Leading Open Source Middleware

Java EE Programmer's Guide

JOnAS Team ( )

- Feb 2008 -

Copyright © OW2 Consortium 2008-2009

This work is licensed under the Creative Commons Attribution-ShareAlike License. To view a copy of this license,visit http://creativecommons.org/licenses/by-sa/2.0/deed.en or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

Table of Contents

.... iv
1. Principles 1
    1.1. Enterprise Bean Creation 1
    1.2. Web Components Creation 1
    1.3. Application Deployer and Administrator 1
    1.4. Java EE Application Assembler 2
2. JOnAS Class Loader 3
    2.1. JOnAS classloading architecture 3
        2.1.1. Understanding class loaders hierarchies 3
        2.1.2. Application Server's class loaders 4
        2.1.3. Java EE module's class loaders 6
        2.1.4. Filtering class loaders 8

List of Figures

2.1. Filtering ClassLoader Principle 9

The target audience for this guide is the application component provider, i.e., the person in charge of developing the software components on the server side (the business tier).

Chapter 1. Principles

JOnAS supports two types of Java EE application components: Enterprise Beans and Web components. In addition to providing guides for construction of application components, guides are supplied for application assembly, deployment, and administration.

1.1. Enterprise Bean Creation

The individual in charge of developing Enterprise Beans should consult the for instructions on how to perform the following tasks:

1. Write the source code for the beans.

2. Specify the deployment descriptor.

3. Bundle the compiled classes and the deployment descriptor into an EJB JAR file.

JOnAS 5 supports both versions 2 and 3 of the Enterprise Java Beans (EJB) specifications. Programming and configuration methods between these two versions have changed drastically, we therefore provide you with two different programmer's guides:

- Enterprise Beans Programmer's Guide, for EJB version 3 (recommended) [ejb3_programmer_guide.html]

- Enterprise Beans Programmer's Guide, for EJB version 2 [ejb2_programmer_guide.html]

1.2. Web Components Creation

Web designers in charge of JSP pages and software developers providing servlets can consult the Web Application Programmer's Guide.

The Developing Web Components [web_pg.html#web.component] guide explains how to construct Web components, as well as how to access Enterprise Beans from within the Web Components.

Deployment descriptor specification is presented in the Defining the Web Deployment Descriptor [web_pg.html#web.deployment] chapter.

Web components can be used as Web application components or as Java EE application components. In both cases, a WAR file will be created, but the content of this file is different in the two situations. In the first case, the WAR contains the Web components and the Enterprise Beans. In the second case, the WAR does not contain the Enterprise Beans. The EJB JAR file containing the Enterprise Beans is packed together with the WAR file containing the Web components, into an EAR file. Principles and tools for providing WAR files are presented in WAR Packaging [web_pg.html#web.packaging] and the Deployment and Installation Guide [deployer_guide.html#deploy.applis].

1.3. Application Deployer and Administrator

JOnAS provides tools for the deployment and administration of Enterprise Beans (EJB JARs), Web applications (WARs), and Java EE applications (EARs).

The Deployment and Installation Guide [deployer_guide.html#deploy.applis] covers issues related to the deployment of application components.

The Administration Guide [administration_guide.html#administration.guide] presents information about how to manage the JOnAS server and the JOnAS services that allow deployment of the different types of application components: EJB Container service, Web Container service, and EAR service.

1.4. Java EE Application Assembler

The application assembler in charge of assembling the application components already bundled in EJB JAR files and WAR files into a Java EE EAR file, can obtain useful information from the Java EE Application Assembler's Guide [eardeploy.html].

Chapter 2. JOnAS Class Loader

2.1. JOnAS classloading architecture

This section introduces the classloading architecture of JOnAS.

2.1.1. Understanding class loaders hierarchies

Obviously, a `ClassLoader` is used to load classes (and resources) that will then be usable from the application. Most of the ClassLoaders are `URLClassLoaders`, that means that they are based on a set of URLs. Theses URLs will be the place where the ClassLoader, when asked for a resource (using its fully qualified name, ie including full package name), will try to find the resource.

An `URLClassLoader` always have a parent ClassLoader. When loading a class, the first thing a ClassLoader do is to ask its parent to load the class (this is recursive). If the parent return the Class, it's returned to the caller. Otherwise, the URLs will be search for the given class. this is the default Java delegation model (parent first).

Let's roll with an example: you have a ClassLoader (let's call it A) that can load the class `Test`, A is the parent of B, but B do not contains the `Test` class, and B is the parent of C, with C that contains also the `Test` class. Clearly, there is a ClassLoader hierarchy here (C->B->A, with "->" meaning "is child of"). If you ask C to load `Test`, the following steps will be performed:

1. C have B as parent, so asks B to try to load `Test`
2. B have A as parent, so asks A to try to load `Test`
3. A is the top level ClassLoader, no parent, it will try to load test by itself
4. As A contains the `Test` class, it can be loaded and returned
5. B see that A (its parent) have loaded the class, so it return it as is
6. C see that B (its parent) have loaded the class, so it return it as is

Note

Using the Java delegation model, the loaded class is always coming from the parent ClassLoader that is closer to the System class loader (the well known `CLASSPATH`), even if the class exists in a lower level ClassLoader (aka child ClassLoaders).

Java EE makes an heavy use of ClassLoaders: the application server in itself may be composed of multiple ClassLoaders and the Java EE applications (EAR, Rar, War, EjbJar) themselves are using ClassLoaders.

An application is deployed by its own class loader. This means, for example, that if a Web Application and an EjbJar are deployed separately, the classes contained in the two archives are loaded with two separate classloaders with no hierarchy between them. Thus, the EJBs from within the JAR will not be visible to the Web components in the WAR. This is not acceptable in cases where the Web components of an application need to reference and use some of the EJBs (this concerns local references in the same JVM).

Note

Class loaders are used to ensure *class space isolation*.

For this reason, prior to EAR files, when a Web application had to be deployed using EJBs, the EjbJar had to be located in the `WEB-INF/lib` directory of the Web application.

Currently, with the Java EE integration and the use of the EAR packaging, that kind of class visibility problems no longer exist and the EjbJar is no longer required in the `WEB-INF/lib` directory.

The following sections describe the JOnAS class loaders and explain the mechanism used to locate the referenced classes.

2.1.2. Application Server's class loaders

JOnAS is basically an assembly of modules providing the features that makes JOnAS an application server. Each of the modules have an associated class loader. All theses loaders have visibility constraints, meaning that some of the internal classes or libraries used by JOnAS are hidden from the application class space. In clear, even if JOnAS may use a library also used by a deployed application, there will be no conflicts.

Note

Having visibility constraints on class loaders reduce the risks of conflicting libraries.

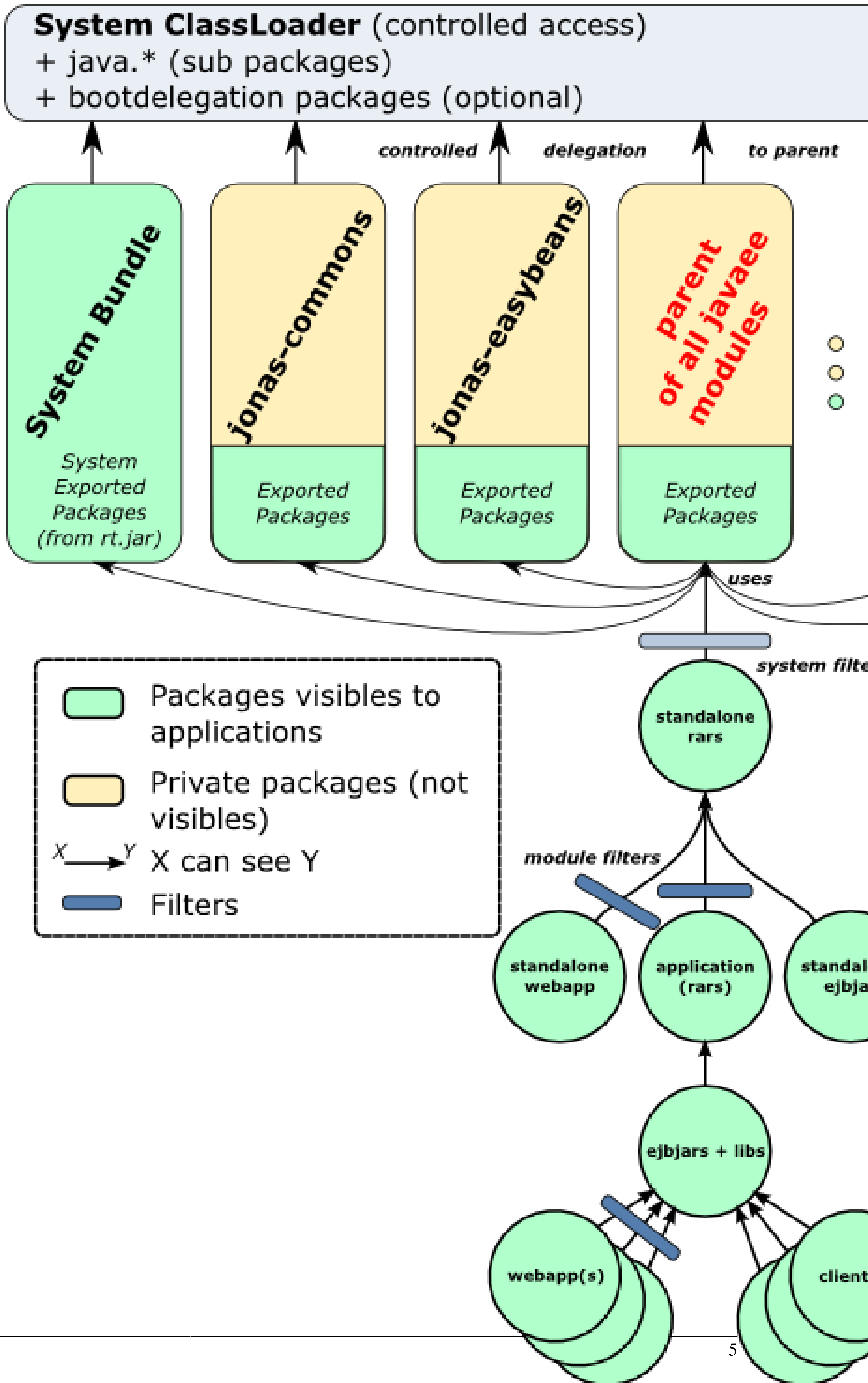
The loaders of JOnAS are not in a tree hierarchy. They all share the same parent class loader (the System classloader), so it's a flat hierarchy. Nevertheless, there are links (aka wires) between theses class loaders. A module can export some of its packages (and thus, all the classes from that package) to the system. Theses exported packages can then be imported by other modules. That creates a wire. As a module can be wired to multiple others modules, the class loaders of the application server forms an oriented graph of loaders.

Note

The class loading rules are described in much more details in the OSGi™ specification.

Note

Only exported packages are available to the applications


System ClassLoader (controlled access)
+ java.* (sub packages)
+ bootdelegation packages (optional)
controlled
delegation
to parent
System Bundle
System Exported Packages (from rt.jar)
jonas-commons
Exported Packages
jonas-easybeans
Exported Packages
parent of all javaee modules
Exported Packages
uses
system filte
standalone rars
module filters
standalone webapp
application (rars)
ejbjars + libs
webapp(s)
Packages visibles to applications
Private packages (not visibles)
X can see Y
Filters

2.1.2.1. Extending JOnAS class loaders

As JOnAS is build on a modular system (OSGi™), it's very natural to "extends" the application server's class space.

There are 2 way that can be used to augment the class space:

1. Place an OSGi™ bundle containing your classes in the `${jonas.base}/deploy/` directory

   It will be deployed as any normal bundle, and if it's resolved (ie all imported packages are found), all the packages exported by this bundle will be made available to the server (and applications).

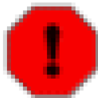
   Note

   This is the preferred way to augment the class space.

2. Place a classic jar file in the `${jonas.base}/lib/ext/` (or `${jonas.root}/lib/ext/`). During JOnAS startup, theses jar files will automatically be transformed into bundle (using default creation rules) and installed like all other bundles.

   Note

   The default rule is: every required packages that is not in the jar file will be imported, and every packages of the jar will be exported.

   Warning

   This option, even if it is seducing, should be used with caution: it can break application server modularity (if they export a package required by some of the JOnAS modules, maybe that can cause weird classes exceptions), and, because of the default transformation rule (all is exported), it's highly possible that a jar file will export some internal packages that should not have been exposed to the world.

   So the right method to augment the class space is to provide a well constructed bundle (with imports and exports constrained with versions).

2.1.2.2. Using JDBC driver jars

In order to connect the applications to an external database (not HSQL which is embedded in JOnAS), the user must provide the jar file that contains the JDBC Driver's classes.

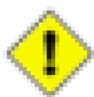
Providing theses jar files are just like extending JOnAS class loaders:

- The driver's jar file is already an OSGi™ bundle, so it can be placed in `${jonas.base}/deploy` and it will be deployed automatically
- The jar file is not a bundle, it could be placed in `${jonas.base}/lib/ext` and will be (*at startup only*) transformed into a bundle (see previous section for detailed information), or the user takes the responsability of transformation (more fine control on the bundle).

2.1.3. Java EE module's class loaders

Java EE modules are following the classic, hierarchical, parent oriented graph of class loaders.

The class loader hierarchy for Java EE modules that allows the deployment of EAR applications without placing the EJB JAR in the `WEB-INF/lib` directory consists of the following:



Caution

Work in progress

2.1.3.1. Standalone resource adapters loader

This class loader, child of the OSGi™ class loader, contains only the libraries provided by the system level resource adapters (i.e. resource adapters not part of an EAR application).

This class loader is the parent of all the deployed Java EE modules:

- Standalone Web Applications (`.war` files)
- Standalone EjbJars (`.jar` files)
- Java EE Applications (`.ear` files)

This means that all the classes available from this loader are also made available for the child loaders.

2.1.3.2. Standalone EjbJar loader

A standalone EjbJar classloader contains the classes packaged inside of the ejbjar. The parent of this classloader is the Section 2.1.3.1, “Standalone resource adapters loader”. It can therefore see all the classes available from the system level resource adapters and its ancestors.

The classloading policy of this loader is to ask its parent first, and then (if the class was not found) try to search in itself. *There is no way to invert the classloading policy of an EjbJar classloader.*

2.1.3.3. Standalone WebApp loader

The classloader for standalone web applications modules contains the classes and libraries packaged inside of the war.

It contains the `WEB-INF/classes/` directory (that usually matches the classes of the application) and all the jar files located under the `WEB-INF/lib/` directory. Notice that only jars directly under the `lib/` directory are available, jars in `WEB-INF/lib/subdirectory/` will not be accessible from the classloader.

The parent of the standalone webapp classloader is the Section 2.1.3.1, “Standalone resource adapters loader”, meaning that all standalone web applications will be able to use system level resource adapters classes/libraries and the classes available from its ancestors.

A webapp loader also have a direct link with the system classloader (classes of the JVM + what's available from the `CLASSPATH` environment variable). *All class loading actions are first directed to the system classloader* to avoid JVM class overloading from the web application (in short, guards the developer from overloading primordial classes).

Then, if the system classlaoder did not find the required class, the loader will ask either the parent, or itself. Asking the parent or itself is configured by the delegation model choosed by the developer.

Note

By default, java2 delegation model is turned on.

2.1.3.3.1. Java 2 delegation model

The default classloading policy of a web application in an application server (in opposition of a webapp hosted in a standalone servlet container) is to ask the parent loader first, and then look inside itself (if the parent did not find the required class), this is the default java2 delegation model.

Nevertheless, a web application developer can configure the web classloader to use first itself, then ask the parent loader. That means that the web application classes will have more priority than classes from the parent loader, in other words it can be used to overload some classes provided by the application server.

The delegation model can be configured using the `WEB-INF/jonas-web.xml` specific deployment descriptor, as shown below.

```
<?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>
```

2.1.3.4. EAR class loader

The EAR class loader is the common parent (directly or indirectly) of all the class loaders of the inner modules (ejbjars, rar, webapps). There is only one EAR class loader per EAR application. This class loader is the child class loader, thus making JOnAS classes visible to it. The parent of all the EAR class loaders is a JOnAS special class loader that is able to seek for classes/resources exported by all the other module's class loaders of JOnAS.

2.1.3.4.1. Class-Path: entries and lib/folder

In Java EE 5, if libraries are required to any module packaged into the EAR file, these libraries could be added in the lib/ folder of the EAR.

By using this mechanism, the use of "Class-Path" entries in the MANIFEST file of each application (WAR / EJB-JAR) is no longer required.

2.1.3.5. EJB class loader

The EJB class loader is responsible for loading all the EJB JARs of the EAR application, thus all the EJBs of the same EAR application are loaded with the same EJB classloader. This class loader is the child of the EAR class loader.

2.1.3.6. WEB class loader

The WEB class loader is responsible for loading the Web components. There is one WEB class loader per WAR file, and this class loader is the child of the EJB class loader. Using this class loader hierarchy (the EJB class loader is the parent of the WEB class loader) eliminates the problem of visibility between classes when a WEB component tries to reference EJBs; the classes loaded with the EJB class loader are definitely visible to the classes loaded by its child class loader (WEB class loader).

The compliance of the class loader of the web application to the java 2 delegation model can be changed by using the `WEB-INF/jonas-web.xml` file. This is described in the section Defining the Web Deployment Descriptor [j2eeprogrammerguide.html#j2ee.pg.classloader].

If the java2-delegation-model element is set to false, the class loader of the web application looks for the class in its own repository before asking its parent class loader. See Section 2.1.3.3.1, “Java 2 delegation model” for more detailed explanations.

2.1.4. Filtering class loaders

2.1.4.1. The problem

By default, all classes exported by the application server can be accessed within an application. Sometimes this may cause problems if there are different versions of the library (in the AS and in the application).

For example, JOnAS is using Apache Commons Digester. If an application wants to use an older version of this library, there is a problem as the default loading mechanism of classes is searching first in the parent classloaders. Then the library provided by JOnAS and not by the application will be used.

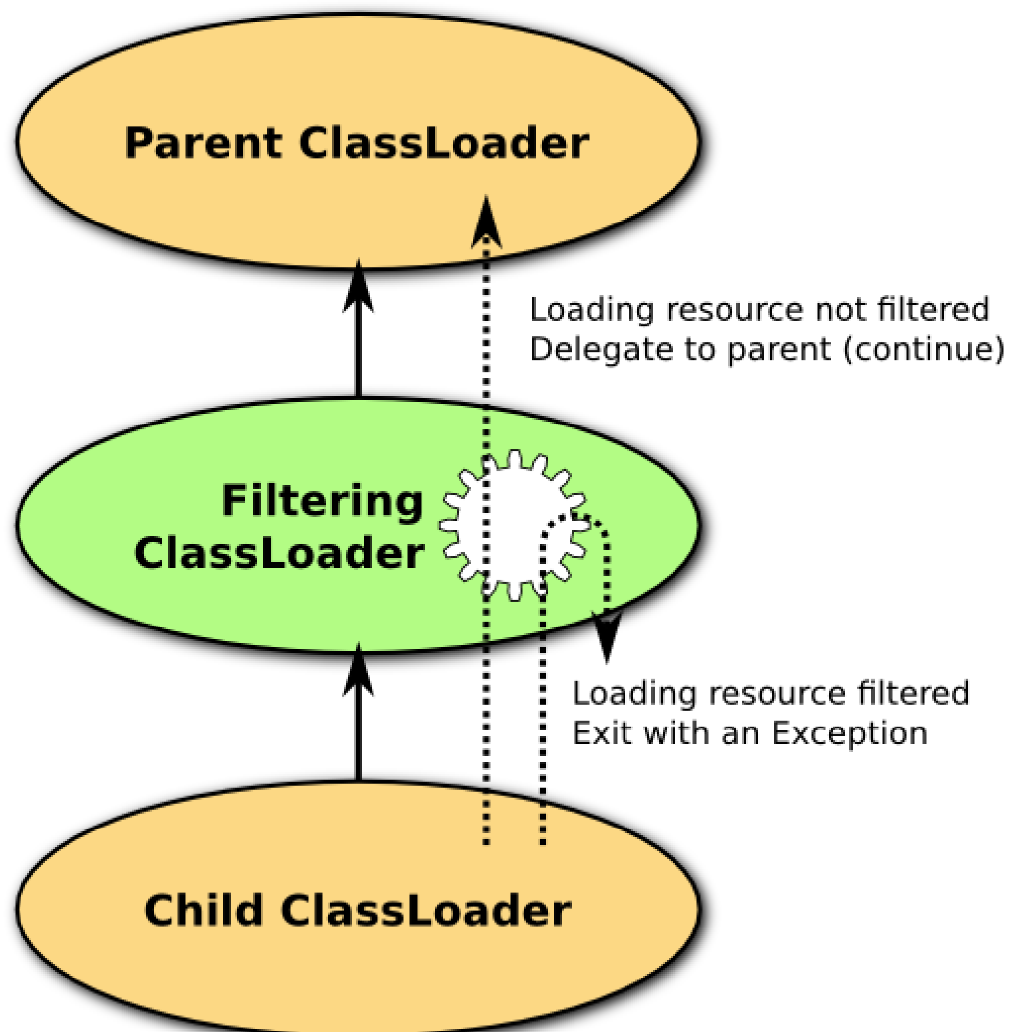
For web applications, this mechanism could be changed by using `java2-delegation-model` but this may raise problems for EAR case, etc.

2.1.4.2. The solution

JOnAS is providing the concept of filtering classloader. A filtered classloader will hide to applications some packages/resources exported by the Application Server.

A `FilteringClassLoader` is simply placed between a child `ClassLoader` and its parent and filter any requests coming from the child to the parent.

Figure 2.1. Filtering ClassLoader Principle



This approach allows a fine grain control over what is visible or not from a given ClassLoader.

Any loading request for a resource will trigger a matching against a list of excluded patterns. If any of the pattern match, the resource will not be loaded and the flow will return to the caller.

Loading of a filtered resource (image, static content, ...) will result in `null` (the usual return for a resource not found) being returned. If the loading request a filtered class, a `ClassNotFoundException` will be thrown back to the caller (the usual return when a class is not found).

2.1.4.3. Filtering system resources

Some filters are configured by default in JOnAS. They will be applied to the "standalone rars" ClassLoader, effectively filtering resource/class request from all the deployed applications (among just the standalone resource adapters). This is a *system wide* configuration.

These default filters are located in the `JONAS_BASE/conf/classloader-default-filtering.xml` file.

```
<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>
        <filter-name>org.apache.commons.digester.*</filter-name>
    </default-filters>
</class-loader-filtering>
```

By modifying the set of filters, some packages will be hidden to the applications. Note that resources name can be specified. For example, when putting a resource in `JONAS_BASE/conf` folder, this resource can be obtained by using `ClassLoader.getResource()` method. By adding a filter on this resource, the resource won't be available to applications (This includes EAR, WAR and EJB-JAR modules).

2.1.4.4. Filtering deployed modules

In addition of the system wide filters configuration, JOnAS allows Java EE modules to provide themselves the filters they requires.

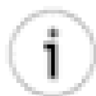
It can be done by providing a `classloader-filtering.xml` file in the info directory of the deployed module (`META-INF/` or `WEB-INF/`).

2.1.4.4.1. Supported module types

Not all modules supports the classloader filtering feature.

Supported modules includes:

- EAR applications (`.ear`)
- Web Applications (`.war`), inside or outside an application
- EjbJars (`.jar`), only the ones outside of an application



Tip

Web Applications filtering rules also apply when they are in an application (EAR).

2.1.4.4.2. Declaring the filters

A Java EE module declares the filters it requires with an XML file named `META-INF/classloader-filtering.xml` (`WEB-INF/classloader-filtering.xml` for a web application). its format is very similar to the system wide configuration file (`<default-filters>` simply become `<filters>`):

```
<class-loader-filtering xmlns="http://org.ow2.jonas.lib.loader.mapping">

    <filters>
        <!-- Excludes all CXF resources as another version is embed in the web application
 -->
        <filter-name>org.apache.cxf.*</filter-name>
        <!-- plus some other dependencies of CXF ... -->
        <filter-name>org.springframework.*</filter-name>
        <filter-name>net.sf.cglib.*</filter-name>
        <filter-name>org.objectweb.asm.*</filter-name>
    </filters>

</class-loader-filtering>
```

2.1.4.4.3. What about Rars and Ejbjars ?

Standalone Resource Adapters are all sharing the same `ClassLoader`. This loader is the parent of all the applications and is the one that is filtered by the system wide configuration file. That means that standalone Rars cannot have their own filtering configuration. They have to use (or modify) the `classloader-default-filtering.xml` file to fit their needs.

EjbJars in Ear do not have support for the filtering. First, an EjbJar should not contains any library, so what could you filter ? Then EjbJars are all placed (along with Ear libraries) inside a unique `ClassLoader`, direct child of the application loader (containing Rars embed in the Ear), so they can indirectly benefit from the filters declared inside the application.