also provided. This stylesheet produces an XML document that consists of a single element, result. The text content of this element is obtained by reading the content of the example.xml file through the use of the document() function. Invoke this transformation as follows:

```
java CatalogTransformer document.xml example.xsl
```

This produces a result of the form:

```
<result>This is an example document</result>
```

Because the application is catalog aware, it is possible to have the transformation obtain the text content for the result from an alternative file, for example using a catalog entry of the form:

```
<uri name="example.xml" uri="$EXAMPLES_HOME/jaxp/another-example.xml"/>
```

The file uri.cat in the same directory provides this mapping. Before using the catalog, substitute $EXAMPLES_HOME for the real value of that environment variable (the absolute path to the example directory). To run the CatalogTransformer application using this catalog, use the following command-line:

```
java -Dxml.catalog.files=uri.cat CatalogTransformer document.xml example.xsl
```

The result of this transformation is the following document. Note that the text content of the result element is taken from the XML document specified in the catalog mapping and not that specified in the transformation:

```
<result>This text from an alternative XML document</result>
```

It should be obvious from this trivial example how a URIResolver can be used as an alternative way to parameterize any XSL transformation that reads external

© Copyright IBM Corporation 2003. All rights reserved.

resources.

Section recap

This section introduced the two interfaces provided by the JAXP API that allow an XML application to be extended with XML catalog support. Collectively, these interfaces describe the interaction between an XML application and a catalog processor as introduced in Entity management basics.

The first interface, `EntityResolver`, is defined as part of the SAX API and describes an object capable of mapping an external identifier to an `InputSource`, which allows the parser to obtain the content of the entity.

The second interface, `URIResolver`, is defined as part of the `javax.xml.transform` package and describes an object capable of mapping a URI into a `Source` object that allows the parser to obtain that resource. URI Resolvers are typically used to obtain resources referenced from XSL transformations, specifically the `xsl:import` and `xsl:include` elements along with those provided as parameters to the `document()` function.

The XML Resolver API provides an implementation of both of these interfaces in the form of the `CatalogResolver` class. Example code demonstrated the use of this implementation. The `CatalogResolver` class can be configured in several ways, including a system property, a property file, or through direct programmatic manipulation.

Section 6. Summary

Entity management in XML applications summary

This tutorial introduced the concept of entity management using XML catalogs. The basic model for entity management was reviewed introducing the concept of a catalog processor, which is responsible for mapping resource identifiers provided by an XML application into URI references. The application can then use these URI references to obtain the identified resources. A catalog processor typically reads several catalog files, which define mappings from identifiers to URIs. The tutorial also reviewed the advantages of extending an application to incorporate a catalog processor. These advantages included increased performance, stability, security, and interoperability.

© Copyright IBM Corporation 2003. All rights reserved.

The XML Catalog specification describes a standard format for catalog files that allows mappings from identifiers to URI references to be declared. The format supports both explicitly defined mappings and a simple rewriting mechanism. XML catalogs may also be partitioned up into several individual catalog entry files that collectively form a single logical catalog. I provided example catalog files and illustrated their use with a simple command-line application.

Finally, the tutorial introduced the XML Resolver API, which can be used to extend an XML application with support for the XML Catalog specification. The API provides an implementation of the two key interfaces used to resolve identifiers and URIs in the JAXP application. Example code demonstrated the use of the `EntityResolver` and `URIResolver` interfaces.

The concept of entity management is often overlooked by XML developers but it can greatly improve application performance and flexibility. Indeed, supporting XML catalogs should be considered a best practice among XML developers.

© Copyright IBM Corporation 2003. All rights reserved.

Downloads

Description	Name	Size	Download method
Sample code	x-entmngexamples.zip	12KB	HTTP

Information about download methods

© Copyright IBM Corporation 2003. All rights reserved.

Resources

Learn

- For more information on the Apache XML Commons project and the Resolver API, see the project home page.
- Follow the activities of the OASIS Entity Resolution Technical Committee through their mailing list.
- Visit the XML Cover Pages excellent series of links to further information on SGML and XML catalogs and public identifiers.
- For a refresher on DTD syntax and their use in validation, see the "Validating XML" tutorial (developerWorks, August 2003).
- Learn more about the `xml:base` attribute as defined in the W3C XML Base specification.
- Visit the Ant project home page and manual -- great resources for learning about the Ant build tool.
- Take Nicholas Chase's "Understanding SAX" (developerWorks, July 2003) and "Understanding DOM" (developerWorks, July 2003) tutorials for a good introduction to the SAX and DOM APIs.
- Read "All About JAXP" for a quick review of the JAXP API (developerWorks, May 2005).
- Try the Sun Java/XML tutorial for a more in-depth look at the JAXP API.
- Find more resources on the developerWorks XML and Java technology zones.
- IBM's DB2 database provides not only relational database storage, but also XML-related tools such as the DB2 XML Extender which provides a bridge between XML and relational systems. Visit the DB2 Developer Domain to learn more about DB2.
- Find out how you can earn IBM XML 1.1 certification.
- Stay current with developerWorks technical events and Webcasts.

Get products and technologies

- Download x-entmngexamples.zip and install the sample files for this tutorial.
- Build your next development project with IBM trial software, available for download directly from developerWorks.

About the author

© Copyright IBM Corporation 2003. All rights reserved.

Leigh Dodds

Leigh Dodds is currently employed as an Engineering Manager at Ingenta. He has been developing applications on the Java platform since 1997, and has spent the last four years working with XML and related technologies. Leigh is also a contributing editor to xmlhack, and between February 2000 and June 2002, wrote the weekly "XML-Deviant" column for XML.com. He holds a Bachelors degree in biological Science, and a Masters in computing. As being the father of a lively 18 month old (Ethan) is a full-time job in itself, Leigh currently spends his copious amount of free time investigating how to improve the speed of manufacturing of Round Tuits, which he believes will revolutionize the parenting business.

© Copyright IBM Corporation 2003. All rights reserved.