

OSTwoUsrManJavaBB

Before you Begin

Before you being using the library there are some tips and conventions that you should be aware of.

Basic Object Hierarchy

It is important to understand the basic hierarchy of fundamental objects within OpenSAML prior to reading this manual so that you understand the difference between, for example, an `XMLObject` and a `SAMLObject`. So, here are the basics:

- **XMLObject** is the most basic object in the system. It represents an XML Element turned into a Java object.
- **ValidatingXMLObject** is a specialization of `XMLObject` that adds support for attaching various validation handlers (discussed later)
- **SAMLObject** is a specialization of `ValidatingXMLObject` that denotes SAML XML Elements that have been turned into Java objects.

So, in this documentation statements about `XMLObjects` apply to all `SAMLObject`s but the reverse is not true.

- **XMLObjectBuilder** is the most basic builder interface for `XMLObjects`
- **SAMLObjectBuilder** is a specialization of `XMLObjectBuilder` that only creates `SAMLObject`s and adds a default, no-argument, builder method applicable to nearly all `SAMLObject`s.

XML Parsing

Parsing XML is an expensive operation. Some of this expense can be mitigated by pooling parsers. OpenSAML provides this functionality through its `org.opensaml.xml.parse.ParserPool` class. It is strongly recommended that developers create instances of this class and use the cached `JAXP DocumentBuilder` that it provides to perform XML parsing. Separate instances representing each unique set of configurations can be created. For example you may wish to create a parser pool that performs schema validation during parsing while another instance does not.

Collection Usage

Java provides a fairly robust set of collection APIs which OpenSAML has chosen to use in a more direct manner than some other projects. Many `XMLObjects` contain a collection of child objects, for example a SAML Assertion may contain many Statements. Instead cluttering the interface for these containing objects with methods to add, remove, clear, iterate over, and get children by index, these `XMLObjects` simply expose the collection of children directly. Again, in the example of Assertions and Statements, the Assertion class exposes a `java.util.List` of statements.

This offers developers the full range of collection semantics when working with this data. You may use a `java.util.Iterator` to traverse or edit the collection, use JDK 1.5 foreach loops, perform mutation operations like add, remove, and clear, and retrieval operations, and the returned collection class will "do the right thing" in respect to adding/removing the objects as children of the contain class. For completeness sake, though, do note that while the returned collection object does implement the specified Java Collection API fully it is not an instance of the collections that ship with the JDK (e.g. `HashSet`, `ArrayList`).

Logging

OpenSAML 2 uses the [SLF4J](#) logging facade. This library is very similar to the Jakarta Commons Logging (JCL) library but differs in two significant ways. First, SLF4J does not attempt to automatically discover and bind to a logging framework implementation. The auto-binding behavior of the JCL can be very problematic if you have more than just a basic classloader hierarchy (a scenario found in every application container currently available). Second, SLF4J allows you to log to the facade before it is even configured (it ships with a very very minimal logging implementation and configuration).

In order enable logging you must either download one of the bindings that bridge SLF4J with another logging system (e.g. Log4J, `java.util.logging`) or provide a logging system that natively implements the SLF4J interfaces (e.g. `logback`). The developers of OpenSAML use the [logback](#) logging framework because it provides some features (e.g. log rotation w/ configurable naming and compression) that don't appear in other logging frameworks. If you wish to use `logback` you need the `logback-core` and `logback-classic` jars from the `logback` distribution.



If you use Jakarta Commons Logging or Log4J as the logging system behind SLF4J you must remove the `jcl104-over-slf4j` or `log4j-over-slf4j`, respectively, jar from the classpath.