Axiom User Guide

Axiom User Guide

Table of Contents

List of Figures

List of Examples

Chapter 1. Introduction

What is Axiom?

Axiom stands for *Axis Object Model* and refers to the XML infoset model that is initially developed for Apache Axis2. XML infoset refers to the information included inside the XML, and for programmatic manipulation it is convenient to have a representation of this XML infoset in a language specific manner. For an object oriented language the obvious choice is a model made up of objects. DOM [http://www.w3.org/DOM/] and JDOM [http://www.jdom.org/] are two such XML models. Axiom is conceptually similar to such an XML model by its external behavior but deep down it is very much different. The objective of this tutorial is to introduce the basics of Axiom and explain the best practices to be followed while using Axiom. However, before diving in to the deep end of Axiom it is better to skim the surface and see what it is all about!

For whom is this Tutorial?

This tutorial can be used by anyone who is interested in Axiom and needs to gain a deeper knowledge about the model. However, it is assumed that the reader has a basic understanding of the concepts of XML (such as Namespaces) and a working knowledge of tools such as Ant. Knowledge in similar object models such as DOM will be quite helpful in understanding Axiom, mainly to highlight the differences and similarities between the two, but such knowledge is not assumed. Several links are listed in the section called "Links" that will help understand the basics of XML.

What is Pull Parsing?

Pull parsing is a recent trend in XML processing. The previously popular XML processing frameworks such as SAX [http://en.wikipedia.org/wiki/Simple_API_for_XML] and DOM [http://en.wikipedia.org/wiki/Document_Object_Model] were "push-based" which means the control of the parsing was in the hands of the parser itself. This approach is fine and easy to use, but it was not efficient in handling large XML documents since a complete memory model will be generated in the memory. Pull parsing inverts the control and hence the parser only proceeds at the users command. The user can decide to store or discard events generated from the parser. Axiom is based on pull parsing. To learn more about XML pull parsing see the XML pull parsing introduction [http://www.bearcave.com/software/java/xml/xmlpull.html].

A Bit of History

As mentioned earlier, Axiom was initially developed as part of Axis and simply called *OM*. The original OM was proposed as a store for the pull parser events for later processing, at the Axis summit held in Colombo, Sri Lanka, in September 2004. However, this approach was soon improved and OM was pursued as a complete XML infoset [http://dret.net/glossary/xmlinfoset] model due to its flexibility. Several implementation techniques were attempted during the initial phases. The two most promising techniques were the table based technique and the link list based technique. During the intermediate performance tests the link list based technique proved to be much more memory efficient for smaller and mid sized XML documents. The advantage of the table based OM was only visible for the large and very large XML documents, and hence, the link list based technique was chosen as the most suitable. Initial efforts were focused on implementing the XML infoset (XML Information Set) items which are relevant to the SOAP specification (DTD support, Processing Instruction support, etc were not considered). The advantage of having a tight integration was evident at this stage and this resulted in having SOAP specific interfaces as part of OM rather than a layer on top of it. OM was deliberately made API [http://

en.wikipedia.org/wiki/Application_programming_interface] centric. It allows the implementations to take place independently and swapped without affecting the program later.

Features of Axiom

Axiom is a lightweight, deferred built XML infoset representation based on StAX (JSR 173 [http://www.jcp.org/en/jsr/detail?id=173]), which is the standard streaming pull parser API. The object model can be manipulated as flexibly as any other object model (Such as JDOM [http://www.jdom.org/]), but underneath, the objects will be created only when they are absolutely required. This leads to much less memory intensive programming. Following is a short feature overview of OM.

- **Lightweight**: Axiom is specifically targeted to be lightweight. This is achieved by reducing the depth of the hierarchy, number of methods and the attributes enclosed in the objects. This makes the objects less memory intensive.

- **Deferred building**: By far this is the most important feature of Axiom. The objects are not made unless a need arises for them. This passes the control of building over to the object model itself rather than an external builder.

- **Pull based**: For a deferred building mechanism a pull based parser is required. Axiom is based on StAX [http://today.java.net/pub/a/today/2006/07/20/introduction-to-stax.html], the standard pull parser API.

> Axiom is tightly bound to StAX API. To work with Axiom a StAX compliant parser and the API *must* be present in the classpath.

The Following image shows how Axiom API is viewed by the user

Figure 1.1. Architecture overview

OM Builder wraps the raw xml character stream through the StAX reader API. Hence, the complexities of the pull event stream is transparent to the user.

A Bit About Caching

Since Axiom is a deferred built object model, It incorporates the concept of caching. Caching refers to the creation of the objects while parsing the pull stream. The reason why this is so important is because caching can be turned off in certain situations. If so the parser proceeds without building the object structure. User can extract the raw pull stream from Axiom and use that instead of the object model. In this case it is sometimes beneficial to switch off caching. Chapter 3, *Advanced Operations with Axiom* explains more on accessing the raw pull stream and switching on and off the caching.

Where Does SOAP Come into Play?

In a nutshell SOAP [http://www.w3schools.com/SOAP/soap_intro.asp] is an information exchange protocol based on XML. SOAP has a defined set of XML elements that should be used in messages. Since Axis2 is a "SOAP Engine" and Axiom is built for Axis2, a set of SOAP specific objects were also defined along with Axiom. These SOAP Objects are extensions of the general object model classes.

Chapter 2. Working with Axiom

Obtaining the Axiom Binary

There are several methods through which the Axiom binary can be obtained:

1. If your project uses Maven, then it is sufficient to add Axiom as a dependency, as described in the section called "Using Axiom in a Maven 2 project". Releases are available from the central repository, and snapshots are available from `http://repository.apache.org/snapshots/`.

2. A prebuilt binary distribution can be downloaded [http://ws.apache.org/axiom/download.cgi] from the site. Source distributions are also available. They can be built using Maven 2, by executing **mvn install** in the root directory of the distribution.

3. It is also possible to check out the source code for the current development version (trunk) or previous releases from the Subversion repository and build it using Maven 2. Detailed information on getting the source code from the Subversion repository is found here [http://ws.apache.org/axiom/source-repository.html].

Once the Axiom binary is obtained by any of the above ways, it should be included in the classpath for any of the Axiom based programs to work. Subsequent sections of this guide assume that this build step is complete and `axiom-api-.jar` and `axiom-impl-.jar` are present in the classpath along with the StAX API jar file and a StAX implementation.

Creating an object model programmatically

An object model instance can be created programmatically by instantiating the objects representing the individual nodes of the document and then assembling them into a tree structure. Axiom defines a set of interfaces representing the different node types. E.g. `OMElement` represents an element, while `OMText` represents character data that appears inside an element. Axiom requires that all node instances are created using a factory. The reason for this is to cater for different implementations of the Axiom API, as shown in Figure 2.1, "The Axiom API with different implementations".

Figure 2.1. The Axiom API with different implementations

Two implementations are currently shipped with Axiom:

- The Linked List implementation (LLOM). This is the standard implementation. As the name implies, it uses linked lists to store collections of nodes.

- DOOM (DOM over OM), which adds support for the DOM API.

For each implementation, there are actually three factories: one for plain XML, and the other ones for the two SOAP versions. The factories for the default implementation can be obtained by calling the appropriate static methods in `OMAbstractFactory`. E.g. `OMAbstractFactory.getOMFactory()` will return the proper factory for plain XML. Example 2.1, "Creating an object model programmatically" shows how this factory is used to create several `OMElement` instances.

Example 2.1. Creating an object model programmatically

```
//create a factory
OMFactory factory = OMAbstractFactory.getOMFactory();
//use the factory to create two namespace objects
OMNamespace ns1 = factory.createOMNamespace("bar","x");
OMNamespace ns2 = factory.createOMNamespace("bar1","y");
//use the factory to create three elements
OMElement root = factory.createOMElement("root",ns1);
OMElement elt11 = factory.createOMElement("foo1",ns1);
OMElement elt12 = factory.createOMElement("foo2",ns1);
```

The Axiom API defines several methods to assemble individual objects into a tree structure. The most prominent ones are the following two methods available on `OMElement` instances:

```
public void addChild(OMNode omNode);
public void addAttribute(OMAttribute attr);
```

`addChild` will always add the child as the last child of the parent. Example 2.2, "Usage of `addChild`" shows how this method is used to assemble the three elements created in Example 2.1, "Creating an object model programmatically" into a tree structure.

Example 2.2. Usage of `addChild`

```
//set the children
elt11.addChild(elt21);
elt12.addChild(elt22);
root.addChild(elt11);
root.addChild(elt12);
```

A given node can be removed from the tree by calling the `detach()` method. A node can also be removed from the tree by calling the remove method of the returned iterator which will also call the detach method of the particular node internally.

Creating an object model by parsing an XML document

Creating an object model from an existing document involves a second concept, namely that of a *builder*. The responsibility of the builder is to instantiate nodes corresponding to the information items in the document being parsed. Note that as for programmatically created object models, this still involves the factory, but it is now the builder that will call the `createXxx` methods of the factory.

There are different types of builders, corresponding to different types of input documents, namely: plain XML, SOAP, XOP and MTOM. The appropriate type of builder should be created using the corresponding static method in `OMXMLBuilderFactory`. Example 2.3, "Creating an object model from an input stream" shows the correct method of creating an object model for a plain XML document from an input stream.

> As explained in the section called "Resurrection of the `OMXMLBuilderFactory` API", this is the recommended way of creating a builder starting with Axiom 1.2.11. In previous versions, this was done by instantiating `StAXOMBuilder` or one of its subclasses directly. This approach is still supported as well.

Example 2.3. Creating an object model from an input stream

```
//create the input stream
InputStream in = new FileInputStream(file);

//create the builder
OMXMLParserWrapper builder = OMXMLBuilderFactory.createOMBuilder(in);

//get the root element
OMElement documentElement = builder.getDocumentElement();
```

Several differences exist between a programmatically created OMNode and OMNode instances created by a builder. The most important difference is that the former will have no builder object enclosed, where as the latter always carries a reference to its builder.

As stated earlier, since the object model is built as and when required, each and every OMNode should have a reference to its builder. If this information is not available, it is due to the object being created without a builder. This difference becomes evident when the user tries to get a non caching pull parser from the OMElement. This will be discussed in more detail in Chapter 3, *Advanced Operations with Axiom*.

In order to understand the requirement of the builder reference in each and every OMNode, consider the following scenario. Assume that the parent element is built but the children elements are not. If the parent is asked to iterate through its children, this information is not readily available to the parent element and it should build its children first before attempting to iterate them. In order to provide a reference of the builder, each and every node of the object model should carry the reference to its builder. Each and every OMNode carries a flag that states its build status. Apart from this restriction there are no other constraints that keep the programmer away from mixing up programmatically made OMNode objects with OMNode objects built from builders.

The SOAP object hierarchy is made in the most natural way for a programmer. An inspection of the API will show that it is quite close to the SAAJ API but with no bindings to DOM or any other model. The SOAP classes extend basic Axiom classes (such as the OMElement) hence, one can access a SOAP document either with the abstraction of SOAP or drill down to the underlying XML Object model with a simple casting.

Namespaces

Namespaces are a tricky part of any XML object model and is the same in Axiom. However, the interface to the namespace have been made very simple. OMNamespace is the class that represents a namespace with intentionally removed setter methods. This makes the OMNamespace immutable and allows the underlying implementation to share the objects without any difficulty.

Following are the important methods available in OMElement to handle namespaces.

```
public OMNamespace declareNamespace(String uri, String prefix);
public OMNamespace declareNamespace(OMNamespace namespace);
public OMNamespace findNamespace(String uri, String prefix) throws OMException;
```

The declareNamespaceXX methods are fairly straightforward. Add a namespace to namespace declarations section. Note that a namespace declaration that has already being added will not be added twice. findNamespace is a very handy method to locate a namespace object higher up the object tree. It searches for a matching namespace in its own declarations section and jumps to the parent if it's not found. The search progresses up the tree until a matching namespace is found or the root has been reached.

During the serialization a directly created namespace from the factory will only be added to the declarations when that prefix is encountered by the serializer. More of the serialization matters will be discussed in the section called "Serializer".

The following simple code segment shows how the namespaces are dealt in OM

Example 2.4. Creating an OM document with namespaces

```
OMFactory factory = OMAbstractFactory.getOMFactory();
OMNamespace ns1 = factory.createOMNamespace("bar","x");
OMElement root = factory.createOMElement("root",ns1);
OMNamespace ns2 = root.declareNamespace("bar1","y");
OMElement elt1 = factory.createOMElement("foo",ns1);
OMElement elt2 = factory.createOMElement("yuck",ns2);
OMText txt1 = factory.createOMText(elt2,"blah");
elt2.addChild(txt1);
elt1.addChild(elt2);
root.addChild(elt1);
```

Serialization of the root element produces the following XML:

```
<x:root xmlns:x="bar" xmlns:y="bar1"><x:foo><y:yuck>blah</y:yuck></x:foo></x:root>
```

Traversing

Traversing the object structure can be done in the usual way by using the list of children. Note however, that the child nodes are returned as an iterator. The Iterator supports the 'Axiom way' of accessing elements and is more convenient than a list for sequential access. The following code sample shows how the children can be accessed. The children are of the type `OMNode` that can either be `OMText` or `OMElement`.

```
Iterator children = root.getChildren();
while(children.hasNext()){
        OMNode node = (OMNode)children.next();
}
```

Apart from this, every `OMNode` has links to its siblings. If more thorough navigation is needed the `getNextOMSibling()` and `getPreviousOMSibling()` methods can be used. A more selective set can be chosen by using the `getChildrenWithName(QName)` methods. The `getChildWithName(Qname)` method returns the first child that matches the given `QName` and `getChildrenWithName(QName)` returns a collection containing all the matching children. The advantage of these iterators is that they won't build the whole object structure at once, until its required.

> As explained in the section called "Changes in the behavior of certain iterators", in Axiom 1.2.10 and earlier, all iterator implementations internally stayed one step ahead of their apparent location. This could have the side effect of building elements that are not intended to be built at all.

Serializer

An Axiom tree can be serialized either as the pure object model or the pull event stream. The serialization uses a `XMLStreamWriter` object to write out the output and hence, the same serialization mechanism can be used to write different types of outputs (such as text, binary, etc.).

A caching flag is provided by Axiom to control the building of the in-memory object model. The `OMNode` has two methods, `serializeAndConsume` and `serialize`. When `serializeAndConsume` is

called the cache flag is reset and the serializer does not cache the stream. Hence, the object model will not be built if the cache flag is not set.

The serializer serializes namespaces in the following way:

1. When a namespace that is in the scope but not yet declared is encountered, it will then be declared.

2. When a namespace that is in scope and already declared is encountered, the existing declarations prefix is used.

3. When the namespaces are declared explicitly using the elements `declareNamespace()` method, they will be serialized even if those namespaces are not used in that scope.

Because of this behavior, if a fragment of the XML is serialized, it will also be *namespace qualified* with the necessary namespace declarations.

Here is an example that shows how to write the output to the console, with reference to the earlier code sample- Example 2.3, "Creating an object model from an input stream" that created a SOAP envelope.

```
XMLStreamWriter writer = XMLOutputFactory.newInstance().createXMLStreamWriter(Syst
//dump the output to console with caching
envelope.serialize(writer);
writer.flush();
```

or simply

```
System.out.println(root.toStringWithConsume());
```

The above mentioned features of the serializer forces a correct serialization even if only a part of the Axiom tree is serialized. The following serializations show how the serialization mechanism takes the trouble to accurately figure out the namespaces. The example is from Example 2.4, "Creating an OM document with namespaces" which creates a small object model programmatically. Serialization of the root element produces the following:

```
<x:root xmlns:x="bar" xmlns:y="bar1"><x:foo><y:yuck>blah</y:yuck></x:foo></x:root>
```

However, serialization of only the foo element produces the following:

```
<x:foo xmlns:x="bar"><y:yuck xmlns:y="bar1">blah</y:yuck></x:foo>
```

Note how the serializer puts the relevant namespace declarations in place.

Complete Code for the Axiom based Document Building and Serialization

The following code segment shows how to use Axiom for completely building a document and then serializing it into text pushing the output to the console. Only the important sections are shown here. The complete program listing can be found in Chapter 6, *Appendix*.

```
//create the input stream
InputStream in = new FileInputStream(file);

//create the builder
```

```
OMXMLParserWrapper builder = OMXMLBuilderFactory.createOMBuilder(in);

//get the root element
OMElement documentElement = builder.getDocumentElement();

//dump the out put to console with caching
System.out.println(documentElement.toStringWithConsume());
```

Creating stream readers and writers using `StAXUtils`

The normal way to create `XMLStreamReader` and `XMLStreamWriter` instances is to first request a `XMLInputFactory` or `XMLOutputFactory` instance from the StAX API and then use the factory methods to create the reader or writer.

Doing this every time a reader or writer is created is cumbersome and also introduces some overhead because on every invocation the `newInstance` methods in `XMLInputFactory` and `XMLOutputFactory` go through the process of looking up the StAX implementation to use and creating a new instance of the factory. The only case where this is really needed is when it is necessary to configure the factory in a special way (by setting properties on it).

Axiom has a utility class called `StAXUtils` that provides methods to easily create readers and writers configured with default settings. It also keeps the created factories in a cache to improve performance. The caching occurs by (context) class loader and it is therefore safe to use `StAXUtils` in a runtime environment with a complex class loader hierarchy.

> Axiom 1.2.8 implicitly assumed that `XMLInputFactory` and `XMLOutputFactory` instances are thread safe. This is the case for Woodstox (which is the default StAX implementation used by Axiom), but not e.g. for the StAX implementation shipped with Sun's Java 6 runtime environment. Therefore, when using Axiom versions prior to 1.2.9, you should avoid using `StAXUtils` together with a StAX implementation other than Woodstox, especially in a highly concurrent environment. The issue has been fixed in Axiom 1.2.9. See WSCOMMONS-489 [https://issues.apache.org/jira/browse/WSCOMMONS-489] for more details.

`StAXUtils` also enables a property file based configuration mechanism to change the default factory settings at assembly or deployment time of the application using Axiom. This is described in more details in the section called "Changing the default StAX factory settings".

> The `getInputFactory` and `getOutputFactory` methods in `StAXUtils` give access to the cached factories. In versions prior to 1.2.9, Axiom didn't restrict access to the `setProperty` method of these factories. In principle this makes it possible to change the configuration of these factories for the whole application. However, since this depends on the implementation details of `StAXUtils` (e.g. how factories are cached) and since there is a proper configuration mechanism for that purpose, using this possibility is strongly discouraged. Starting with version 1.2.9, Axiom restricts access to `setProperty` to prevent tampering with the cached factories.

The methods in `StAXUtils` to create readers and writers are rather self-explaining. For example to create an `XMLStreamReader` from an `InputStream`, use the following code:

```
InputStream in = ...
XMLStreamReader reader = StAXUtils.createXMLStreamReader(in);
```

Releasing the parser

As we have seen previously, when creating an object model from a stream, all nodes keep a reference to the builder and thus to the underlying parser. Since an XML parser instance is a heavyweight object, it is important to release it as soon as it is no longer required. The `close` method defined by the `OMSerializable` interface it used for that. Note that it doesn't matter an which node this method is called; it will always close and release the parser for the whole tree. The `build` parameter of the `close` method specifies if the node should be built before closing the parser.

To illustrate this, consider Example 2.3, "Creating an object model from an input stream". After finishing the processing of the object model and assuming that it will not access the object model afterwards, the code should be completed by the following instruction:

```
documentElement.close(false);
```

Closing the parser is especially important in applications that process large numbers of XML documents. In addition, some StAX implementation are able to "recycle" parsers, i.e. to reset a parser instance and to reuse it on another input stream. However, this can only work if the parser has been closed explicitly or if the instance has been marked for finalization by the Java VM. Closing the parser explicitly as shown above will reduce the memory footprint of the application if this type of parser is used.

Exception handling

The fact that Axiom uses deferred building means that a call to a method in one of the object model classes may cause Axiom to read events from the underlying StAX parser, unless the node has already been built or if it was created programmatically. If an I/O error occurs or if the XML document being read is not well formed, an exception will be reported by the parser. This exception is propagated to the user code as an `OMException`.

Note that `OMException` is an unchecked exception. Strictly speaking this is in violation of the principle that unchecked exceptions should be reserved for problems resulting from programming problems. There are however several compelling reasons to use unchecked exceptions in this case:

- The same API is used to work with programmatically created object models and with object models created from an XML document. On a programmatically created object model, an `OMException` in general indicates a programming problem. Moreover one of the design goals of Axiom is to give the user code the illusion that it is interacting with a complete in-memory representation of an XML document, even if behind the scenes Axiom will only create the objects on demand. Using checked exceptions would break that abstraction.

- In most cases, code interacting with the object model will not be able to recover from an `OMException`. Consider for example a utility method that receives an `OMElement` as input and that is supposed to extract some data from this information item. When a parsing error occurs while iterating over the children of that element, there is nothing the utility method could do to recover from this error.

  The only place where it makes sense to catch this type of exception and to attempt to recover from it is in the code that creates the `XMLStreamReader` and builder. It is clear that it would not be reasonable to force developers to declare a checked exception on every method that interacts with an Axiom object model only to allow propagation of that exception to the code that initially created the parser.

The situation is actually quite similar to that encountered in three-tier applications, where the DAO layer in general wraps checked exceptions from the database in an unchecked exception because the business logic and the presentation tier will not be able to recover from these errors.

When catching an OMException special attention should be paid if the code handling the exception again tries to access the object model. Indeed this will inevitably result in another exception being triggered, unless the code only accesses those parts of the tree that have been built successfully. E.g. the following code will give unexpected results because the call to serializeAndConsume will almost certainly trigger another exception:

```
OMElement element = ...
try {
    ...
} catch (OMException ex) {
    ex.printStackTrace();
    element.serializeAndConsume(System.out);
}
```

In Axiom versions prior to 1.2.8, an attempt to access the object model after an exception has been reported by the underlying parser may result in an OutOfMemoryError or cause Axiom to lock itself up in an infinite loop. The reason for this is that in some cases, after throwing an exception, the Woodstox parser (which is the default StAX implementation used by Axiom) is left in an inconsistent state in which it will return an infinite sequence of events. Starting with Axiom 1.2.8, the object model builder will never attempt to read new events from a parser that has previously reported an I/O or parsing error. These versions of Axiom are therefore safe; see WSCOMMONS-372 [https://issues.apache.org/jira/browse/WSCOMMONS-372] for more details.

The discussion in this section suggests that Axiom should make a clear distinction between exceptions caused by parser errors and exceptions caused by programming problems or other errors, e.g. by using distinct subclasses of OMException. This is currently not the case. This issue may be addressed in a future version of Axiom.

Chapter 3. Advanced Operations with Axiom

Accessing the Pull Parser

Axiom is tightly integrated with StAX and the `getXMLStreamReader()` and `getXMLStreamReaderWithoutCaching()` methods in the `OMElement` provides a `XMLStreamReader` object. This `XMLStreamReader` instance has a special capability of switching between the underlying stream and the Axiom object tree if the cache setting is off. However, this functionality is completely transparent to the user. This is further explained in the following paragraphs.

Axiom has the concept of caching, and the Axiom tree is the actual cache of the events fired. However, the requester can choose to get the pull events from the underlying stream rather than the Axiom tree. This can be achieved by getting the pull parser with the cache off. If the pull parser was obtained without switching off cache, the new events fired will be cached and the tree updated. This returned pull parser will switch between the object structure and the stream underneath, and the users need not worry about the differences caused by the switching. The exact pull stream the original document would have provided would be produced even if the Axiom tree was fully or partially built. The `getXMLStreamReaderWithoutCaching()` method is very useful when the events need to be handled in a pull based manner without any intermediate models. This makes such operations faster and efficient.

For consistency reasons once the cache is switched off it cannot be switched on again.

Chapter 4. Integrating Axiom into your project

Using Axiom in a Maven 2 project

Adding Axiom as a dependency

If your project uses Maven 2, it is fairly easy to add Axiom to your project. Simply add the following entries to the `dependencies` section of `pom.xml`:

```
<dependency>
    <groupId>org.apache.ws.commons.axiom</groupId>
    <artifactId>axiom-api</artifactId>
    <version></version>
</dependency>
<dependency>
    <groupId>org.apache.ws.commons.axiom</groupId>
    <artifactId>axiom-impl</artifactId>
    <version></version>
</dependency>
```

All Axiom releases are deployed to the Maven central repository and there is no need to add an entry to the `repositories` section. However, if you want to work with the development (snapshot) version of Axiom, it is necessary to add the Apache Snapshot Repository:

```
<repository>
    <id>apache.snapshots</id>
    <name>Apache Snapshot Repository</name>
    <url>http://repository.apache.org/snapshots/</url>
    <releases>
        <enabled>false</enabled>
    </releases>
</repository>
```

If you are working on another Apache project, you don't need to add the snapshot repository in the POM file since it is already declared in the `org.apache:apache` parent POM.

Managing the JAF and JavaMail dependencies

Axiom requires the Java Activation Framework (JAF) and the JavaMail API to work. There are two commonly used incarnations of these libraries: one is Sun's reference implementation, the other is part of the Geronimo [http://geronimo.apache.org/] project. Axiom declares dependencies on the Geronimo versions (though that might change [https://issues.apache.org/jira/browse/WSCOMMONS-417] in the future). If your project uses another library that depends on JAF and/or JavaMail, but that refers to Sun's implementation, your project will end up with dependencies on two different artifacts implementing the same API.

If you prefer Sun's implementations, then you should change the declaration of the Axiom dependencies in your POM file as follow:

```
<dependency>
```

```
        <groupId>org.apache.ws.commons.axiom</groupId>
        <artifactId>axiom-xxx</artifactId>
        <version></version>
        <exclusions>
            <exclusion>
                <groupId>org.apache.geronimo.specs</groupId>
                <artifactId>geronimo-activation_1.1_spec</artifactId>
            </exclusion>
            <exclusion>
                <groupId>org.apache.geronimo.specs</groupId>
                <artifactId>geronimo-javamail_1.4_spec</artifactId>
            </exclusion>
        </exclusions>
</dependency>
```

If you prefer Geronimo's implementation, then you need to identify the libraries depending on Sun's artifacts (`javax.activation:activation` and `javax.mail:mail`) and add the relevant exclusions. You can use **`mvn dependency:tree`** to easily identify where a transitive dependency comes from.

The choice between Sun's and Geronimo's implementation is to a large extend a question of belief. Note however that the `geronimo-javamail_1.4_spec` artifact used by Axiom only contains the JavaMail API, while Sun's library bundles the API together with the providers for IMAP and POP3. Depending on your use case that might be an advantage or disadvantage.

Applying application wide configuration

Sometimes it is necessary to customize some particular aspects of Axiom for an entire application. There are several things that can be configured through system properties and/or property files. This is also important when using third party applications or libraries that depend on Axiom.

Changing the default StAX factory settings

The information in this section only applies to `XMLStreamReader` or `XMLStreamWriter` instances created using `StAXUtils` (see the section called "Creating stream readers and writers using `StAXUtils`"). Readers and writers created using the standard StAX APIs will keep their default settings as defined by the implementation (or dictated by the StAX specifications).

The feature described in this section was introduced in Axiom 1.2.9.

When creating a new `XMLInputFactory` (resp. `XMLInputFactory`), `StAXUtils` looks for a property file named `XMLInputFactory.properties` (resp. `XMLOutputFactory.properties`) in the classpath, using the same class loader as the one from which the factory is loaded (by default this is the context classloader). If a corresponding resource is found, the properties in that file are applied to the factory using the `XMLInputFactory#setProperty` (resp. `XMLOutputFactory#setProperty`) method.

This feature can be used to set factory properties of type `Boolean`, `Integer` and `String`. The following sections present some sample use cases.

Changing the serialization of the CR-LF character sequence

Section 2.11 of [XML] specifies that an "XML processor must behave as if it normalized all line breaks in external parsed entities (including the document entity) on input, before parsing, by translating both the

two-character sequence #xD #xA and any #xD that is not followed by #xA to a single #xA character." This implies that when a Windows style line ending, i.e. a CR-LF character sequence is serialized literally into an XML document, the CR character will be lost during deserialization. Depending on the use case this may or may not be desirable.

The only way to strictly preserve CR characters is to serialize them as character entities, i.e. ``. This is the default behavior of Woodstox. This can be easily checked using the following Java snippet:

```
OMFactory factory = OMAbstractFactory.getOMFactory();
OMElement element = factory.createOMElement("root", null);
element.setText("Test\r\nwith CRLF");
element.serialize(System.out);
```

This code produces the following output:

```
<root>Test&#xd;
with CRLF</root>
```

> From Axiom's point of view this is actually a reasonable behavior. The reason is that when creating an `OMText` node programmatically, it is easy for the user code to normalize the text content to avoid the appearance of the character entity. On the other hand, if the default behavior was to serialize CR-LF literally (implying that the CR character will be lost during deserialization), it would be difficult (if not impossible) for user code that needs to strictly preserve the text data to construct the object model in such a way as to force serialization of the CR as character entity.

In some cases this behavior may be undesirable[1]. Fortunately Woodstox allows to modify this behavior by changing the value of the `com.ctc.wstx.outputEscapeCr` property on the `XMLOutputFactory`. If Axiom is used (and in particular `StAXUtils`) than this can be achieved by adding a `XMLOutputFactory.properties` file with the following content to the classpath (in the default package):

```
com.ctc.wstx.outputEscapeCr=false
```

Now the output of the Java snippet shown above will be:

```
<root>Test
with CRLF</root>
```

Preserving CDATA sections during parsing

By default, `StAXUtils` creates StAX parsers in coalescing mode. In this mode, the parser will never return two character data events in sequence, while in non coalescing mode, the parser is allowed to break up character data into smaller chunks and to return multiple consecutive character events, which may improve throughput for documents containing large text nodes. It should be noted that `StAXUtils` overrides the default settings mandated by the StAX specification, which specifies that by default, a StAX parser must be in non coalescing mode. The primary reason is compatibility: older versions of Woodstox had coalescing switched on by default.

A side effect of the default settings chosen by Axiom is that by default, CDATA sections are not reported by parser created by `StAXUtils`. The reason is that in coalescing mode, the parser will not only coaelsce adjacent text nodes, but also CDATA sections. Applications that require correct reporting of CDATA sections should therefore disable coalescing. This can be achieved by creating a `XMLInputFactory.properties` file with the following content:

---

[1]See WSTX-94 [http://jira.codehaus.org/browse/WSTX-94] for a discussion about this.

```
javax.xml.stream.isCoalescing=false
```

Migrating from older Axiom versions

This section provides information about changes in Axiom that might impact existing code when migrating from an older version of Axiom. Note that this section is not meant as a change log that lists all changes or new features. Also, before upgrading to a newer Axiom version, you should always check if your code uses methods or classes that have been deprecated. You should fix all deprecation warnings before changing the Axiom version. In general the Javadoc of the deprecated class or method gives you a hint on how to change your code.

Changes in Axiom 1.2.9

System properties used by `OMAbstractFactory`

Prior to Axiom 1.2.9, `OMAbstractFactory` used system properties as defined in the following table to determine the factory implementations to use:
Object model: Plain XML
Method: `getOMFactory()`
System property: `om.factory`
Default: `org.apache.axiom.om.impl.llom.factory.OMLinkedListImplFactory`
Object model: SOAP 1.1
Method: `getSOAP11Factory()`
System property: `soap11.factory`
Default: `org.apache.axiom.soap.impl.llom.soap11.SOAP11Factory`
Object model: SOAP 1.2
Method: `getSOAP12Factory()`
System property: `soap12.factory`
Default: `org.apache.axiom.soap.impl.llom.soap12.SOAP12Factory`

This in principle allowed to mix default factory implementations from different implementations of the Axiom API (e.g. an OMFactory from the LLOM implementation and SOAP factories from DOOM). This however doesn't make sense. The system properties as described above are no longer supported in 1.2.9 and the default Axiom implementation is chosen using the new `org.apache.axiom.om.OMMetaFactory` system property. For LLOM, you should set:

```
org.apache.axiom.om.OMMetaFactory=org.apache.axiom.om.impl.llom.factory.OMLinkedLi
```

This is the default and is equivalent to the defaults in 1.2.8. For DOOM, you should set:

```
org.apache.axiom.om.OMMetaFactory=org.apache.axiom.om.impl.dom.factory.OMDOMMetaFa
```

Factories returned by `StAXUtils`

In versions prior to 1.2.9, the `XMLInputFactory` and `XMLOutputFactory` instances returned by `StAXUtils` were mutable, i.e. it was possible to change the properties of these factories. This is obviously an issue since the factory instances are cached and can be shared among several thread. To avoid programming errors, starting from 1.2.9, the factories are immutable and any attempt to change their state will result in an `IllegalStateException`.

Note that the possibility to change the properties of these factories could be used to apply application wide settings. Starting with 1.2.9, Axiom has a proper mechanism to allow this. This feature is described in the section called "Changing the default StAX factory settings".

Changes in XOP/MTOM handling

In Axiom 1.2.8, `XMLStreamReader` instances provided by Axiom could belong to one of three different categories:

1. `XMLStreamReader` instances delivering plain XML.

2. `XMLStreamReader` instances delivering plain XML and implementing a custom extension to retrieve optimized binary data.

3. `XMLStreamReader` instances representing XOP encoded data.

As explained in WSCOMMONS-485 [https://issues.apache.org/jira/browse/WSCOMMONS-485] and WSCOMMONS-488 [https://issues.apache.org/jira/browse/WSCOMMONS-488], in Axiom 1.2.8, the type of stream reader provided by the API was not always well defined. Sometimes the type of the stream reader even depended on the state of the Axiom tree (i.e. whether some part of it has been accessed or not).

In release 1.2.9 the behavior of Axiom was changed such that it never delivers XOP encoded data unless explicitly requested to do so. By default, any `XMLStreamReader` provided by Axiom now represents plain XML data and optionally implements the `DataHandlerReader` extension to retrieve optimized binary data. An XOP encoded stream can be requested from the `getXOPEncodedStream` method in `XOPUtils`.

Changes in Axiom 1.2.11

Resurrection of the `OMXMLBuilderFactory` API

Historically, `org.apache.axiom.om.impl.llom.factory.OMXMLBuilderFactory` was used to create Axiom trees from XML documents. Unfortunately, this class is located in the wrong package and JAR (it is implementation independent but belongs to LLOM). In Axiom 1.2.10, the standard way to create an Axiom tree was therefore to instantiate `StAXOMBuilder` or one of its subclasses directly. However, this is not optimal for two reasons:

• It relies on the assumption that every implementation of the Axiom API necessarily uses `StAXOMBuilder`. This means that an implementation doesn't have the freedom to provide its own builder implementation (e.g. in order to implement some special optimizations).

• `StAXOMBuilder` and its subclasses belong to packages which have `impl` in their names. This tends to blur the distinction between the public API and internal implementation classes.

Therefore, in Axiom 1.2.11, a new abstract API for creating builder instances was introduced. It is again called `OMXMLBuilderFactory`, but located in the org.apache.axiom.om package. The methods defined by this new API are similar to the ones in the original (now deprecated) `OMXMLBuilderFactory`, so that migration should be easy.

Changes in the behavior of certain iterators

In Axiom 1.2.10 and previous versions, iterators returned by methods such as `OMIterator#getChildren()` internally stayed one step ahead of the node returned by the `next()` method. This meant that sometimes, using such an iterator had the side effect of building elements that were not intended to be built. In Axiom 1.2.11 this behavior was changed such that `next()` no longer builds the nodes it returns. In a few cases, this change may cause issues in existing code. One known instance is the following construct (which was used internally by Axiom itself):

```
while (children.hasNext()) {
```

```
    OMNodeEx omNode = (OMNodeEx) children.next();
    omNode.internalSerializeAndConsume(writer);
}
```

One would expect that the effect of this code is to consume the child nodes. However, in Axiom 1.2.10 this is not the case because `next()` actually builds the node. Note that the code actually doesn't make sense because once a child node has been consumed, it is no longer possible to retrieve the next sibling. Since in Axiom 1.2.11 the call to `next()` no longer builds the child node, this code will indeed trigger an exception.

Another example is the following piece of code which removes all child elements with a given name:

```
Iterator iterator = element.getChildrenWithName(qname);
while (iterator.hasNext()) {
    OMElement child = (OMElement)iterator.next();
    child.detach();
}
```

In Axiom 1.2.10 this works as expected. Indeed, since the iterator stays one node ahead, the current node can be safely removed using the `detach()` method. In Axiom 1.2.11, this is no longer the case and the following code (which also works with previous versions) should be used instead:

```
Iterator iterator = element.getChildrenWithName(qname);
while (iterator.hasNext()) {
    iterator.next();
    iterator.remove();
}
```

Note that this is actually compatible with the behavior of the Java 2 collection framework, where a `ConcurrentModificationException` may be thrown if a thread modifies a collection directly while it is iterating over the collection with an iterator.

Chapter 5. Common mistakes, problems and anti-patterns

This chapter presents some of the common mistakes and problems people face when writing code using Axiom, as well as anti-patterns that should be avoided.

Violating the `javax.activation.DataSource` contract

When working with binary (base64) content, it is sometimes necessary to write a custom `DataSource` implementation to wrap binary data that is available in a different form (and for which Axiom or the Java Activation Framework has no out-of-the-box data source implementation). Data sources are also sometimes (but less frequently) used in conjunction with `OMSourcedElement` and `OMDataSource`.

The documentation of the `DataSource` is very clear on the expected behavior of the `getInputStream` method:

```
/**
 * This method returns an InputStream representing
 * the data and throws the appropriate exception if it can
 * not do so. Note that a new InputStream object must be
 * returned each time this method is called, and the stream must be
 * positioned at the beginning of the data.
 *
 * @return an InputStream
 */
public InputStream getInputStream() throws IOException;
```

A common mistake is to implement the data source in a way that makes `getInputStream` "destructive". Consider the implementation shown in Example 5.1, "`DataSource` implementation that violates the interface contract"[1]. It is clear that this data source can only be read once and that any subsequent call to `getInputStream` will return an already closed input stream.

---

[1]The example shown is actually a simplified version of code that is part of Axis2 1.5 [http://svn.apache.org/repos/asf/webservices/axis2/tags/java/v1.5/modules/kernel/src/org/apache/axis2/builder/unknowncontent/InputStreamDataSource.java].

Example 5.1. `DataSource` implementation that violates the interface contract

```java
public class InputStreamDataSource implements DataSource {
    private final InputStream is;

    public InputStreamDataSource(InputStream is) {
        this.is = is;
    }

    public String getContentType() {
        return "application/octet-stream";
    }

    public InputStream getInputStream() throws IOException {
        return is;
    }

    public String getName() {
        return null;
    }

    public OutputStream getOutputStream() throws IOException {
        throw new UnsupportedOperationException();
    }
}
```

What makes this mistake so vicious is that very likely it will not cause problems immediately. The reason is that Axiom is optimized to read the data only when necessary, which in most cases means only once! However, in some cases it is unavoidable to read the data several times. When that happens, the broken `DataSource` implementation will cause problems that may be extremely hard to debug.

Imagine for example[2] that the implementation shown above is used to produce an MTOM message. At first this will work without any problems because the data source is read only once when serializing the message. If later on the MTOM threshold feature is enabled, the broken implementation will (in the worst case) cause the corresponding MIME parts to be empty or (in the best case) trigger an I/O error because Axiom attempts to read from an already closed stream. The reason for this is that when an MTOM threshold is set, Axiom reads the data source twice: once to determine if its size exceeds the threshold[3] and once during serialization of the message.

Issues that "magically" disappear

Quite frequently users post messages on the Axiom related mailing lists about issues that seem to disappear by "magic" when they try to debug them. The reason why this can happen is simple. As explained earlier, Axiom uses deferred building, but at the same time does its best to hide that from the user, so that he doesn't need to worry about whether the object model has already been built or not. On the other hand, when serializing the object model to XML or when requesting a pull parser (`XMLStreamReader`) from a node, the code paths taken may be radically different depending on whether or not the corresponding part of the tree has already been built. This is especially true when caching is disabled.

While the end result should be the same in all cases, it is also clear that in some circumstances an issue that occurs with an incompletely built tree may disappear if there is something that causes Axiom to build

---

[2]For another example, see http://markmail.org/thread/omx7umk5fnpb6dnc.
[3]To do this, Axiom doesn't read the entire data source, but only reads up to the threshold.

the rest of the object model. What is important to understand is that the "something" may be as trivial as a call to the `toString` method of an `OMNode`! The fact that adding `System.out.println` statements or logging instructions is a common debugging technique then explains why issues sometimes seem to "magically" disappear during debugging.

Finally, it should be noted that inspecting an `OMNode` in a debugger also causes a call to the `toString` method on that object. This means that by just clicking on something in the "Variables" window of your debugger, you may completely change the state of the process that is being debugged!

The OM-inside-OMDataSource anti-pattern

Weak version

`OMDataSource` objects are used in conjunction with `OMSourcedElement` to build Axiom object model instances that contain information items that are represented using a framework or API other than Axiom. Wrapping this "foreign" data in an `OMDataSource` and adding it to the Axiom object model using an `OMSourcedElement` in most cases avoids the conversion of the data to the "native" Axiom object model[4]. The `OMDataSource` contract requires the implementation to support two different ways of providing the data, both relying on StAX:

- The implementation must be able to provide a pull parser (`XMLStreamReader`) from which the infoset can be read.

- The data source must be able to serialize the infoset to an `XMLStreamWriter` (push).

For the consumer of an event based representation of an XML infoset, it is in general easier to work in pull mode. That is the reason why StAX has gained popularity over push based approaches such as SAX. On the other hand for a producer such as an `OMDataSource` implementation, it's exactly the other way round: it is far easier to serialize an infoset to an `XMLStreamWriter` (push) than to build an `XMLStreamReader` from which a consumer can read (pull) events.

Experience indeed shows that the most challenging part in creating an `OMDataSource` implementation is to write the `getReader` method. To avoid that difficulty some implementations simply build an Axiom tree and return the `XMLStreamReader` provided by `OMElement#getXMLStreamReader()`. For example, some ADB (Axis2 Data Binding) versions use the following code[5]:

Example 5.2. `OMDataSource#getReader()` implementation used by ADB

```
public XMLStreamReader getReader() throws XMLStreamException {
    MTOMAwareOMBuilder mtomAwareOMBuilder = new MTOMAwareOMBuilder();
    serialize(mtomAwareOMBuilder);
    return mtomAwareOMBuilder.getOMElement().getXMLStreamReader();
}
```

The `MTOMAwareOMBuilder` class referenced by this code is a special implementation of `XMLStreamWriter` that builds an Axiom tree from the sequence of events send to it. The code than uses this Axiom tree to get the `XMLStreamReader` implementation. While this is a functionally correct implementation of the `getReader` method, it is not a good solution from a performance perspective and also contradicts some of the ideas on which Axiom is based, namely that the object model should only be built when necessary.

---

[4]An exception is when code tries to access the children of the `OMSourcedElement`. In this case, the `OMSourcedElement` will be *expanded*, i.e. the data will be converted to the native Axiom object model.

[5]For the complete code, see http://svn.apache.org/repos/asf/webservices/axis2/tags/java/v1.5/modules/adb/src/org/apache/axis2/databinding/ADBDataSource.java.

Indeed, it should not be necessary to build an intermediary tree when requesting a pull parser from the `OMDataSource` because all the required information is already present in the ADB beans. Worse, if the `OMSourcedElement` is expanded, the object model instance will be built twice: once by the `getReader` and once by Axiom itself!

While constructing an Axiom tree inside the `getReader` method is clearly an anti-pattern, at least in the case of ADB it is not as bad as it seems at first glance. The reason is that in the case which is the most relevant for performance (which is sending a Web Service response prepared using ADB), Axiom will only invoke the `serialize` method and not make use of `getReader`.

> At the time of writing there is no general solution available to avoid the weak version of the OM-inside-OMDataSource anti-pattern in cases where it would be far too difficult to build a proper `XMLStreamReader` implementation. Future versions of Axiom may implement a solution that avoids the complexity of implementing `XMLStreamReader` without too much performance trade-offs.

Strong version

There is also a stronger version of the anti-pattern which consists in implementing the `serialize` method by building an Axiom tree and then serializing the tree to the `XMLStreamWriter`. Except for very special cases, there is **no valid reason whatsoever** to do this! To see why this is so, consider the two possible cases:

1. The `OMDataSource` already implements the `getReader` method in a proper way, i.e. without building an intermediary Axiom tree. To properly implement `serialize`, it is then sufficient to pull the events from the reader returned by a call to `getReader` and copy them to the `XMLStreamReader`. The easiest and most efficient way to do this is using `StreamingOMSerializer`:

Example 5.3. Proper implementation of the `OMDataSource#serialize` method

```
public void serialize(XMLStreamWriter xmlWriter) throws XMLStreamException {
    StreamingOMSerializer serializer = new StreamingOMSerializer();
    serializer.serialize(getReader(), xmlWriter);
}
```

There is thus no need to build an intermediary object model in this case.

2. The `getReader` method also uses an intermediary Axiom tree[6]. In that case it doesn't make sense to use an `OMSourcedElement` in the first place! At least it doesn't make sense if one assumes that in general the `OMSourcedElement` will either be serialized or its content accessed after being added to the tree. Indeed, in this case the Axiom tree will be built at least once (if not multiple times), so that the code might as well use a normal `OMElement`.

This only leaves the very special case where the `OMSourcedElement` is in general neither accessed nor serialized, either because it will usually be somehow discarded or because the code uses `OMDataSourceExt#getObject()` to retrieve the raw data. Even in that case one can argue that in general it should not be too hard to implement at least the `serialize` method properly by transforming the raw or foreign data directly to StAX events written to the `XMLStreamWriter`.

> Implementing the `serialize` method to serialize directly to an `XMLStreamWriter` instead of using an intermediary Axiom tree of course still leaves the question about the `getReader` method open. Since we are assuming that implementing `getReader` properly would be too complex (otherwise one could use the code shown in Example 5.3,

"Proper implementation of the `OMDataSource#serialize` method" to avoid the OM-inside-OMDataSource anti-pattern entirely), one is forced to use the code shown in Example 5.2, "`OMDataSource#getReader()` implementation used by ADB" (and thus the weaker version of the anti-pattern). However this code depends on the `MTOMAwareOMBuilder` class which is part of `axis2-adb`. In some cases, depending on that library may not be an option. Therefore this class should probably be moved to Axiom.

QED

Chapter 6. Appendix

Program Listing for Build and Serialize

```java
import org.apache.axiom.om.OMElement;
import org.apache.axiom.om.OMXMLBuilderFactory;
import org.apache.axiom.om.OMXMLParserWrapper;

import javax.xml.stream.XMLStreamException;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.InputStream;

public class TestOMBuilder {

    /**
     * Pass the file name as an argument
     * @param args
     */
    public static void main(String[] args) {
        try {
            //create the input stream
            InputStream in = new FileInputStream(args[0]);
            //create the builder
            OMXMLParserWrapper builder = OMXMLBuilderFactory.createOMBuilder(in);
            //get the root element
            OMElement documentElement = builder.getDocumentElement();

            //dump the out put to console with caching
            System.out.println(documentElement.toStringWithConsume());

        } catch (XMLStreamException e) {
            e.printStackTrace();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

Links

For basics in XML

- Developerworks Introduction to XML [http://www-128.ibm.com/developerworks/xml/newto/index.html]

- Introduction to Pull parsing [http://www.bearcave.com/software/java/xml/xmlpull.html]

- Introduction to StAX [http://today.java.net/pub/a/today/2006/07/20/introduction-to-stax.html]

- Fast and Lightweight Object Model for XML [http://www.jaxmag.com/itr/online_artikel/psecom,id,726,nodeid,147.html]

- Get the most out of XML processing with AXIOM [http://www-128.ibm.com/developerworks/library/x-axiom/]

References

Specifications

[XML] *Extensible Markup Language (XML) 1.0 (Fifth Edition) [http://www.w3.org/TR/2008/REC-xml-20081126/]*.
W3C Recommendation. 26 November 2008.