

Axiom Developer Guide

Axiom Developer Guide

2.0.0

Licensed to the Apache Software Foundation (ASF) under one or more contributor license agreements. See the NOTICE file distributed with this work for additional information regarding copyright ownership. The ASF licenses this file to you under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Table of Contents

1. Working with the Axiom source code	1
Importing the Axiom source code into Eclipse	1
Testing	1
Unit test organization	1
Testing Axiom with different StAX implementations	2
2. Design	3
General design principles and goals	3
LifecycleManager design (Axiom 1.3)	3
Issues with the LifecycleManager API in Axiom 1.2.x	3
Cleanup strategy for temporary files	4
3. Release process	6
Release preparation	6
Prerequisites	6
Release	7
Post-release actions	9
References	10
A. Appendix	11
Installing IBM's JDK on Debian Linux	11

Chapter 1. Working with the Axiom source code

Importing the Axiom source code into Eclipse

Use the following steps to import the Axiom source code into Eclipse Photon (4.8.0):

1. Install AJDT using the following update site:

<http://download.eclipse.org/tools/ajdt/48/dev/update>

Only the "AspectJ Development Tools" feature is required.

2. Install Workspace Mechanic using the following update site:

<https://alfsch.github.io/eclipse-updates/workspacemechanic>

3. Import the Axiom sources as "Existing Maven Projects" into a new Eclipse workspace. M2Eclipse will propose to install additional Maven plugin connectors; make sure that you install them all.
4. Configure Workspace Mechanic using the files under `etc/workspacemechanic` and accept the proposed preference changes.

Testing

Unit test organization

Historically, all unit tests were placed in the `axiom-tests` project. One specific problem with this is that since all tests are in a common Maven module which depends on both `axiom-impl` and `axiom-dom`, it is not rare to see DOOM tests that accidentally use the LLOM implementation (which is the default). The project description in `axiom-tests/pom.xml` indicates that it was the intention to split the `axiom-tests` project into several parts and make them part of `axiom-api`, `axiom-impl` and `axiom-dom`. This reorganization is not complete yet¹. For new test cases (or when refactoring existing tests), the following guidelines should be applied:

1. Tests that validate the code in `axiom-api` and that do not require an Axiom implementation to execute should be placed in `axiom-api`. This primarily applies to tests that validate utility classes in `axiom-api`.
2. The code of unit tests that apply to all Axiom implementations and that check conformance to the specifications of the Axiom API should be added to `axiom-api` and executed in `axiom-impl` and `axiom-dom`. Currently, the recommended way is to create a base class in `axiom-api` (with suffix `TestBase`) and to create subclasses in `axiom-impl` and `axiom-dom`. This makes sure that the DOOM tests never accidentally use LLOM (because `axiom-impl` is not a dependency of `axiom-dom`).
3. Tests that check integration with other libraries should be placed in `axiom-integration`. Note that this is the only module that requires Java 1.5 (so that e.g. integration with JAXB2 can be tested).

¹See AXIOM-311 [<https://issues.apache.org/jira/browse/AXIOM-311>].

4. Tests related to code in `axiom-api` and requiring an Axiom implementation to execute, but that don't fall into category 2 should stay in `axiom-tests`.

Testing Axiom with different StAX implementations

The following StAX implementations are available to test compatibility with Axiom:

Woodstox

This is the StAX implementation that Axiom uses by default.

Sun Java Streaming XML Parser (SJSXP)

This implementation is available as Maven artifact `com.sun.xml.stream:sjsxp:1.0.1`.

StAX Reference Implementation

The reference implementation was written by BEA and is available as Maven artifact `stax:stax:1.2.0`. The homepage is <http://stax.codehaus.org/Home>. Note that the JAR doesn't contain the necessary files to enable service discovery. Geronimo's implementation of the StAX API library will not be able to locate the reference implementation unless the following system properties are set:

```
javax.xml.stream.XMLInputFactory=com.bea.xml.stream.MXParserFactory
javax.xml.stream.XMLOutputFactory=com.bea.xml.stream.XMLOutputFactoryBase
```

XL XP-J

“XL XML Processor for Java” is IBM's implementation of StAX 1.0 and is part of IBM's JRE/JDK v6. Note that due to an agreement between IBM and Sun, IBM's Java implementation for the Windows platform is not available as a separate download, but only bundled with another IBM product, e.g. WebSphere Application Server for Developers [<http://www.ibm.com/developerworks/downloads/ws/wasdevelopers/>].

On the other hand, the JDK for Linux can be downloaded as a separate package from the developerWorks site [<https://www.ibm.com/developerworks/java/jdk/linux/download.html>]. There are versions for 32-bit x86 (“xSeries”) and 64-bit AMD. They are available as RPMs and tarballs. To install the JDK properly on a Debian based system (including Ubuntu), follow the instructions given in the section called “Installing IBM's JDK on Debian Linux”.

Chapter 2. Design

General design principles and goals

Consistent serialization. Axiom supports multiple methods and APIs to serialize an object model to XML or to transform it to another (non Axiom) representation. This includes serialization to byte or character streams, transformation to StAX in push mode (i.e. writing to an `XMLStreamWriter`) or pull mode (i.e. reading from an `XMLStreamReader`), as well as transformation to SAX. The representations produced by these different methods should be consistent with each other. If a given use case can be implemented using more than one of these methods, then the end result should be the same, whichever method is chosen.

AXIOM-430 [<https://issues.apache.org/jira/browse/AXIOM-430>] provides an example where this principle was not respected.

It should be noted that this principle can obviously only be respected within the limits imposed by a given API. E.g. if a given API has limited support for DTDs, then a `DOCTYPE` declaration may be skipped when that API is used.

LifecycleManager design (Axiom 1.3)

The `LifecycleManager` API is used by the MIME handling code in Axiom to manage the temporary files that are used to buffer the content of attachment parts. The `LifecycleManager` implementation is responsible to track the temporary files that have been created and to ensure that they are deleted when they are no longer used. In Axiom 1.2.x, this API has multiple issues and a redesign is required for Axiom 1.3.

Issues with the LifecycleManager API in Axiom 1.2.x

1. Temporary files that are not cleaned up explicitly by application code will only be removed when the JVM stops (`LifecycleManagerImpl` registers a shutdown hook and maintains a list of files that need to be deleted when the JVM exits). This means that temporary files may pile up, causing the file system to fill.
2. `LifecycleManager` also has a method `deleteOnTimeInterval` that deletes a file after some specified time interval. However, the implementation creates a new thread for each invocation of that method, which is generally not acceptable in high performance use cases.
3. One of the stated design goals (see AXIOM-192 [<https://issues.apache.org/jira/browse/AXIOM-192>]) of the `LifecycleManager` API was to wrap the files in `FileAccessor` objects to “keep track of activity that occurs on the files”. However, as pointed out in AXIOM-185 [<https://issues.apache.org/jira/browse/AXIOM-185>], since `FileAccessor` has a method that returns the corresponding `File` object, this goal has not been reached.
4. As noted in AXIOM-382 [<https://issues.apache.org/jira/browse/AXIOM-382>], the fact that `LifecycleManagerImpl` registers a shutdown hook which is never unregistered causes a class loader leak in J2EE environments.
5. In an attempt to work around the issues related to `LifecycleManager` (in particular the first item above), AXIOM-185 [<https://issues.apache.org/jira/browse/AXIOM-185>] introduced another class called `AttachmentCacheMonitor` that implements a timer based mechanism to clean up temporary files. However, this change causes other issues:

- The existence of this API has a negative impact on Axiom's architectural integrity because it has functionality that overlaps with `LifecycleManager`. This means that we now have two completely separate APIs that are expected to serve the same purpose, but none of them addresses the problem properly.
- `AttachmentCacheMonitor` automatically creates a timer, but there is no way to stop that timer. This means that this API can only be used if Axiom is integrated into the container, but not when it is deployed with an application.

Fortunately, that change was only meant as a workaround to solve a particular issue in WebSphere (see APAR PK91497 [<http://www-01.ibm.com/support/docview.wss?rs=180&uid=swg1PK91497>]), and once the `LifecycleManager` API is redesigned to solve that issue, `AttachmentCacheMonitor` no longer has a reason to exist.

6. `LifecycleManager` is an abstract API (interface), but refers to `FileAccessor` which is placed in an `impl` package.
7. `FileAccessor` uses the `MessagingException` class from `JavaMail`, although Axiom no longer relies on this API to parse or create MIME messages.

Cleanup strategy for temporary files

As pointed out in the previous section, one of the primary problems with the `LifecycleManager` API in Axiom 1.2.x is that temporary files that are not cleaned up explicitly by application code (e.g. using the `purgeDataSource` method defined by `DataHandlerExt`) are only removed when the JVM exits. A timer based strategy that deletes temporary file after a given time interval (as proposed by `AttachmentCacheMonitor`) is not reliable because in some use cases, application code may keep a reference to the attachment part for a long time before accessing it again.

The only reliable strategy is to take advantage of finalization, i.e. to rely on the garbage collector to trigger the deletion of temporary files that are no longer used. For this to work the design of the API (and its default implementation) must satisfy the following two conditions:

1. All access to the underlying file must be strictly encapsulated, so that the file is only accessible as long as there is a strong reference to the object that encapsulates the file access. This is necessary to ensure that the file can be safely deleted once there is no longer a strong reference and the object is garbage collected.
2. Java guarantees that the finalizer is invoked before the instance is garbage collected. However, instances are not necessarily garbage collected before the JVM exits, and in that case the finalizer is never invoked. Therefore, the implementation must delete all existing temporary files when the JVM exits. The API design should also take into account that some implementations of the `LifecycleManager` API may want to trigger this cleanup before the JVM exits, e.g. when the J2EE application in which Axiom is deployed is stopped.

The first condition can be satisfied by redesigning the `FileAccessor` such that it never leaks the name of the file it represents (neither as a `String` nor a `File` object). This in turn means that the `CachedFileDataSource` class must be removed from the Axiom API. In addition, the `getInputStream` method defined by `FileAccessor` must no longer return a simple `FileInputStream` instance, but must use a wrapper that keeps a strong reference to the `FileAccessor`, so that the `FileAccessor` can't be garbage collected while the input stream is still in use.

To satisfy the second condition, one may want to use `File#deleteOnExit`. However, this method causes a native memory leak, especially when used with temporary files, which are expected to

have unique names (see bug 4513817 [http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4513817]). Therefore this can only be implemented using a shutdown hook. However, a shutdown hook will cause a class loader leak if it is used improperly, e.g. if it is registered by an application deployed into a J2EE container and not unregistered when that application is stopped. For this particular case, it is possible to create a special `LifecycleManager` implementation, but for this to work, the lifecycle of this type of `LifecycleManager` must be bound to the lifecycle of the application, e.g. using a `ServletContextListener`. This is not always possible and this approach is therefore not suitable for the default `LifecycleManager` implementation.

To avoid the class loader leak, the default `LifecycleManager` implementation should register the shutdown hook when the first temporary file is registered and automatically unregister the shutdown hook again when there are no more temporary files. This implies that the shutdown hook is repeatedly registered and unregistered. However, since these are relatively cheap operations¹, this should not be a concern.

An additional complication is that when the shutdown hook is executed, the temporary files may still be in use. This contrasts with the finalizer case where encapsulation guarantees that the file is no longer in use. This situation doesn't cause an issue on Unix platforms (where it is possible to delete a file while it is still open), but needs to be handled properly on Windows. This can only be achieved if the `FileAccessor` keeps track of created streams, so that it can forcibly close the underlying `FileInputStream` objects.

¹Since the JRE typically uses an `IdentityHashMap` to store shutdown hooks, the only overhead is caused by Java 2 security checks and synchronization.

Chapter 3. Release process

Release preparation

The following items should be checked before starting the release process:

- Check that the generated Javadoc contains the appropriate set of packages, i.e. only the public API. This excludes classes from `axiom-impl` and `axiom-dom` as well as classes related to unit tests.
- Check that all dependencies and plugins are available from standard repositories. To do this, clean the local repository and execute **`mvn clean install`** followed by **`mvn site`**.
- Check that the set of license files in the `legal` directory is complete and accurate (by checking that in the binary distribution, there is a license file for every third party JAR in the `lib` folder).
- Check that the Maven site conforms to the latest version of the Apache Project Branding Guidelines [<http://apache.org/foundation/marks/pmcs>].
- Check that the `apache-release` profile can be executed properly. To do this, issue the following command:

```
mvn clean install -Papache-release -DskipTests=true
```

You may also execute a dry run of the release process:

```
mvn release:prepare -DdryRun=true
```

After this, you need to clean up using the following command:

```
mvn release:clean
```

- Check that the Maven site can be generated and deployed successfully, and that it has the expected content.
- Complete the release note (`src/site/markdown/release-notes/version.md`). It should include a description of the major changes in the release as well as a list of resolved JIRA issues.

Prerequisites

The following things are required to perform the actual release:

- A PGP key that conforms to the requirement for Apache release signing [<http://www.apache.org/dev/release-signing.html>]. To make the release process easier, the passphrase for the code signing key should be configured in `${user.home}/.m2/settings.xml`:

```
<settings>
...
<profiles>
  <profile>
    <id>apache-release</id>
    <properties>
      <pgp.passphrase><!-- KEY PASSPHRASE --></pgp.passphrase>
    </properties>
  </profile>
</profiles>
```

```
    </profile>
  </profiles>
  ...
</settings>
```

- The release process uses a Nexus staging repository. Every committer should have access to the corresponding staging profile in Nexus. To validate this, login to `repository.apache.org` and check that you can see the `org.apache.ws` staging profile. The credentials used to deploy to Nexus should be added to `settings.xml`:

```
<servers>
  ...
  <server>
    <id>apache.releases.https</id>
    <username><!-- ASF username --></username>
    <password><!-- ASF LDAP password --></password>
  </server>
  ...
</servers>
```

Release

In order to prepare the release artifacts for vote, execute the following steps:

1. If necessary, update the copyright date in the top level NOTICE file.
2. Start the release process with the following command - use 'mvn release:rollback' to undo and be aware that in the main pom.xml there is an apache parent that defines some plugin versions. See <https://maven.apache.org/pom/asf>

```
mvn release:prepare
```

When asked for the "SCM release tag or label", keep the default value (x.y.z).

The above command will create a tag in Subversion and increment the version number of the trunk to the next development version. It will also create a `release.properties` file that will be used in the next step.

3. Perform the release using the following command:

```
mvn release:perform
```

This will upload the release artifacts to the Nexus staging repository.

4. Log in to the Nexus repository (<https://repository.apache.org/>) and close the staging repository. The name of the staging profile is `org.apache.ws`. See <https://maven.apache.org/developers/release/maven-project-release-procedure.html> for a more thorough description of this step.
5. Execute the `target/checkout/etc/dist.py` script to upload the source and binary distributions to the development area of the <https://dist.apache.org/repos/dist/> repository.

If not yet done, export your public key and append it to <https://dist.apache.org/repos/dist/release/ws/axiom/KEYS>. The command to export a public key is as follows:

```
gpg --armor --export key_id
```

If you have multiple keys, you can define a `~/.gnupg/gpg.conf` file for a default. Note that while `'gpg --list-keys'` will show your public keys, using maven-release-plugin with the command `'release:perform'` below requires `'gpg --list-secret-keys'` to have a valid entry that matches your public key, in order to create `'asc'` files that are used to verify the release artifacts. `'release:prepare'` creates the sha512 checksum files.

The created artifacts i.e. zip files can be checked with, for example, `sha512sum 'axiom-2.0.0-bin.zip'` which should match the generated sha512 files. In that example, use `'gpg --verify axiom-2.0.0-bin.zip.asc axiom-2.0.0-bin.zip'` to verify the artifacts were signed correctly.

6. Delete `https://svn.apache.org/repos/asf/webservices/website/axiom-staging/` if it exists. Create a new staging area for the site:

```
svn copy \  
https://svn.apache.org/repos/asf/webservices/website/axiom \  
https://svn.apache.org/repos/asf/webservices/website/axiom-staging
```



This step can be skipped if the staging area has already been created earlier (e.g. to test a snapshot version of the site).

7. Change to the `target/checkout` directory and prepare the site using the following commands:

```
mvn site-deploy  
mvn scm-publish:publish-scm -Dscmpublish.skipCheckin=true
```

The staging area will be checked out to `target/scmpublish-checkout` (relative to `target/checkout`). Do a sanity check on the changes and then commit them.

8. Start the release vote by sending a mail to `dev@ws.apache.org`. The mail should mention the following things:
 - The list of issues solved in the release (by linking to the relevant JIRA view).
 - The location of the Nexus staging repository.
 - The link to the source and binary distributions: `https://dist.apache.org/repos/dist/dev/ws/axiom/version`.
 - A link to the preview of the Maven site: `http://ws.apache.org/axiom-staging/`.

If the vote passes, execute the following steps:

1. Promote the artifacts in the staging repository. See `https://central.sonatype.org/publish/release/#close-and-drop-or-release-your-staging-repository` for detailed instructions for this step.
2. Publish the distributions:

```
svn mv https://dist.apache.org/repos/dist/dev/ws/axiom/version \  
https://dist.apache.org/repos/dist/release/ws/axiom/
```

`version` is the release version, e.g. `1.2.9`.

3. Publish the site:

```
svn co --depth=immediates https://svn.apache.org/repos/asf/webservices/website
```

```
cd ws-site
svn rm axiom
svn mv axiom-staging axiom
svn commit
```

It may take several hours before all the updates have been synchronized to the relevant ASF systems. Before proceeding, check that

- the Maven artifacts for the release are available from the Maven central repository;
- the Maven site has been synchronized to <http://ws.apache.org/axiom/>;
- the binary and source distributions can be downloaded from <http://ws.apache.org/axiom/download.html>.

Once everything is in place, send announcements to users@ws.apache.org and announce@apache.org. Since the two lists have different conventions, audiences and moderation policies, to send the announcement separately to the two lists.

Sample announcement:

Apache Axiom Team is pleased to announce the release of Axiom x.y.z. The release is available for download at:

<http://ws.apache.org/axiom/download.cgi>

Apache Axiom is a StAX-based, XML Infoset compliant object model which supports on-demand building of the object tree. It supports a novel "pull-through" model which allows one to turn off the tree building and directly access the underlying pull event stream. It also has built in support for XML Optimized Packaging (XOP) and MTOM, the combination of which allows XML to carry binary data efficiently and in a transparent manner. The combination of these is an easy to use API with a very high performant architecture!

Developed as part of Apache Axis2, Apache Axiom is the core of Apache Axis2. However, it is a pure standalone XML Infoset model with novel features and can be used independently of Apache Axis2.

Highlights in this release:

- ...
- ...

Resolved JIRA issues:

- [WSCOMMONS-513] Behavior of `insertSiblingAfter` and `insertSiblingBefore` is not well defined for orphan nodes
- [WSCOMMONS-488] The sequence of events produced by `OMStAXWrapper` with `inlineMTOM=false` is inconsistent

For users@ws.apache.org, the subject ("Axiom x.y.z released") should be prefixed with "[ANN][Axiom]", while for announce@apache.org "[ANN]" is enough. Note that mail to announce@apache.org must be sent from an `apache.org` address.

Post-release actions

- Update the DOAP file (see `etc/axiom.rdf`) and add a new entry for the release.

- Update the status of the release version in the AXIOM project in JIRA.
- Remove archived releases from <https://dist.apache.org/repos/dist/release/ws/axiom/>.

References

The following documents are useful when preparing and executing the release:

- ASF Source Header and Copyright Notice Policy [<http://www.apache.org/legal/src-headers.html>]
- Apache Project Branding Guidelines [<http://apache.org/foundation/marks/pmcs>]
- DOAP Files [<http://projects.apache.org/doap.html>]
- Publishing Releases [<http://www.apache.org/dev/release-publishing.html>]

Appendix A. Appendix

Installing IBM's JDK on Debian Linux

1. Make sure that `fakeroot` and `java-package` are installed:

```
# apt-get install fakeroot java-package
```

2. Download the `.tgz` version of the JDK from <http://www.ibm.com/developerworks/java/jdk/linux/download.html>.
3. Edit `/usr/share/java-package/ibm-j2sdk.sh` and (if necessary) add an entry for the particular version of the IBM JDK downloaded in the previous step.
4. Build a Debian package from the tarball:

```
$ fakeroot make-jpkg xxxx.tgz
```

5. Install the Debian package.