Mastering JOnAS ClassLoaders

JOnAS Team (Guillaume Sauthier)

- -

Copyright © OW2 Consortium 2011

Table of Contents

List of Figures

List of Tables

List of Examples

Chapter 1. Basics of ClassLoading

1.1. ClassLoader

1.1.1. Usage

A `ClassLoader` knows how to load resources (class, images, ...).

Where the `ClassLoader` locates the resources is an implementation detail: it could search on a network drive, a local file, in a jar or in a directory, ...

Additionaly, for Java classes, a `ClassLoader` supports the class definition process: turning a `byte[]` into a `Class<T>` instance.

> **Warning**
>
> Beware of ClassLoaders, behind the interface, how they works can differ a lot from an implementation to another.
>
> Ex: an OSGi™ Classloader has no automatic delegation to parent loader, unlike an `URLClassLoader`.
>
> Understanding the execution chain requires a knowledge of the ClassLoader's internals

1.1.2. Delegation

ClassLoaders delegate loading of resources to other ClassLoaders, under certains circumstances and conditions.

> **Note**
>
> a Class using `java.lang.String` must use the same `Class<String>` definition to be interoperable with other Class (potentially loaded by different loaders).
>
> *Delegation is essential !*

In order to delegate resource loading to other loaders, a ClassLoader defines some relationships. At least, a ClassLoader have a parent loader (Only system loader do not have a parent : it is the root loader). It may (or may not) also have other links to other loaders. Theses links may (or may not) be used to delegate loading of a resource (or a class) to another loader.

> **Warning**
>
> The way the ClassLoader uses theses links to other loaders, including link to the parent loader are depending on the ClassLoader implementation.

1.2. Class

Class is a Java object (`Class<T>`), it is uniquely identified with a `String`, `ClassLoader` couple. The `String` being the Class' name, and the `ClassLoader` being the loader which has effectively loaded the Class (Not necessarily the one used primarily to load the class !).

> **Note**
>
> `ClassLoader.loadClass(String)` may return a `Class<T>` definition that was not loaded by the `ClassLoader` itself but that coms from an "ancestor".

That means that `ClassCastException` may happen between classes having the same name !

> **Note**
>
> Two classes with the same name but loaded by 2 different loaders are incompatibles : they don't have the same definition.

1.3. CLASSPATH
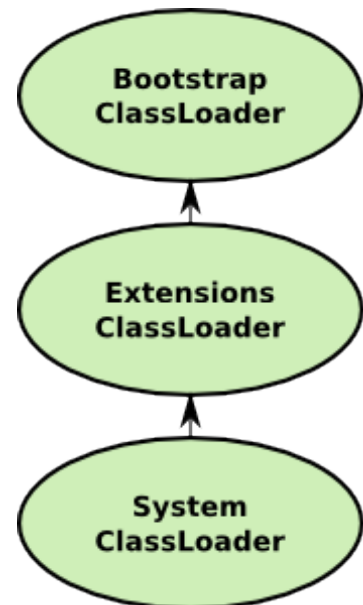
When a Java VM starts 3 loaders are created: `<bootstrap>`, `<extensions>` and `<system>`.

System delegates to Extension, itself delegating to Boostrap, all using a parent first delegation model.

1.3.1. Bootstrap loader

The `<bootstrap>` loader is the primordial VM ClassLoader. It is responsible to load the core Java libraries (`rt.jar`, ...) located in `JAVA_HOME/lib/*.jar`.

The loader has no parent and is implemented with native code.

1.3.2. Extensions loader

The `<extensions>` loader is the only child of `<bootstrap>`, it is responsible to load Java extensions (security, ...) from `JAVA_HOME/lib/ext/*.jar`. The content of this loader can be adapted using the java.ext.dirs system property. This property accepts a comma separated list of paths, all `.jar` files in theses directories will be added in the `<extension>` loader.

This loader is implemented (at least when using Hotspot VM) with `sun.misc.Launcher $ExtClassLoader`.

1.3.3. System loader

The `<system>` loader is the only child of the `<extensions>` loader, it contains the content of the `CLASSPATH` environment variable. It is implemented using `sun.misc.Launcher $AppClassLoader` (at least when using Hotsport VM).

Chapter 2. ClassLoading in JOnAS

2.1. JOnAS Internals (OSGi)

2.1.1. Modular Application Server

JOnAS is a modular application server, it relies on OSGi™ to provide the module layer.

As a consequence, JOnAS is simply an aggregation of Bundles (more or less): a right sized (no more, no less) JOnAS assembly is possible by just choosing the right set of bundles needed by the application.

Figure 2.1. JOnAS modular architecture

2.1.2. OSGi™ ClassLoading

2.1.2.1. Bundle

An OSGi™ `Bundle` is the module unit, it contains classes and resources.It may also contain other jar files (useful for privatizing resources).

Bundle's metadata are defined in `META-INF/MANIFEST.MF`. Theses metadata are providing information that helps to:

- Identify uniquely the Bundle (`Bundle-SymbolicName` + `Bundle-Version`)

- Defines what is published to the outside of the module

- Define module's boundaries

2.1.2.1.1. Bundle ClassPath

A `Bundle` may also have access to resources provided by inner jar files. Theses jar files will form the *Bundle's ClassPath*.

2.1.2.1.2. Exporting Packages

Bundle's metadata may declare exported packages. That means that theses packages (and all contained resources/classes) will be available for other Bundles to use (*imported*).

Theses packages have to be contained in the `Bundle`.

2.1.2.1.3. Importing Packages

Bundle's metadata may also declares imported packages. A wire is created for each imported package that matches a corresponding exported package.

2.1.2.2. Class Space

A class space is a notion associated to a `Bundle`: it represent all the resources accessibles from the bundle. A `Bundle` cannot access a resource outside of its class space.

The figure below show the Class space of Bundle A. This Bundle can access all resources from its own classpath (the Bundle's content) plus all imported packages' resources.

Figure 2.2. Bundle Class Space
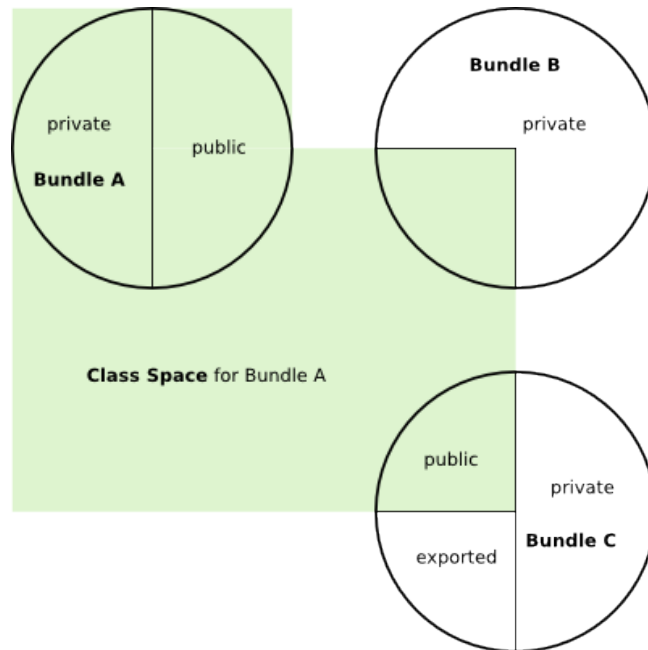
2.1.2.3. Delegation

The OSGi™ specification defines strict classloading rules. Theses rules are applied when a `Bundle` is asked to load a class (or find a resource).

> **Note**
>
> Parent classloader is usually the system Classloader (true for JOnAS), but that may change depending on the underlying OSGi™ framework and its configuration.

This workflow may look complex, but it is well documented and have to be compared with custom loader with unclear behavior (and unspecified delegation rules) ...

This is the price to pay to avoid the well known "Classpath Hell" !

Figure 2.3. OSGi Classloading Algorithm

2.1.2.4. System Bundle

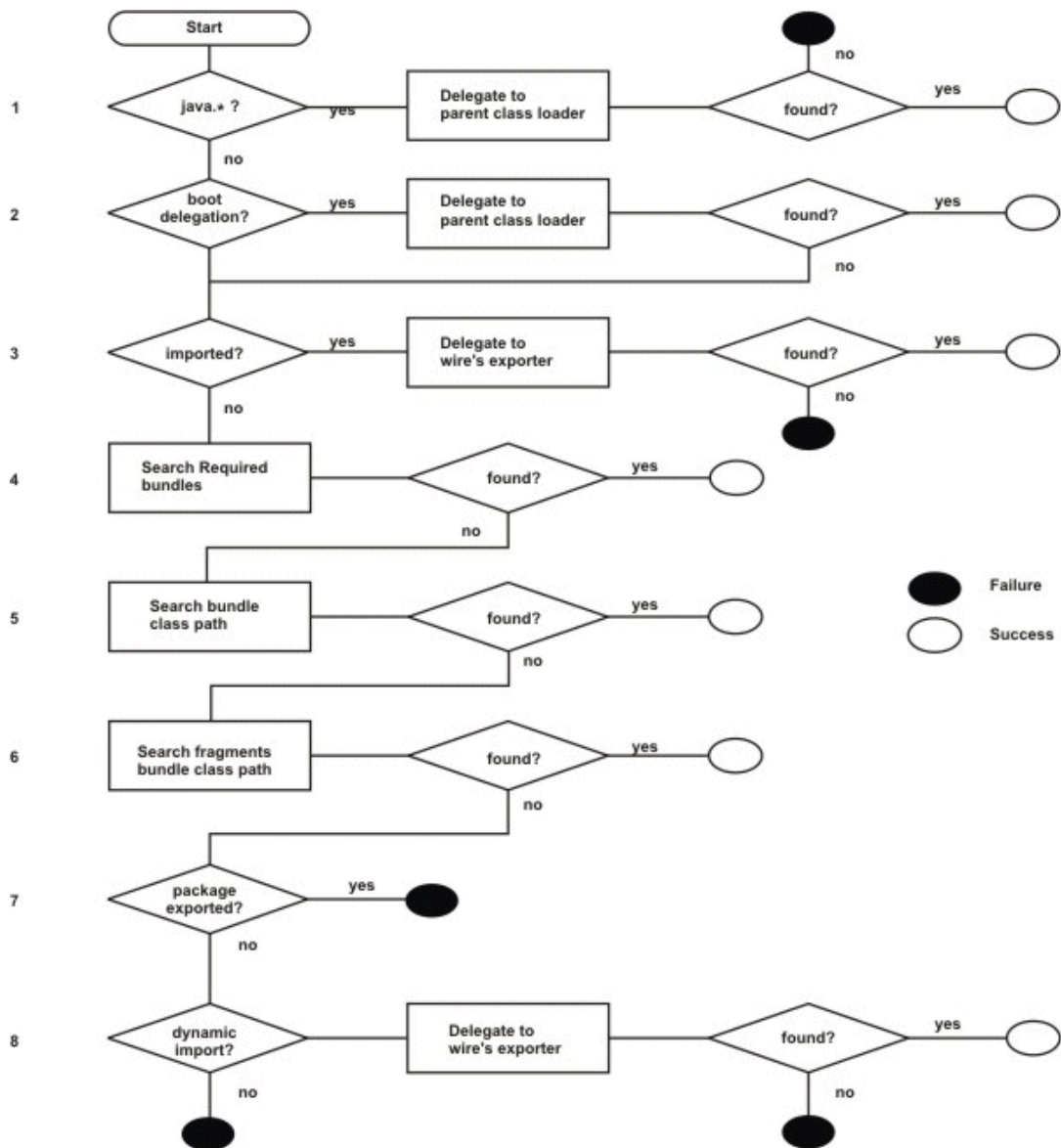The *System Bundle* is a "Wrapper" around the System loaders to make it looks like a `Bundle`: it has a Bundle symbolic name (value: `system.bundle`), is always the first "installed" Bundle (and so it's ID is `0`) and have export packages.

Wrapping the system ClassLoaders as a Bundle is important because other Bundles may imports some packages that are only available in `rt.jar` (and other libraries provided in the System Classloaders - `<bootstrap>`, `<extensions>` and `<system>`). So for the standard package resolution mechanism to work, a `Bundle` representing the system was necessary.

The System Bundle exports some of the system packages: not all, only a selected subset (can be configured). This is some kind of mask or filter allowing to hide some packages.

Hiding a package is then as simple as not exporting it from System Bundle (Ex : javax.transaction because the JVM provides an incomplete package).

2.1.2.5. JOnAS

JOnAS being built on top of OSGi™, it's building blocks are OSGi™ bundles. Each of them having their own `Bundle ClassLoader`, exporting more or less of their content and importing packages from other bundles.

Figure 2.4. JOnAS OSGi™ Modules
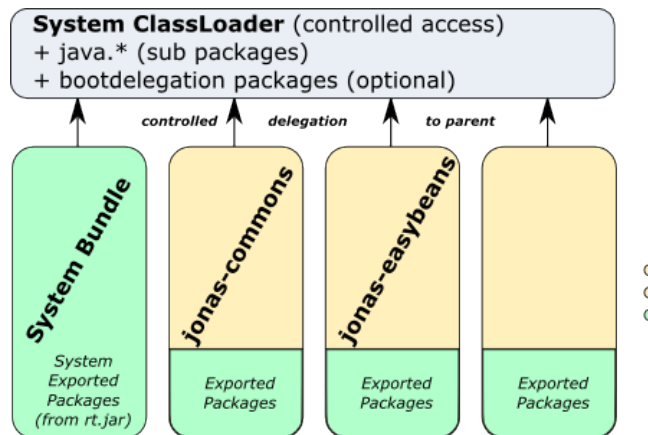
The figure above shows a simplified view of JOnAS Classloading architecture. Each Bundle correspond to a JOnAS' module, green parts represents the subset of module's packages that are exported to the environment. Each Bundle have wire to other Bundles for each resolved imported package (not shown on the picture). All of theses loaders have the System ClassLoader as parent (not to be confound with System Bundle: the all-green module on the left). The usage of this parent loader is very controlled: only java.* packages (java.lang, java.nio, ...) and package's patterns configured in boot delegation are delegated to the parent loader (the two first steps in bundle delegation section).

2.2. Endorsed

Endorsed is a system level (JVM) mechanism allowing to override the classes provided by the System ClassLoaders of the JVM (`rt.jar`, ...).

It is traditionally used in Java EE server to force usage of newer version of some packages/libraries.

> **Note**
>
> In JOnAS, Apache Xerces, Apache Xalan, JAXP APIs plus RMI/IIOP APIs (CORBA) are provided in `${jonas.root}/lib/endorsed/`.

The endorsed mechanism is configurable through a system property: java.endorsed.dirs. the value provides a list of directories (separator : `':'` or `';'` depending on the host Operating System). All jars in theses directories will be inserted before `rt.jar` (thus gaining priority at load-time).

> **Important**
>
> Jars in endorsed directories are inserted in the `<bootstrap>` loader (the very first loader created by the JVM).
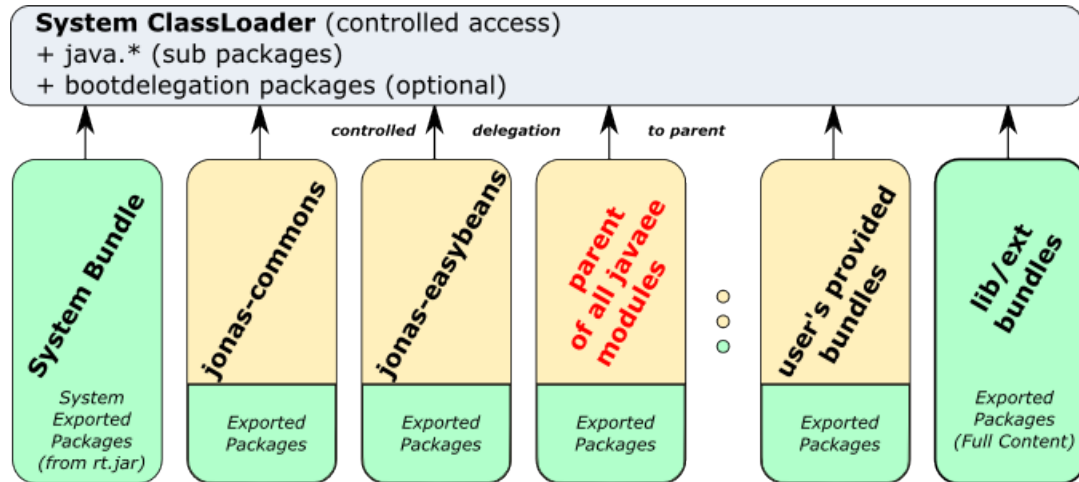>
> They are not auto-magicaly visible to applications.
>
> This is because their visibility is constrained to the list of system exported packages (System Bundle).

2.3. Code Sharing

Sharing additional resources is done by adding bundles to JOnAS. That's it!

Figure 2.5. Sharing Code with OSGi Bundles

2.3.1. Existing Bundles

Existing Bundles have to be deployed using the usual JOnAS deployment mechanisms:

- Deployment directory: Dropping the Bundle in the `${jonas.base}/deploy/` directory is enough

- Command line: **jonas admin add ${path-to}/users-bundle.jar**

- Web console

2.3.2. Jar Files

Jar files (do not have OSGi™ metadata in their manifest) cannot be deployed by Just dropping a jar in `${jonas.base}/deploy/`. They have to be turned into bundle first.

JOnAS offers a fast, easy and efficient mechanism to perform that operation: *Extension Loader*.

This mechanism look for jar files in the `lib/ext/` directories of `${jonas.root}` and `${jonas.base}`. Every jar file (extension: `*.jar`) in these directorier will be transformed into a Bundle (using aQute Bnd [http://www.aqute.biz/Bnd/Bnd]).

Example 2.1. Default set of bnd instructions

```
# Symbolic Name is computed from source jar filename
# A maven-like artifact name is expected <artifactId>-<version>.jar (-version is optional)
# <artifactId> maps to Symbolic Name
# <version> maps to bundle version (if missing 0.0.0 is used)
Bundle-SymbolicName <artifactId>
Bundle-Version      <version>

# Imports all discovered required packages
# Have them marked as optional to avoid startup resolution errors
Import-Package *;resolution:=optional

# Export all the packages contained in the original jar
Export-Package *

# Can load any non-imported package at runtime
DynamicImport-Package *
```

Note

The `lib/ext/` directories are only traversed once: when JOnAS starts. Any update or jar file removal will be ignored until next restart.

Note

Generated Bundles can be found in `${jonas.base}/work/ext-bundles/`

Once they have been generated they could be used as standard Bundles (placed in `deploy/`).

2.4. Java EE Modules

Java EE modules are not Bundles, so they not obey the same classloading rules. This section will explain how woks Java EE classloading in JOnAS.

2.4.1. Overview

Java EE modules can be represented in two groups: standalone modules and embed modules. The classloading hierarchies are different from one case to another.

Figure 2.6. Java EE Modules Classloading Hierarchy
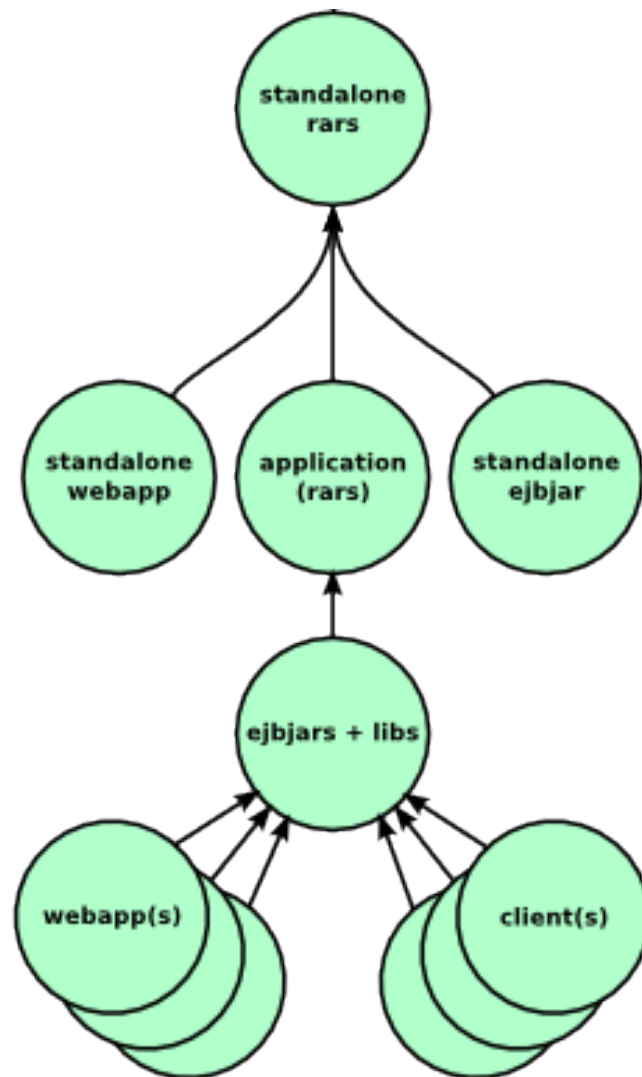
Note

Visibility is from bottom to the top in this schema.

By default, a classloader follow the Java 2 delegation model: asking it's parent first. So any embed WebApp can see the content of the embeds EjbJars, embeds libraries, embed Resource Adapters of its containing Ear. And transitively, it can also access resources provided by standalone Resource adapters. But it will not be able to see resources coming from sibling WebApps and ApplicationClient, niether from standalone EjbJars and WebApps.

2.4.1.1. Standalone modules

Standalone modules means modules primarily deployed on JOnAS (artefact in `deploy/` for example).

- Ears (Java EE Applications) are de facto standalone modules because they cannot be embed in another Java EE module.

- EjbJars and Web Applications are standalone module if deployed outside of an Ear.

- Rars (Resource Adapters) are standalone module if deployed outside of an Ear.

Standalone Rars are specials because they don't have an isolated ClassLoader for each, this unique loader is also the parent of all other standalone modules.

It is necessary for them to share the same loader because Rars contains jars/resources that have to be available to all Java EE deployed modules (thus be accessible in an ancestor of the Java EE loaders).

Note

As Standalone Resource Adapters (not in .ear) are sharing the same ClassLoader, if 2 resources overlaps, the one coming from the earliest deployed Rar is preferred.

Standalone Ears, EjbJars and WebApps are direct childs of this common ancestor: they can load resources/classes from the standalone Rars.

2.4.1.2. Embed modules

Embed modules are all Java EE modules types that are available in Ears:

- EjbJars

- WebApps

- Resource Adapters

2.4.2. Java 2 Delegation Model

The Java2 delegation model is a delegation strategy where the parent loader is asked first, then local sources are probed if resource was not found by parent. This ensure a maximum class space consistence, ensuring a preference for shared sources over local sources.

This model is the default delegation model applied on all JVM provided ClassLoaders (System, `URLClassLoader`, ...).

The Servlet specification states that web applications must run within an "inverted delegation model". In other words : local resources (`WEB-INF/classes/` + `WEB-INF/lib/`) are preferred over server's resources. Standalone Jetty/Tomcat runs apps with this model by default.

On the contrary, the Java EE specification states that the default model must be the normal delegation model.

2.4.3. Web Applications

A WebApp loader can access resource from 3 main locations:

- The System ClassLoader

- `WEB-INF/classes/` and `WEB-INF/lib/*.jar`.

Note

All resources relative to `WEB-INF/classes/` will be available

Note

Only jar files (`*.jar`) directly under `WEB-INF/lib/` are loaded, subdirectories (if any) are not traversed.

For example, a `.properties` file in `WEB-INF/lib/` will not be visible from the ClassLoader.

- Its parent Classloader: if WebApp is standalone, its parent is the Satndalone Rars loader, otherwise it is the EjbJars + Libraries.

2.4.3.1. Delegation Strategy

A WebApp ClassLoader uses the following strategy to load resources/class:

1. Ask to System ClassLoader.

   Note

   Prior to JOnAS 5.1, this step is always executed.

   For JOnAs 5.1, this access is disabled if the execution JVM is superior to Java 6, enabled otherwise (Java 5).

   Post JOnAS 5.2, this access is always disabled (disregarding the Java version)

2. If the delegation model property is true (default):

   a. Ask to the parent.

   b. Ask to local sources.

3. Else (java2 delegation model = false):

   a. Ask to local sources.

   b. Ask to parent.

2.4.4. Ears, EjbJars and Rars

The ClassLoaders for non Web Applications are behaving like traditional ClassLoaders: they use the standard Java2 delegation strategy (parent first).

Chapter 3. Configuration

3.1. Isolating Java EE modules with filters

3.1.1. ClassLoader Filtering

ClassLoaders of Java EE modules can be filtered with JOnAS. Filtering acts as a runtime package mask hiding a specified set of patterns.

When a resource/class is trying to be loaded by a `ClassLoader`, this one first checks if the fully qualified resource name (ie: includes the package name) matches a provided pattern. If so, it cut the execution flow and throw a `ClassnotFoundException` or return `null` when a class (respectively a resource) matches one of the patterns.

ClassLoader Filtering is a fine grained approach to filtering, based on pattern matching (ex: org.springframework.* excludes all resources from the org.springframework packages). It is implemented as an intermediate (pass through) `ClassLoader` seating between 2 other ClassLoaders (ex: Web -> (filtering) -> EjbJars).

Two levels of filters are available: system-wide (shared by all Java EE modules in the system) and per-module (only impacts a given module).
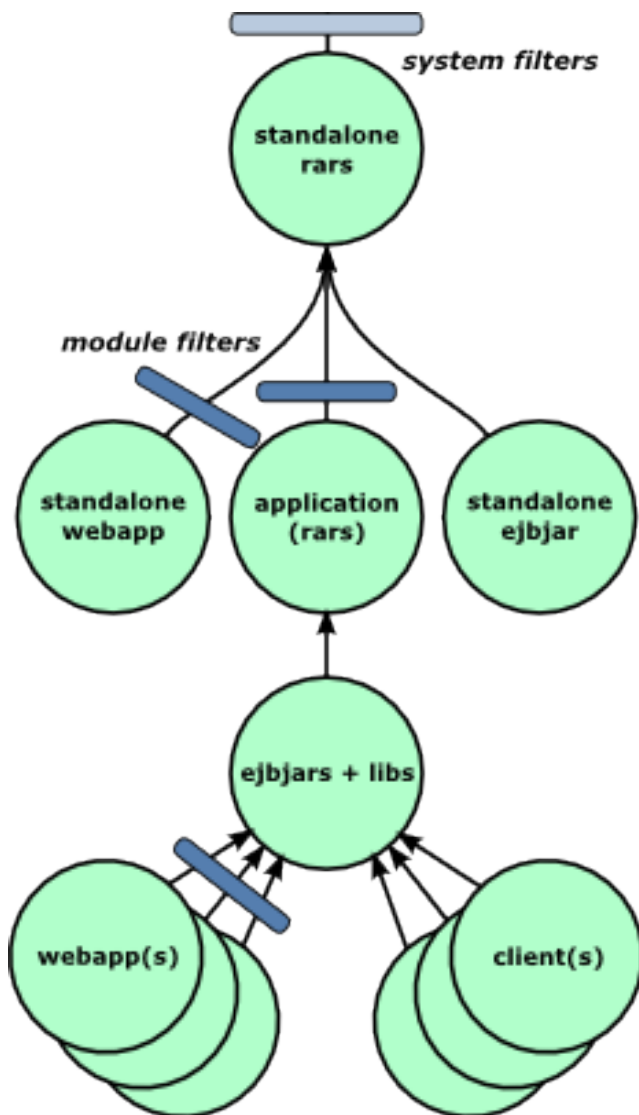
Important

JOnAS 5.1 only offers system wide configuration

It is globally deactivatable using the System property jonas-disable-filtering-class-loader=true

Important

Acts as a barrier between Java EE modules and the application server: isolate modules from resources also available in JOnAS.

Figure 3.1. Filter's position

3.1.1.1. XML Configuration

System wide configuration is applied to the parent loader of applications (the loader of standalone Rars). It is configurable in the `${jonas.base}/conf/classloader-default-filtering.xml` file.

Example 3.1. Default system-wide filtering configuration

```xml
<?xml version="1.0" encoding="UTF-8"?>
<class-loader-filtering xmlns="http://org.ow2.jonas.lib.loader.mapping">

  <!--
    List of filters used to exclude packages/resources that are used
    internally by the Application Server but that will not be available
    to applications.
    An empty list will not hide any packages to the applications
    This list is used both to hide resources and classes to applications.
  -->
  <default-filters>
    <!--
      Filters are using regexp as specified at
      http://java.sun.com/j2se/1.5.0/docs/api/java/lang/
String.html#matches(java.lang.String)
    -->
    <filter-name>org.apache.commons.digester.*</filter-name>
    <filter-name>org.springframework.*</filter-name>
  </default-filters>
</class-loader-filtering>
```

The per-module configuration is available for:

- Webapp (either standalone or embed) with `WEB-INF/classloader-filtering.xml`

- Ear with `META-INF/classloader-filtering.xml`

The configuration style is not applicable for RARs or EjbJars Standalone since they directly uses the system filters. When embed in Ear, they uses filters defined in the application.

Example 3.2. Per-module configuration sample

```xml
<?xml version="1.0" encoding="UTF-8"?>
<class-loader-filtering xmlns="http://org.ow2.jonas.lib.loader.mapping">
  <filters>
    <filter-name>org.apache.xml.*</filter-name>
  </filters>
</class-loader-filtering>
```

3.1.2. Filtering Usage

ClassLoader filtering should be used when JOnAS provides libraries are conflicting with application's embed libraries. That will make sure that application libraries are used instead of application server's ones.

Warning

Make sure that all needed packages are filtered: some libraries aggregates multiple packages.

Ex : Apache Xalan includes org.apache.xalan + org.apache.xml + org.apache.xpath in `Xalan.jar`

3.2. Inverting Java2 delegation model for webapp

As stated in the section related to Java2 delegation model, Java EE and Servlet specifications dictates different default delegation strategy. In order to limit behavioral changes when migrating from a standalone web container to JOnAS, a switch is available through the `WEB-INF/jonas-web.xml`:

- It only applies to web application (included or not in Ear)

- Search ordering is inverted if delegation is disabled

Example 3.3. Invertion of Java2 Delegation Strategy (jonas-web.xml)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<jonas-web-app xmlns="http://www.objectweb.org/jonas/ns"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xsi:schemaLocation="http://www.objectweb.org/jonas/ns
                                    http://jonas.ow2.org/ns/jonas-web-app_4_0.xsd">
  <!-- true : the context uses a classloader using the Java 2 delegation model (default)
       false : the class loader looks inside the web application first, before asking parent
 class loader -->
  <java2-delegation-model>false</java2-delegation-model>
</jonas-web-app>
```

Note

Avoid using `java2-delegation-model`: use filtering instead.

Filtering permits to touch only the required bits while invertion apply for the whole web application.

What happen when a lib needs inversion and another do not ?

Invertion is more *lenient* than filtering, preference over exclusion : can still use server's classes where filtering defines a real barrier.

3.3. Publishing system packages

JOnAS comes with a predefined list of system packages [http://websvn.ow2.org/filedetails.php?repname=jonas&path=%2Fjonas%2Fbranches%2Fjonas_5_2%2Fjonas%2Fmodules%2Ftools%2Flaunchers%2Ffelix-launcher%2Fsrc%2Fmain%2Fresources%2Forg%2Fow2%2Fjonas%2Flauncher%2Ffelix%2Fjavase-profiles.properties] adapted to the runtime JVM version. By default, the System Bundle will exports all of the listed packages.

For some uses cases (endorsed addition, ...), it may be useful to add some packages to this list. This is indeed configurable in JOnAS.

JOnAS 5.1 configuration is provided as a full felix configuration file. This file is located in the `lib/bootstrap/felix-launcher.jar` file (See org/ow2/jonas/launcher/felix/default-config.properties).

Configuration customizers have to get a full copy of that file and perform manual changes in it.

When satisfied, JOnAS must be started with a system property named felix.configuration.file pointing to the modified file's path.

For JOnAS 5.2 (and superior), most of the Felix/OSGi configuration is accessible in `conf/osgi/` folder. It is commented, readable and comprehensive. Most useful properties are easily accessible.

3.3.1. System Packages

The list of System Bundle exported package is expressed using 2 properties : org.osgi.framework.system.packages and org.osgi.framework.system.packages.extra.

The format of theses properties is a comma separated list of package names (ex : `org.osgi.framework;version=1.5.0`), just like a standard `Export-Package` OSGi™ header.

Note

Wildcards are not accepted in this list, all packages (even sub-packages) have to be declared individually.

Theses packages will then be *importable* from other Bundles installed on the OSGi™ gateway.

Tweaking the list of system packages may be used to hide or change some attributes of packages, that let a possibility for other bundles to provide a different version of the exported packages, and leave the choice to the bundle consummer.

That configuration freedom has to be used with care: org.osgi.framework.system.packages property has to be used only when it's needed to remove/modify a package in the system default exported list.

Note

javax.transaction + javax.transaction.xa are not exported by the System Bundle in JOnAS because the JVM misses some of the classes in theses packages

The System extra packages list has to be used when bundles/applications need an additional package from a library placed in endorsed directories (or in any of the place seached for System ClassLoader: extensions + CLASSPATH).

Note

Resolve package constraint resolution error with the addition of a new exported packages declaration to the system.

3.3.2. Boot Delegation

Boot delegation packages is one of the OSGi joker for known non-modular JVM packages. They have priority over all other way of classloading.

It is configurable through the org.osgi.framework.bootdelegation property. The value is a comma separated list of packages patterns: wildcard ('*') is only accepted at the end of the pattern.

Note

`com.sun.image*` : accept classes from com.sun.image and sub-packages

Important

Theses packages do NOT appear in system's exported packages (may lead to unexpected resolution error)

Touching this property should be avoided as much as possible, because it breaks modularity. As per the osgi delegation strategy, bootdelegation have the top most priority in loading, so no overridde is possible: a Bundle cannot use a different version (even if it import it and the import was resolved) !

In clear, it *should be used as last resort*: when adding packages to System Bundle did not worked or when a class in a bundle performs some dynamic class loading and makes the assumption that its own ClassLoader can load the wanted Class. There are multiple condition for this to happen:

• The wanted class is unknown at build time (otherwise a proper import would have been generated)

• The wanted class is in the system ClassLoader (otherwise servicing this class from the System ClassLoader does not make sense)

• The bundle do not import the resource's package

• You cannot change this Bundle :-(

Chapter 4. Tooling

4.1. Web Console: Classloader monitoring

The "Classloader monitoring" module in the jonasAdmin web console aims to help diagnosis of classloading issues.

It provides a graphical view of Java EE classloading hierarchies and gives a direct way to interact with ClassLoaders and see their behaviors.

4.1.1. OSGi Diagnosis

The web console offers a search feature that helps to know from which Bundle a class/package/ resource comes from.It displays the wired consumer Bundle(s).

It helps to see if a Bundle is wired to the right package provider.

Figure 4.1. OSGi Diagnosis

<title>Search for a given class in all the OSGi bundles</title>

4.1.2. Java EE Hierarchies

The web console also offers a Java EE ClassLoading hierarchy view. It is dedicated to a selected Java EE module and show the ancestry of Classloader.

Filtering patterns associated to ClassLoaders are also displayed.

This view permits to test resource loading from the selected ClassLoader. That helps to see if the class comes from the expected source.

Chapter 5. Tips

5.1. Abstract Factory Pattern

5.1.1. Error pattern

In an eye blink, suspect all static `Factory.newInstance()` methods. Period.

The AbstractFactory pattern is a well known and quite used pattern. There is an abstract Factory class with an implemented static method (`newInstance()`). This method is in charge of finding a suitable implementation that will be returned to the consumer. JAXP API are using this pattern.

Usually this method performs the following operations:

1. Try to find the name of a class (the concrete Factory) to load.

   Usually, the search is done in some dedicated property files, system properties, default hardcoded value, ...

2. Try to load the discovered class using a guessed loader.

   Used loaders depends on the code but usually involves (ordre is not significant here):

   • Thread Context ClassLoader

   • ClassLoader of the Factory

   • System ClassLoader

   • A given ClassLoader (could be passed as parameter with some luck)

3. Creates an instance of the loaded Class (if one could be loaded)

> **Important**
>
> This kind of code make the assumption that they can load any class.
>
> This is not true (even completely wrong) in a modular world

5.1.2. Solutions

The environment has to be adapted to what is expected by the code:

• Give an appropriate ClassLoader (if possible)

• Set a Thread Context ClassLoader to a ClassLoader that will be able to load the Class.

> **Note**
>
> Do not forget to reset to old ClassLoader after the call (`try/finally`)!

```
ClassLoader expected = ...
ClassLoader old = Thread.currentThread().getContextClassLoader();
Thread.currentThread.setContextClassLoader(expected);
try {
  // Do whatever you want in this block
  Factory factory = Factory.newInstance();
```

```
} finally {
  // Reset the TCCL
  Thread.currentThread.setContextClassLoader(old);
}
```

5.2. ClassCastException

5.2.1. Error pattern

Conflicting libraries are found in JOnAS and in an application.

Delegation mechanism, and the logic used to find an implementation, produces an instance from a *Class incompatible with expected type*.

For example, the loaded type comes from the System ClassLoader but the expected type comes from the webapp ClassLoader.

5.2.2. Solutions

Use the stacktrace to extract faulty classnames: that's the main suspects.

Use the console to find from where theses classes/packages are loaded.

Display Right/Left ClassLoader of the assignation to discover the ClassLoader sources. -

- Interesting values: Expected type/loader, Returned type/loader, TCCL

- Similar to the info the console provides

Filter the package(s) to force resolution in your application codebase.

Usually go back to first bullet until no more Exceptions .

5.3. Code rules !

To understand what happen, knowing the ClassLoader hierarchies helps a lot but is not always sufficent.

Reading the source code gives the final clues explaining the observed behavior.

> **Note**
>
> Peoples do a lot of things in Java with class loading (some even weird)

5.4. Boot Delegation

Format of boot delegation pattern is somehow sensible. Here are examples and explanations.

- No wildcard (ex: com.sun.xml): Only matches classes directly in the given package

- Dot and wildcard (ex: com.sun.xml.*): Only matches classes in sub packages (direct content does not match)

- Only wildcard (ex: com.sun.xml*): Matches both direct package and sub packages. But also matches for packages sharing the same chars at the beginning of their names.

Table 5.1. Boot Delegation Patterns Examples

Pattern	Match com.sun.xml.Parser ?	Match com.sun.xml.mine.MineParser ?	Match com.sun.xmlaaaaa.AaaaParser ?
com.sun.xml	YES	NO	NO
com.sun.xml.*	NO	YES	NO
com.sun.xml*	YES	YES	YES