Java Product Version Policy

This document sets out the policy used for versioning the Shibboleth Java products and libraries. It defines classifications of releases, indicates how they are identified, and the level of compatibility between them.

Note: these classifications are generally compatible with the definition of semantic versioning when "API" is defined as below.

Terminology

API

The public API of the Java software products and libraries consists of all the public interaction points by which a developer or deployer might interact with the software (as opposed to a more constrained use of the term that refers only to Java classes and methods). This includes:

- Public and protected Java classes and methods exported from **API Packages and Modules** (this is defined subsequently)
- Public and protected Java classes and methods in any 3rd-party library shipped with a product and available on the classpath
- Java or Spring properties defined in product or library reference documentation
- Spring beans defined by name in product or library reference documentation
- Any configuration files and content not covered above and designated as user-modifiable by product documentation
- Documented parameters of command line scripts, utilities, installation tools, etc.
- The default address/location of remotely accessible functionality
- Data storage and logging formats that are documented

This is a non-exhaustive list intended to capture the spirit of "all the public interaction points" and is deliberately meant broadly, not narrowly. This is a constraining definition on our ability to change behavior easily as a trade-off to guaranteeing a high degree of stability and predictability.

API Packages and Modules

There are two types of Java code organization found in our software: single- and multi-module. The terminology refers to the Maven project organization used, and the number of jars produced by the build.

For both single-module and multi-module libraries, **any** public class (and its public and protected methods) should be considered part of the API unless the class's package name contains a segment named "impl". For example, a public class with a qualified name of "net.shibboleth.utilities.java.support.security.Foo" is part of the API. A public class with a qualified name of "net.shibboleth.utilities.java.support.security.impl.Foo" is **not** part of the API. The unqualified name of the class itself is not a consideration, though it would be highly unusual to see an API class called "FooImpl".

Multi-module libraries are now more common for the "major" software products, and result in a number of jars that collectively contain the code. In such cases, the modules are sometimes given a designation of "API" or "Implementation" This is deducible from the name of the module (and the name of the jar file), such that modules containing "-api" in the name are API modules and modules containing "-impl" in the name are Implementation modules. However, we follow the rule outlined above, and so all packages in an "-impl" module will contain "impl" in the name and no packages in an "-api" module will.

Java Version Compatibility

The version policy addresses compatibility at the Java layer in several dimensions:

- Java Source and Target Level – the compiler source and target levels controlling the version of the language used and the version of the bytecode produced, also dictating the minimum version able to run the software
- Compatible Java Versions – the versions of Java deemed compatible[1] with the software
- Supported Java Versions – the versions of Java supported[2] to run the software

[1]The term "compatible" here refers to the commitment by the project to ensure compatibility through testing and to treat any discovered incompatibilities as bugs.

[2]The term "supported" here refers to the commitment by the project to address, research, and debug problems that arise when using a supported version.

Version Numbers

The Shibboleth Java products use a standard triplet format for their version designators: `MAJOR.MINOR.PATCH` where each component is an integer and is separated by a "." (period).

Version Types

We define three types of versions:

- **Patch version** - A software release that addresses bugs in the previous version. Patch versions are denoted by incrementing the `PATCH` component of the `MAJOR.MINOR.PATCH` version identifier.
- **Minor version** - A software release that addresses bugs and adds, but does not remove, functionality. Minor versions are denoted by incrementing the `MINOR` component of the `MAJOR.MINOR.PATCH` version identifier.
- **Major version** - A software release that makes significant changes and may add or remove functionality. Typically but not exclusively, removal of features occurs only after some period of deprecation in earlier releases. Major versions are denoted by incrementing the `MAJOR` component of the `MAJOR.MINOR.PATCH` version identifier.

Plugin Compatibility

As a general rule, the policy is designed to guarantee compatibility for plugins and extensions developed for use with a given minor version with all subsequent patch and minor upgrades until the next major upgrade. Of course, a major upgrade doesn't guarantee breakage, but neither can compatibility be assumed.

Storage Formats

Some functionality may rely on persistent or transient storage, whether in the form of database schema, memcache usage, cookies, or HTML Local Storage. In such cases, we may choose to document these formats and denote them as a stable API, or we may not. If not explicitly designated as an API, storage formats should be assumed to be an implementation detail that may change across releases. The specific changes we will make to such formats are outlined further below.

Third-Party Libraries

It bears noting that by definition we have no control over the versioning policies of 3rd party libraries and indeed in practice one finds a lot of variance across projects in how rigorously they address versioning, or indeed whether they consider rapid evolution to be more important than stability. In practice, this tends to affect which libraries we choose to use and we favor the use of libraries that emphasize stability and clear change management, but there aren't always good alternatives.

One reason we have a policy of not updating 3rd party libraries in patch updates is to avoid accidentally introducing API changes (particularly removal) in a patch. If every 3rd party library followed our policy, it would be safe to introduce patch updates to them in our own patch updates, but they don't, and it's generally time consuming to determine whether APIs have changed or not across a large number of libraries.

In specific cases, if we are confident in the dependent project adhering to a consistent versioning policy similar to ours, we may handle that project more in keeping with our own code (such as applying patch updates to it in our own patch updates).

Since it's hard for us to guarantee against API removal in our own minor updates when we do update 3rd party libraries, caution should always be exercised in relying on 3rd party APIs in extensions to our products. We make every effort to prevent such problems, but with large libraries like Spring, we're ultimately at the mercy of them following their own policies.

The general lack of versioning in Java, despite it being entirely necessary and unavoidable in any system reliant on reusable code, is a problem we can only work around to a degree.

Compatibility

The following compatibility rules should not be interpreted as a straitjacket; every policy has exceptions, this one included. Rather, it means that exceptions will require significant justification and offer significant value, and will be clearly documented in the product's release notes.

Patch Version Compatibility

A patch version has the following general compatibility when compared with a previous version with the same minor version number:

- Java Source and Target Level: identical
- Compatible Java Versions: may add/deprecate, but **not** remove, versions
- Supported Java Versions: may add/deprecate, but **not** remove, versions
- Java API: identical
- 3rd Party Libraries: identical unless a library upgrade was required to address a bug affecting use of the product
- Configuration: identical
- Protocol Messages: wire-compatible
- Storage Formats (API): identical
- Storage Formats (non-API): may adjust in ways that are compatible with all patch versions with the same minor version number

The result of this is that upgrading/downgrading from one patch version to another generally will not require any change other than installing the version to be used, though it is advisable **not** to perform downgrades; preserving one's original configuration and restoring it is a better choice.

Minor Version Compatibility

A minor version has the following compatibility when compared with a previous minor version with the same major version number:

- Java Source and Target Level: identical
- Compatible Java Versions: may add/deprecate, but **not** remove, versions
- Supported Java Versions: may add versions or remove deprecated versions
- Java API: may add/deprecate, but **not** remove, APIs
- 3rd Party Libraries: may add/upgrade, but **not** remove, libraries; upgrades to libraries must follow the Java API compatibility rules
- Configuration: may add/deprecate, but **not** remove, options/properties/beans/etc.
- Protocol Messages: may add, but **not** remove, new protocols/options; protocol implementations in common with previous minor versions remain wire-compatible
- Storage Formats (API): may add to, but **not** remove from, formats in ways that would not be expected to break standard tools interacting with the data
- Storage Formats (non-API): may change in ways that render older minor versions incapable of working with the data

The result of this is that upgrading from one minor version to another does not require any change other than installing the version to be used. Existing configuration files will work unchanged, but may need to be modified to take advantage of new features. Downgrading to an older minor version may require removing configuration options introduced in newer versions.

Note that in adding APIs, abstract or interface methods are not added to existing public interfaces or classes to guarantee runtime compatibility of existing extensions.

Major Version Compatibility

A major version has the following compatibility when compared with another major version:

- Java Source and Target Level: may be different
- Compatible Java Versions: may add or remove versions
- Supported Java Versions: may add or remove versions
- Java API: may add/change/remove APIs
- 3rd Party Libraries: may add/upgrade/remove libraries
- Configuration Files: may add/change/remove options/properties/beans, and alter formats
- Protocol Messages: may add/change/remove protocols; protocol implementations in common with previous major versions remain wire-compatible
- Storage Formats (API): may be different
- Storage Formats (non-API): may be different

Example Compatibility Matrix

Original Version	New Version	Compatible?
2.2.3	2.2.4	Yes, compatibility across patch versions is guaranteed.
2.2.3	2.2.1	Yes, compatibility across patch versions is guaranteed.
2.2.3	2.3.1	Yes, compatibility with later minor versions is guaranteed.
2.2.3	2.1.7	Yes, compatibility with prior minor versions is guaranteed. Configuration files may need to be adjusted to remove options that are not available in the older version.
2.2.3	3.0.0	No, compatibility with prior major versions is not guaranteed.
2.2.3	1.4.7	No, compatibility with prior major versions is not guaranteed.