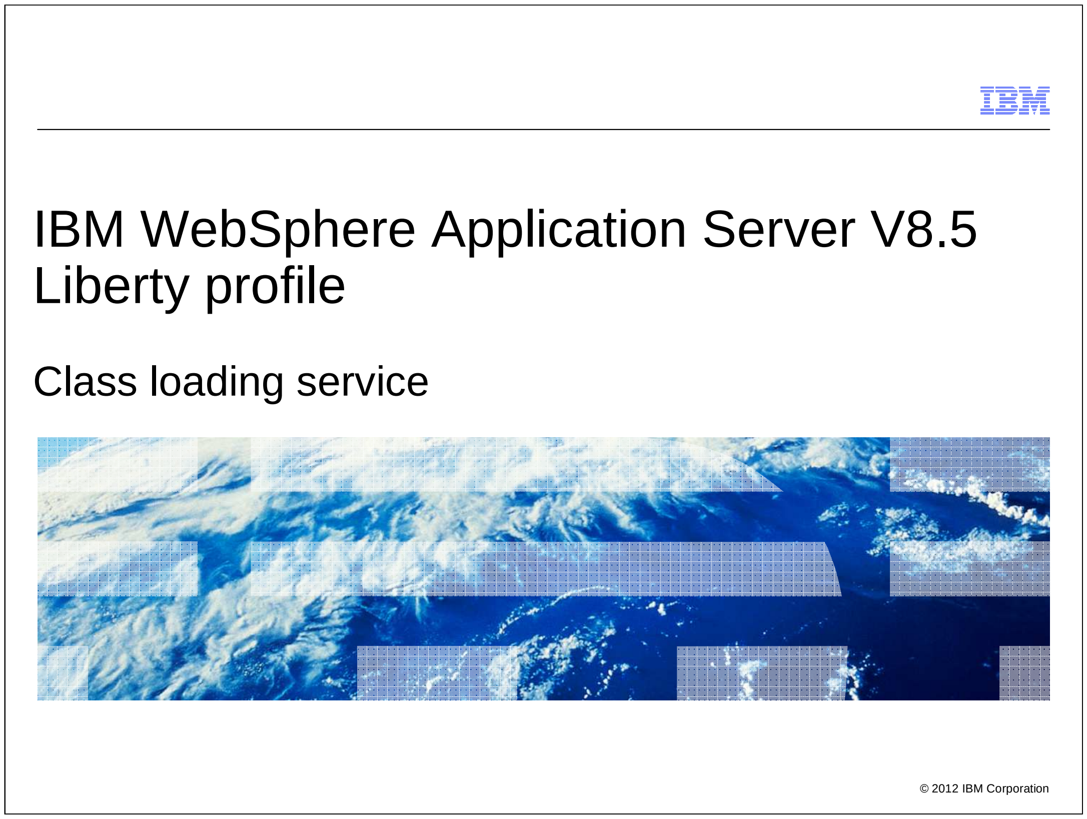


IBM WebSphere Application Server V8.5 Liberty profile

Class loading service

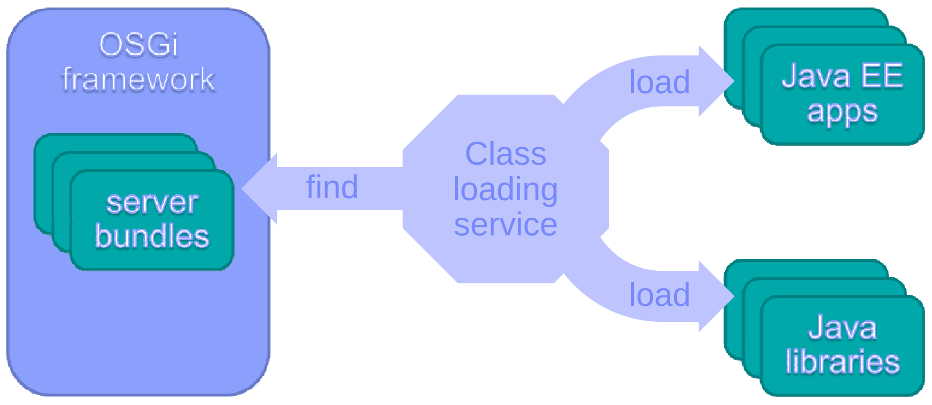
This presentation describes the class loading service in IBM WebSphere Application Server 8.5 Liberty Profile

Section

Overview

The class loading service is responsible for loading Java classes from Java EE applications and from Java libraries external to those applications.

What is the class loading service?



3 Class loading service © 2012 IBM Corporation

OSGi is a modular Java platform that provides class loading for OSGi bundles and clean, *declarative* dependency management.

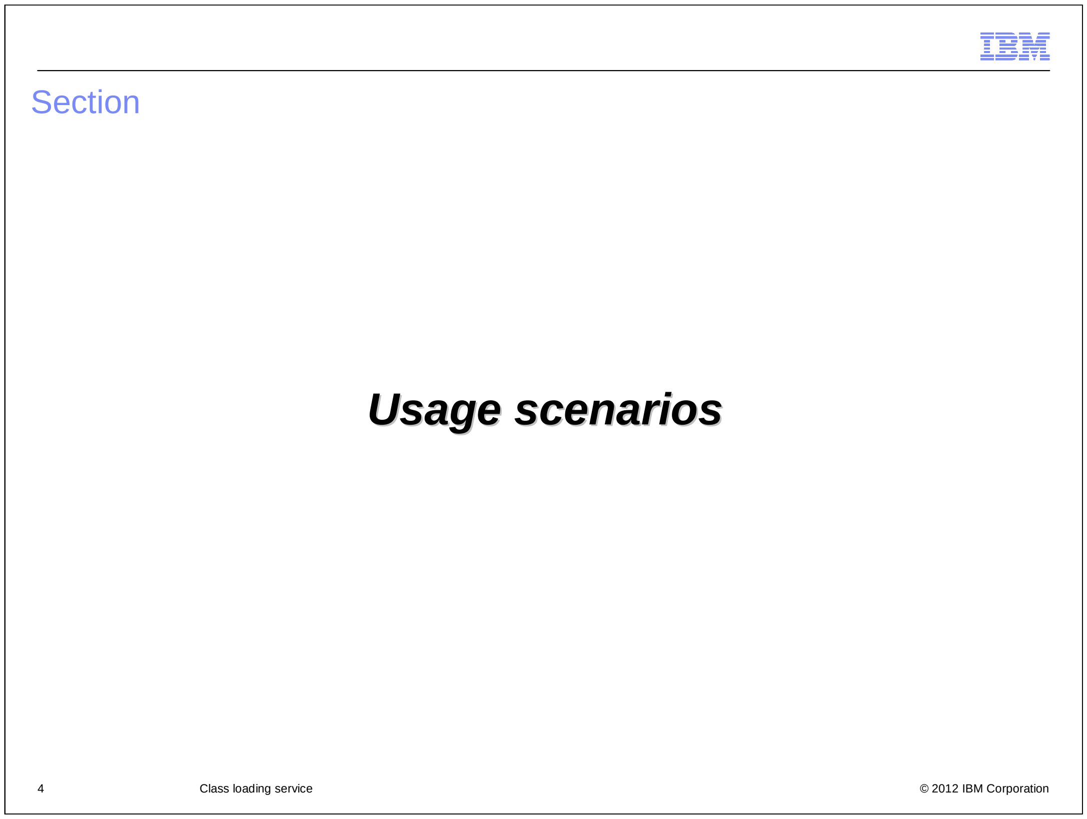
In OSGi, bundles declare what packages they will need, and the framework wires the packages together from different bundles to satisfy the dependencies.

Ordinary Java libraries do not have this sort of declaration and are not handled by the OSGi framework.

The Class Loading Service fills in the gaps: it loads Java EE application classes and Java libraries (libraries that are not OSGi bundles).

Since these classes can depend on classes from the Liberty profile’s OSGi bundles, the Class Loading Service also provides a link into the OSGi framework.

It finds OSGi classes and OSGi resources in common locations. It also puts OSGi classes on the thread context class loader, which is used by several non-OSGi-aware Java mechanisms.

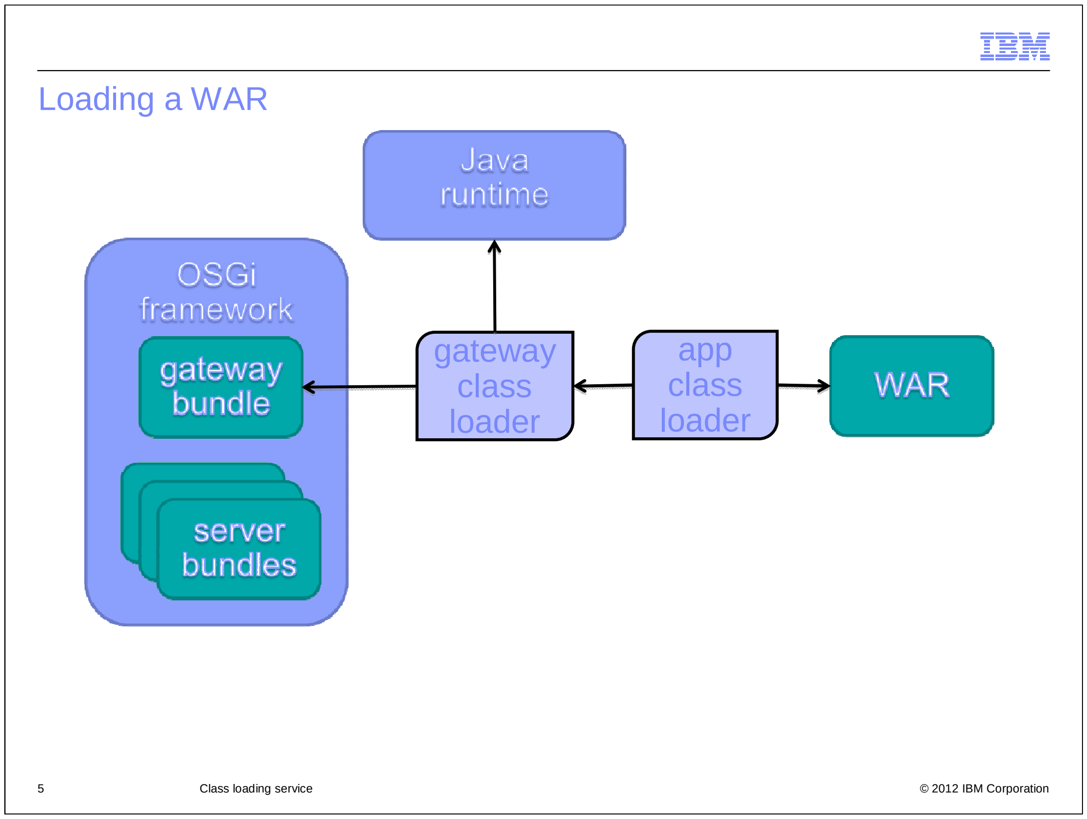


Section

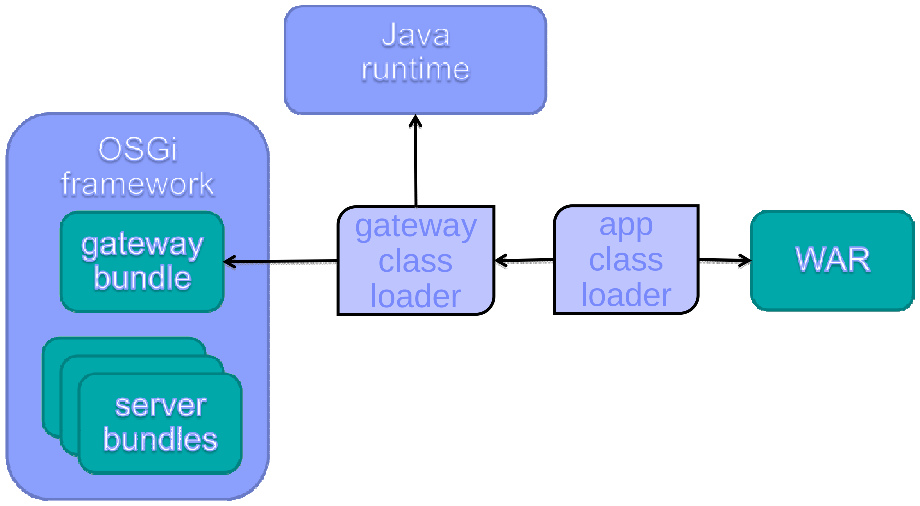
Usage scenarios

4 Class loading service © 2012 IBM Corporation

Let’s look at some of the ways the class loader service can work with applications.



Loading a WAR



5 Class loading service © 2012 IBM Corporation

In version 8.5, the only Java EE applications supported by the Liberty profile are web applications. The simplest packaging for these is the stand-alone WAR.

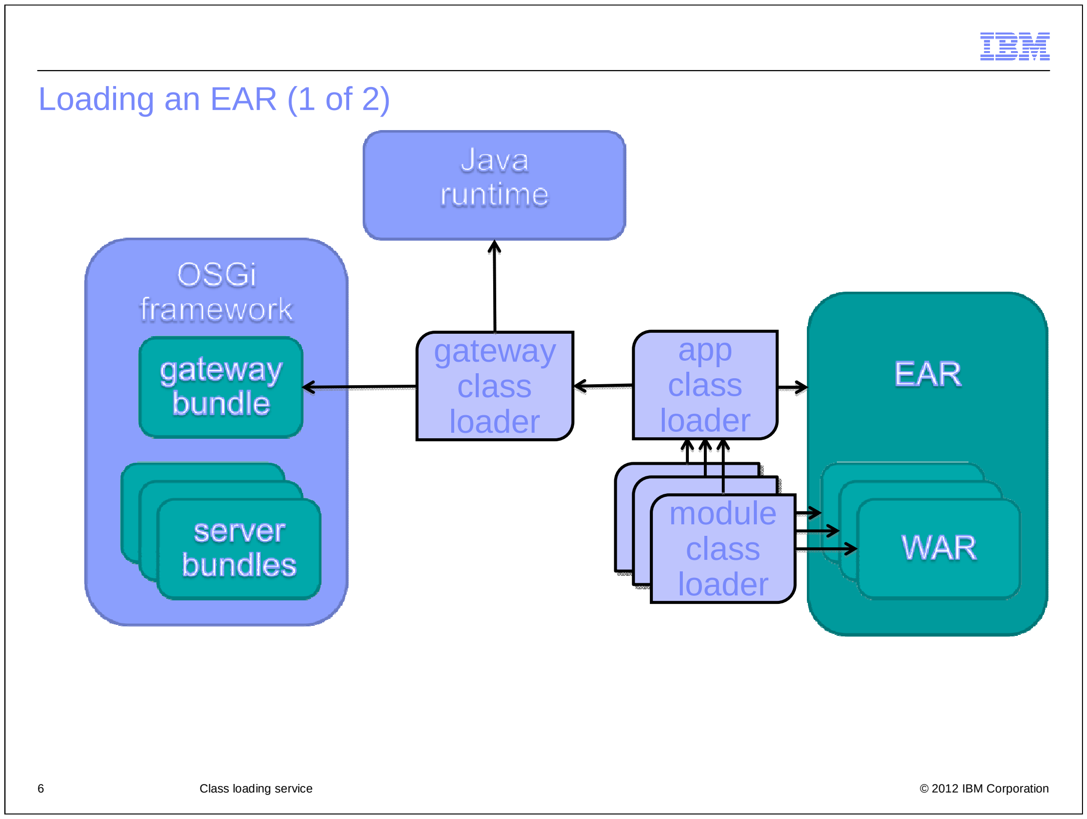
For each WAR an empty bundle is created in the framework. This lets the application 'see' the classes in the OSGi framework. This gateway bundle uses a feature called dynamic package import. The server does not know what packages are going to be used by the application, and the dynamic import allows these to be provided on demand — that is, when the application class loader requests those classes.

This one-bundle-per-application approach allows the OSGi framework to separate the classes used by each application so there is no class space conflict between applications. Each application gets its own set of wirings to the OSGi bundles, and the framework takes responsibility for ensuring consistency even when multiple versions of a package are available.

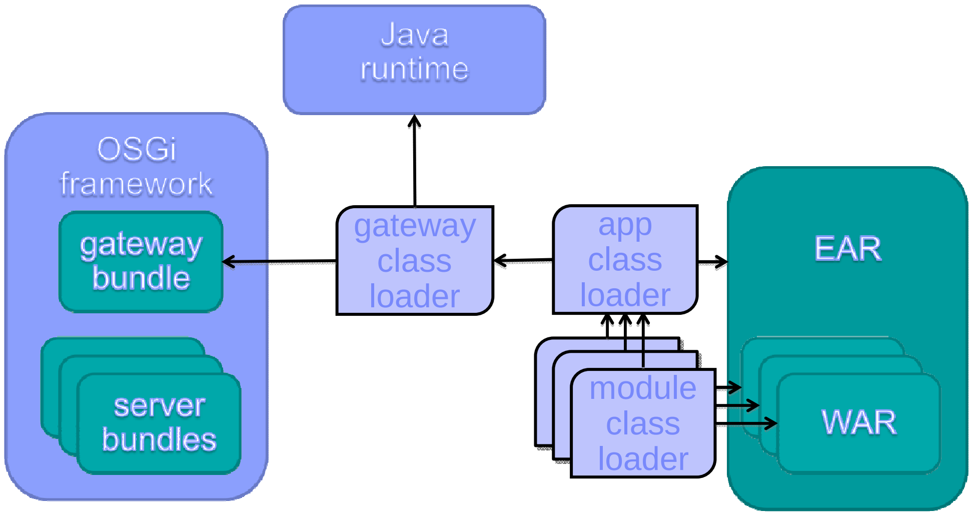
The gateway class loader is a class loading service artifact. It looks for classes in the gateway bundle, and then checks the Java runtime for classes. The reason for this ordering is that the OSGi framework hides some of the Java runtime packages and overrides others. The overriding packages should take precedence, so the OSGi framework is queried first. Since customer applications can require access to JRE-specific packages, these are searched next. The package-overriding behavior of the OSGi framework is preserved, but the package-hiding behavior is circumvented.

The application class loader checks with the gateway class loader first, then looks in the web application for classes, and it is this application class loader that loads application classes at runtime.

Next, let's take a look at how the class loaders are structured for an EAR-based web application.

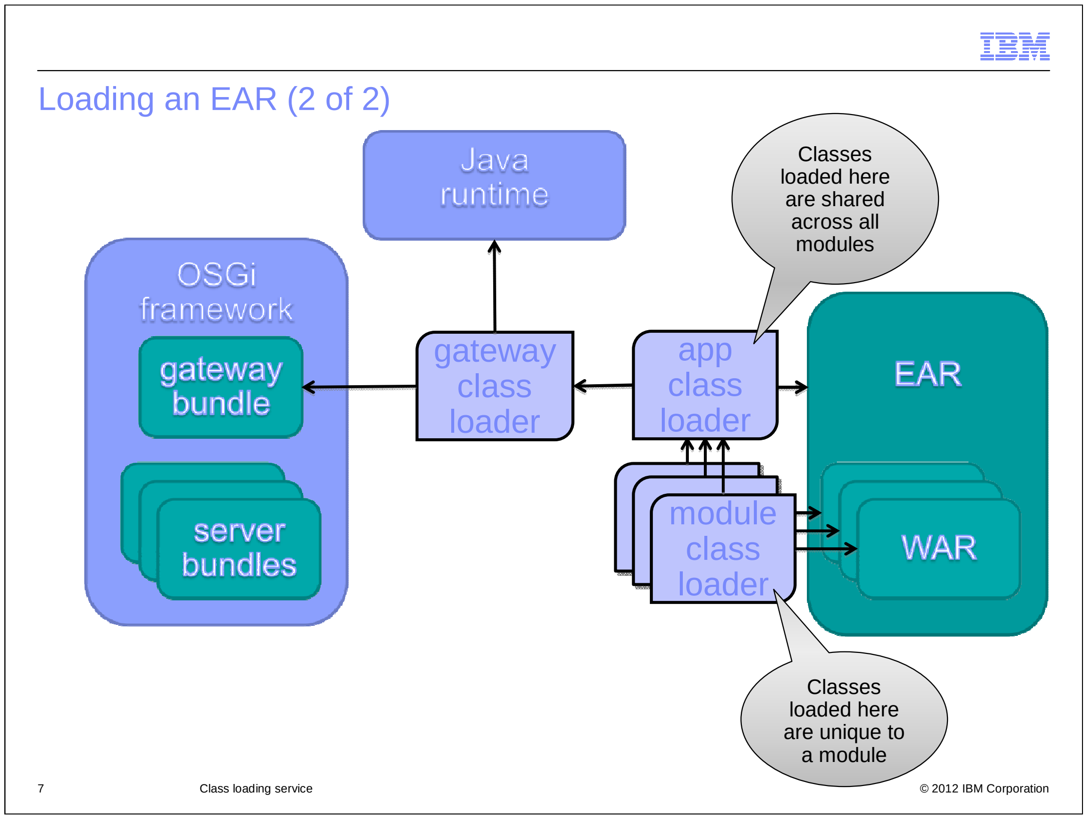


Loading an EAR (1 of 2)

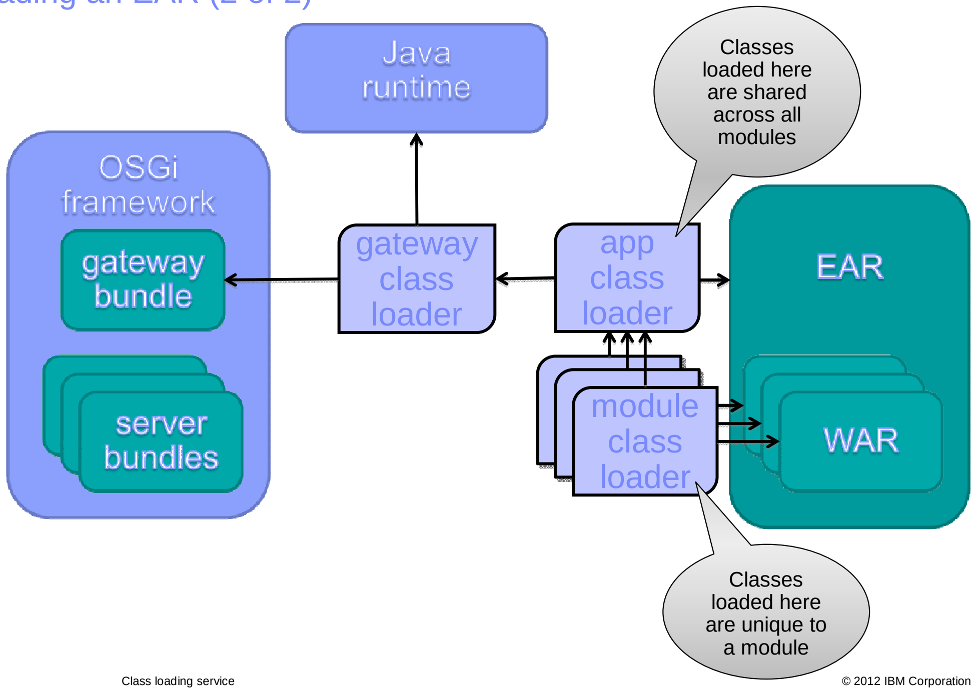


6 Class loading service © 2012 IBM Corporation

As before, there is a unique gateway bundle, a unique gateway class loader, and a unique application class loader for the EAR. There are additional class loaders for each module within the EAR. The runtime class of these module class loaders is the same as the application class loader since the functionality required is almost identical. The module class loaders are unaware of each other. They only see their parent class loader — the application class loader for the EAR.



Loading an EAR (2 of 2)

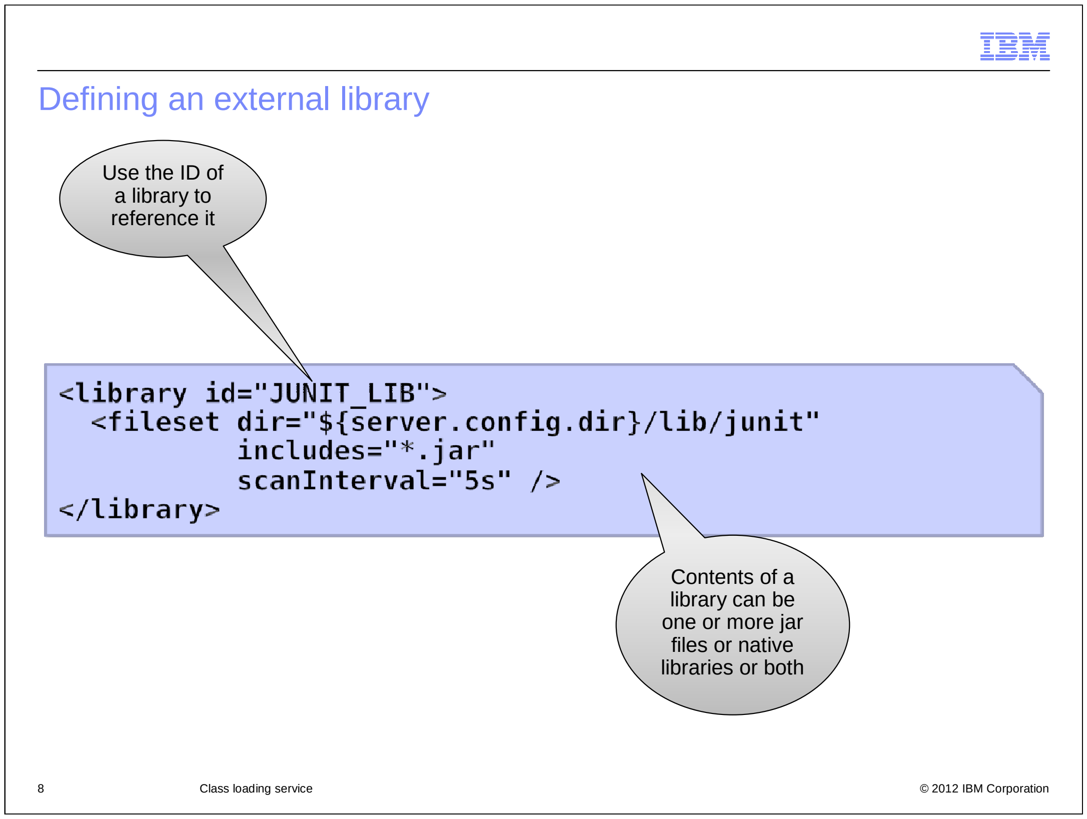


7 Class loading service © 2012 IBM Corporation

This means that packages defined at the EAR level are shared across WAR modules, and objects of those types can be interchanged freely between the modules. Static information in those classes — a counter, for example — is common across all modules.

Classes defined within the WARs are not shared. Two WARs can both contain the same class, and that is fine, but even if they have identical bytecode, the objects are not interchangeable between WARs. Also, Java statics are not shared across these classes.

But what happens when you need to reference an external library?



Defining an external library

Use the ID of a library to reference it
Contents of a library can be one or more jar files or native libraries or both

```
<library id="JUNIT_LIB">
  <fileset dir="${server.config.dir}/lib/junit"
           includes="*.jar"
           scanInterval="5s" />
</library>
```

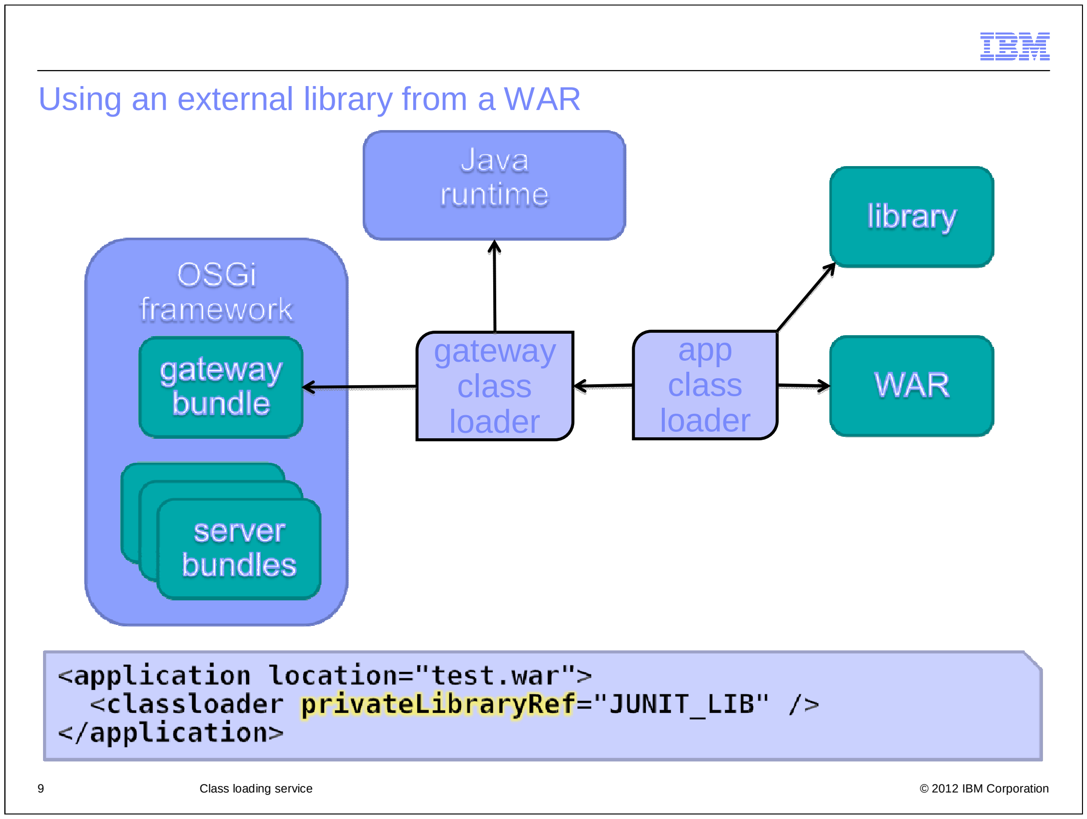
Everything you have seen so far can be achieved by placing an application in the dropins folder, **or** by configuring an application in server.xml. Defining an external library **requires** you to edit the server.xml.

This is an example of the configuration for defining a shared library. This element is a top-level element in the server.xml.

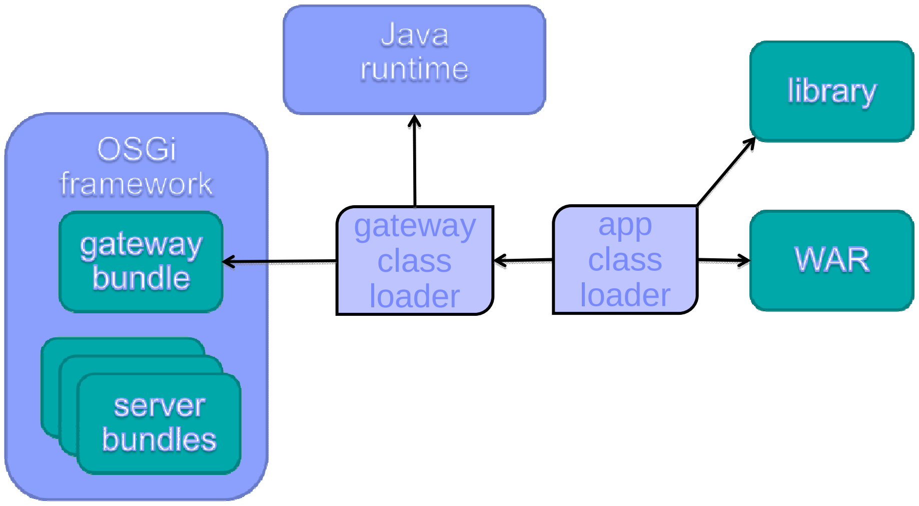
A library is referenced by its ID, and contains only files. There is no support for adding a directory to a library.

Files can be JAR files or native libraries, although the semantics of native library support in Java mean that adding a native library here only adds it to the search path when libraries are loaded. Native libraries are not loaded by a class loader directly.

Note that this library definition is for use by Java EE applications. There is a separate facility known as the bundle repository for configuring additional bundles to be used by OSGi applications.



Using an external library from a WAR



```
<application location="test.war">
  <classloader privateLibraryRef="JUNIT_LIB" />
</application>
```

9 Class loading service © 2012 IBM Corporation

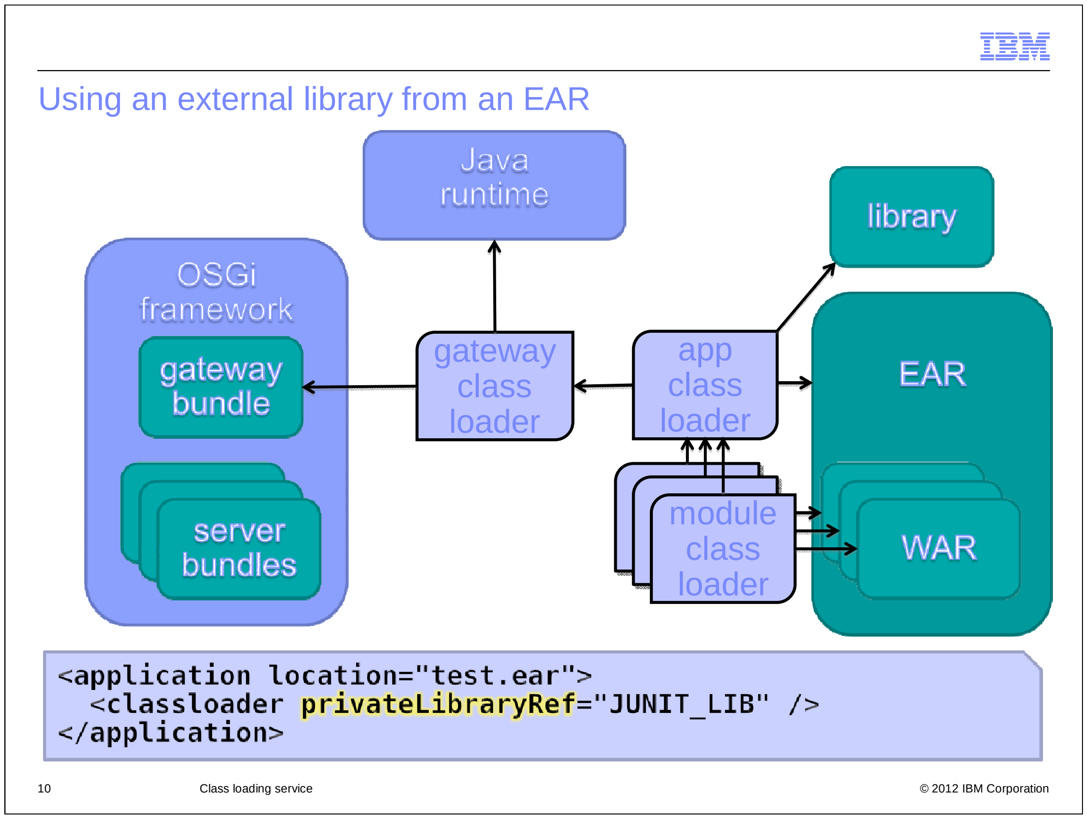
Once the library is defined, it can be used from a configured application by configuring the class loader for that application as demonstrated at the bottom of this slide.

The <application> element **can** exist without the <classloader> child, but as far as the class loading service is concerned, that is identical to a dropped-in application.

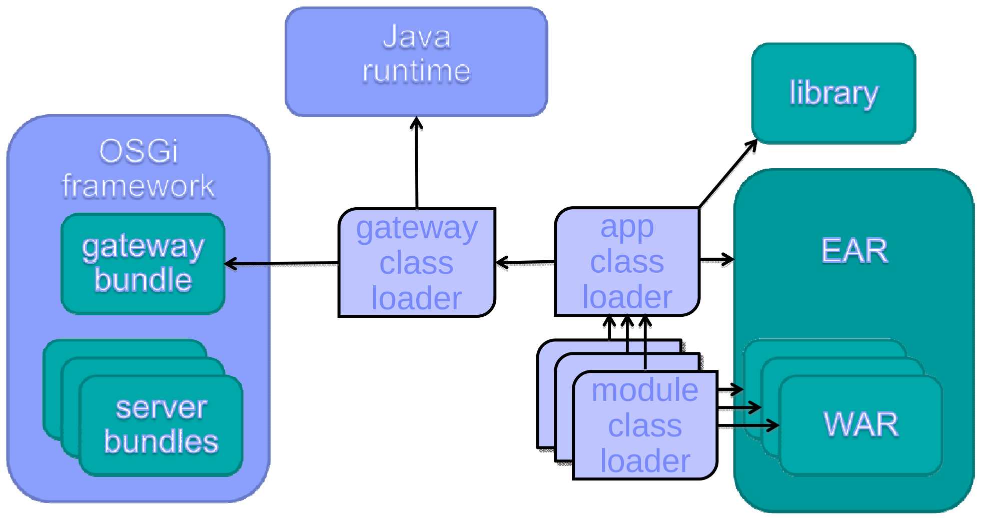
Here the application has had the Junit library added to its class path. The application class loader will check the library for classes and resources.

The class loader configuration is only valid for Java EE applications. If it is provided for an OSGi application, it is ignored.

The picture for an EAR looks very similar…



Using an external library from an EAR

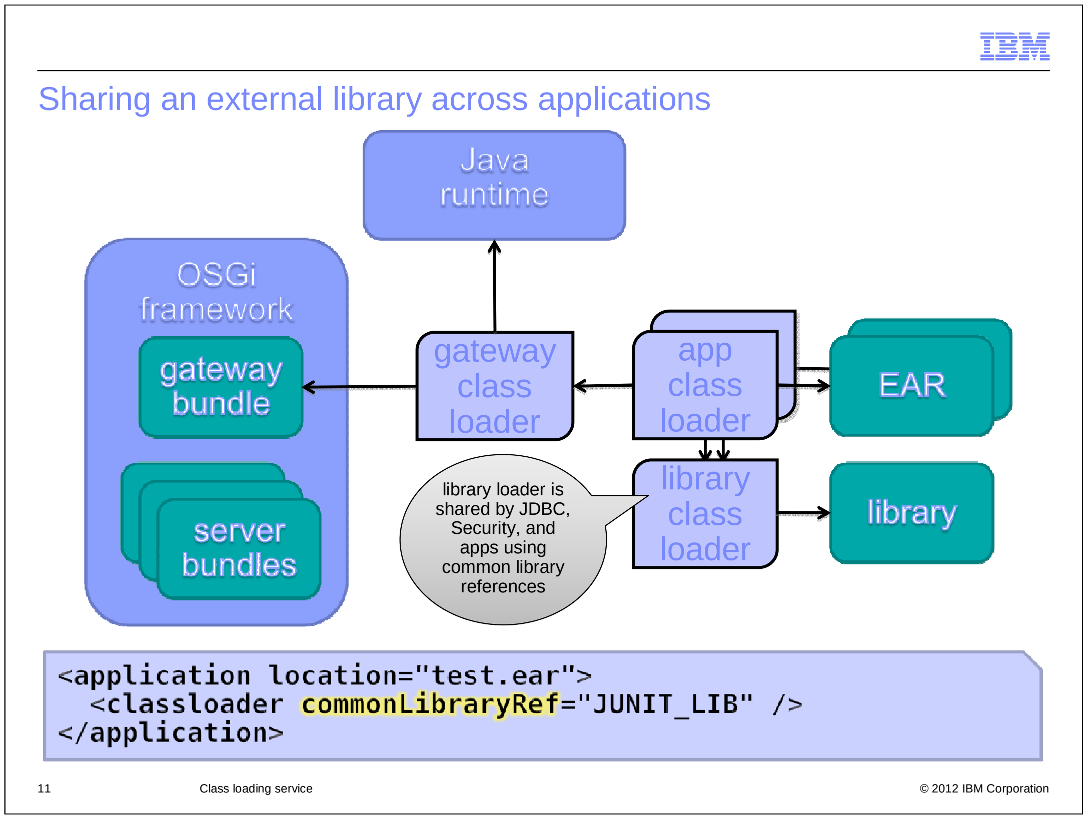


```
<application location="test.ear">
  <classloader privateLibraryRef="JUNIT_LIB" />
</application>
```

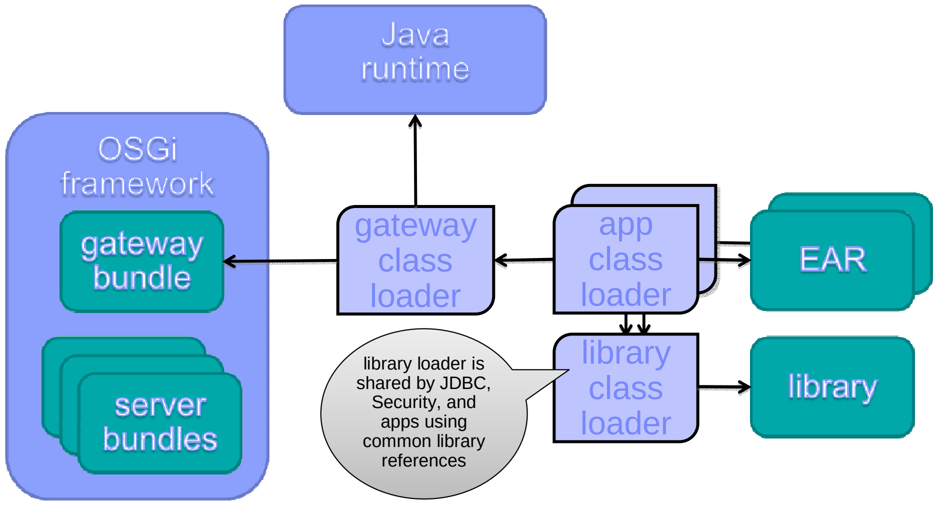
The module class loaders do not see the library directly — they merely have access to it through the application class loader. The Liberty profile does not allow configuration of class loaders at the module level.

On this slide, as on the previous one, the library is loaded by the application class loader directly. Although there can be two apps loading the same library classes from the same files in the file system, they will each see separate copies of these classes. Objects of these classes are not exchangeable between the apps and any static fields seen by one application are completely separate from the static fields seen by the other application.

There is a way that two apps can share the same loaded library classes, reducing the memory footprint and increasing interoperability…



Sharing an external library across applications



```
<application location="test.ear">
  <classloader commonLibraryRef="JUNIT_LIB" />
</application>
```

11 Class loading service © 2012 IBM Corporation

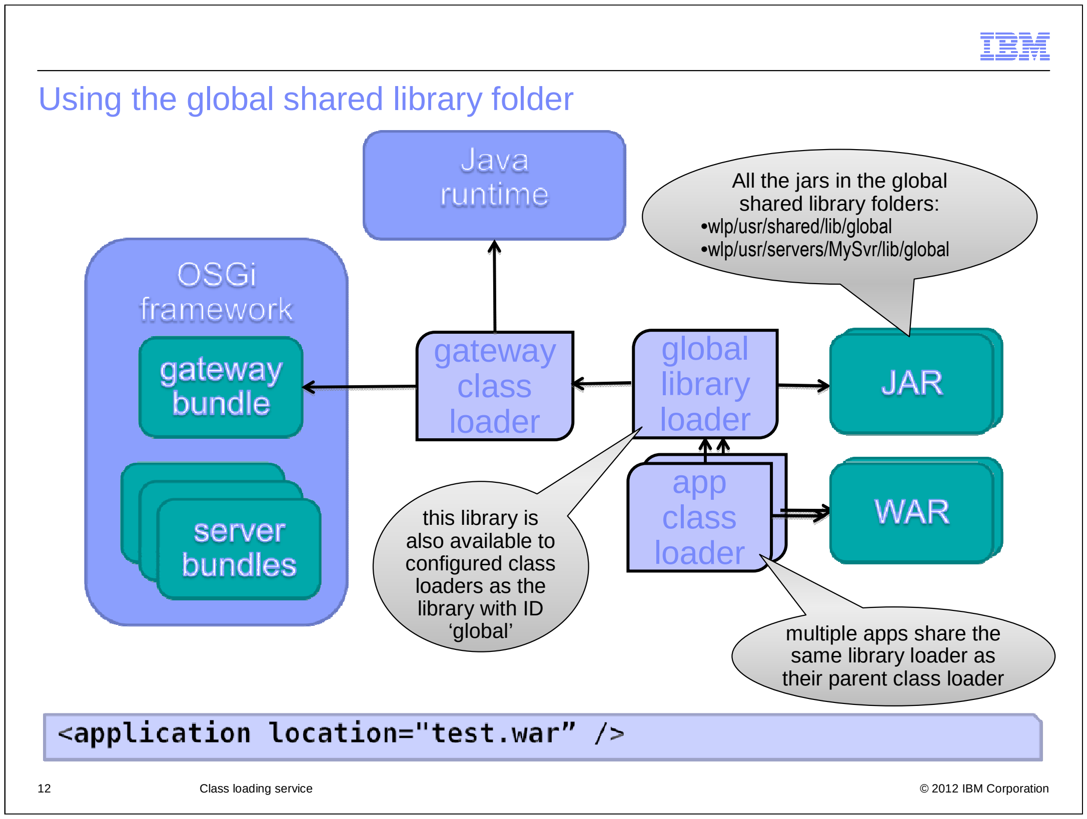
Take a look at the configuration change shown at the bottom of the screen, highlighted in yellow. The reference attribute has changed from *privateLibraryRef* to *commonLibraryRef*. This ensures that the application class loader defers to a common library loader to load an external library.

You have already seen that the module class loaders of an EAR are not directly affected by the use of private library references. The same holds true for common library references. The modules and their loaders are not shown here, and the EAR can just as easily be a WAR.

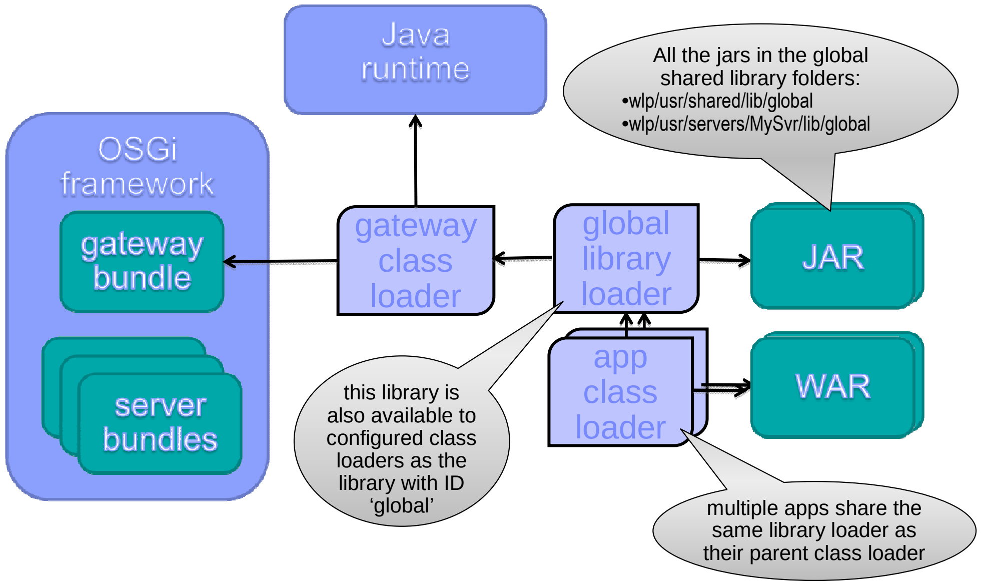
This configuration allows applications to access library classes directly **and** ensure that they are using the same classes as other applications.

There is no special implementation for a library class loader: it uses the same class as the application class loader.

There is at most one library class loader available for a library at a time (it should be replaced when a library is updated). This library loader also gets used for JDBC drivers and for custom login modules. This means that applications can also use this approach to access the same classes that the JDBC and Security components can be using.



Using the global shared library folder


Java runtime
OSGi framework
gateway bundle
server bundles
gateway class loader
global library loader
JAR
app class loader
WAR
All the jars in the global shared library folders:
•wlp/usr/shared/lib/global
•wlp/usr/servers/MySvr/lib/global
this library is also available to configured class loaders as the library with ID 'global'
multiple apps share the same library loader as their parent class loader

```
<application location="test.war" />
```

12 Class loading service © 2012 IBM Corporation

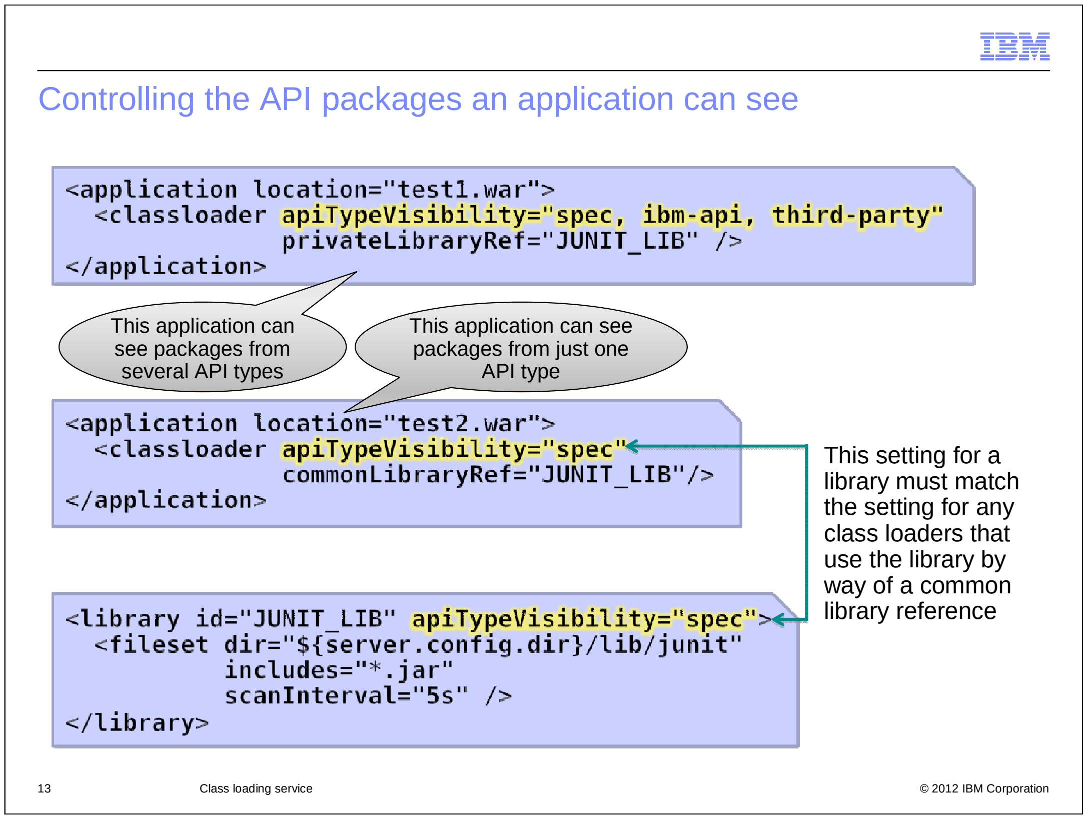
There is a facility for applications to use a pre-defined folder and load shared libraries from there. To use this, libraries must be placed in the server-specific global library folder, or the global library folder shared across all servers. Application class loaders access the shared library through their parent class loader chain rather than by delegation.

This facility is mandated by the servlet specification, but it breaks the nice single-gateway-bundle-per-application pattern established in the Liberty profile. For this reason, the facility is not used unless two very specific conditions are met:

If either of these conditions is not met for any given application, that application will revert to the pattern shown on previous slides. First, at least one of the global shared library folders must exist and must not be empty. Second, the application configuration (if there is one) must not specify a <classloader> element. If either of these conditions are not met for any given application, that application will revert to the pattern shown on previous slides.

It is still possible for an application with a configured class loader to reference this library as a private or a common library by using the special library ID 'global'. However, for that application, the configured class loader will use either the external library or shared external library model rather than this one.

This model of library use is supported purely for compliance with the servlet specification (Servlet 3.0 section 10.7.2) . It is not recommended because it breaks the class space isolation that the Liberty profile can provide.



Controlling the API packages an application can see

```
<application location="test1.war">
  <classloader apiTypeVisibility="spec, ibm-api, third-party"
               privateLibraryRef="JUNIT_LIB" />
</application>
```

This application can see packages from several API types
This application can see packages from just one API type

```
<application location="test2.war">
  <classloader apiTypeVisibility="spec"
               commonLibraryRef="JUNIT_LIB"/>
</application>
```

```
<library id="JUNIT_LIB" apiTypeVisibility="spec">
  <fileset dir="${server.config.dir}/lib/junit"
           includes="*.jar"
           scanInterval="5s" />
</library>
```

This setting for a library must match the setting for any class loaders that use the library by way of a common library reference

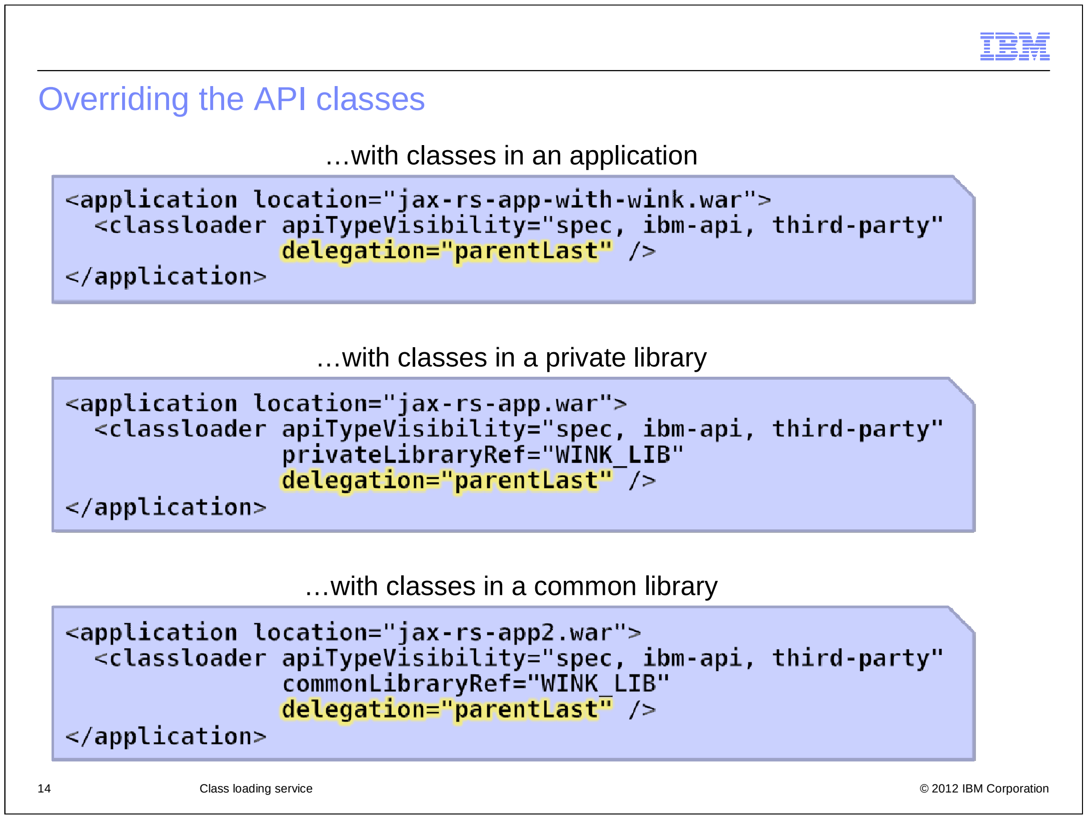
The APIs provided with the Liberty Profile are divided into four distinct types: ‘spec’, ‘ibm-api’, ‘ibm-spi’, and ‘third-party’.

By default, an application can only access two of these: ‘spec’ and ‘ibm-api’

To change which types an application can see, you can specify an attribute called ‘apiTypeVisibility’ on the class loader, with a comma-separated list of the types required.

If an application uses any libraries through a common library reference, it is sharing the common class loader for that library. In order to preserve class space consistency for all common users of that library, the same attribute with the same value must be specified on the library definition.

Remember that this is only for Java EE apps. OSGi allows much finer-grained control at the package level, so this coarse level of control is unnecessary.


Overriding the API classes

…with classes in an application

```
<application location="jax-rs-app-with-wink.war">
  <classloader apiTypeVisibility="spec, ibm-api, third-party"
               delegation="parentLast" />
</application>
```

…with classes in a private library

```
<application location="jax-rs-app.war">
  <classloader apiTypeVisibility="spec, ibm-api, third-party"
               privateLibraryRef="WINK_LIB"
               delegation="parentLast" />
</application>
```

…with classes in a common library

```
<application location="jax-rs-app2.war">
  <classloader apiTypeVisibility="spec, ibm-api, third-party"
               commonLibraryRef="WINK_LIB"
               delegation="parentLast" />
</application>
```

14 Class loading service © 2012 IBM Corporation

If an application needs to override a provided API, either with classes in the application itself or in an external library, this can be achieved by using the delegation setting. This defaults to 'parentFirst' but can be set to 'parentLast'. This causes the application class loader to look in the application classes and the library classes before consulting the logical parent class loader — the gateway bundle.

Consider the Apache Wink APIs. These are provided as vendor APIs with the Liberty profile, but an application can provide and use its own version of Wink. In this case, the 'parentLast' delegation setting can be used to ensure that the application uses an alternative version of the classes in preference to those available in the Liberty profile.

This setting can be used to prefer classes from an application, a private library, or a common library over those provided by the Liberty profile.

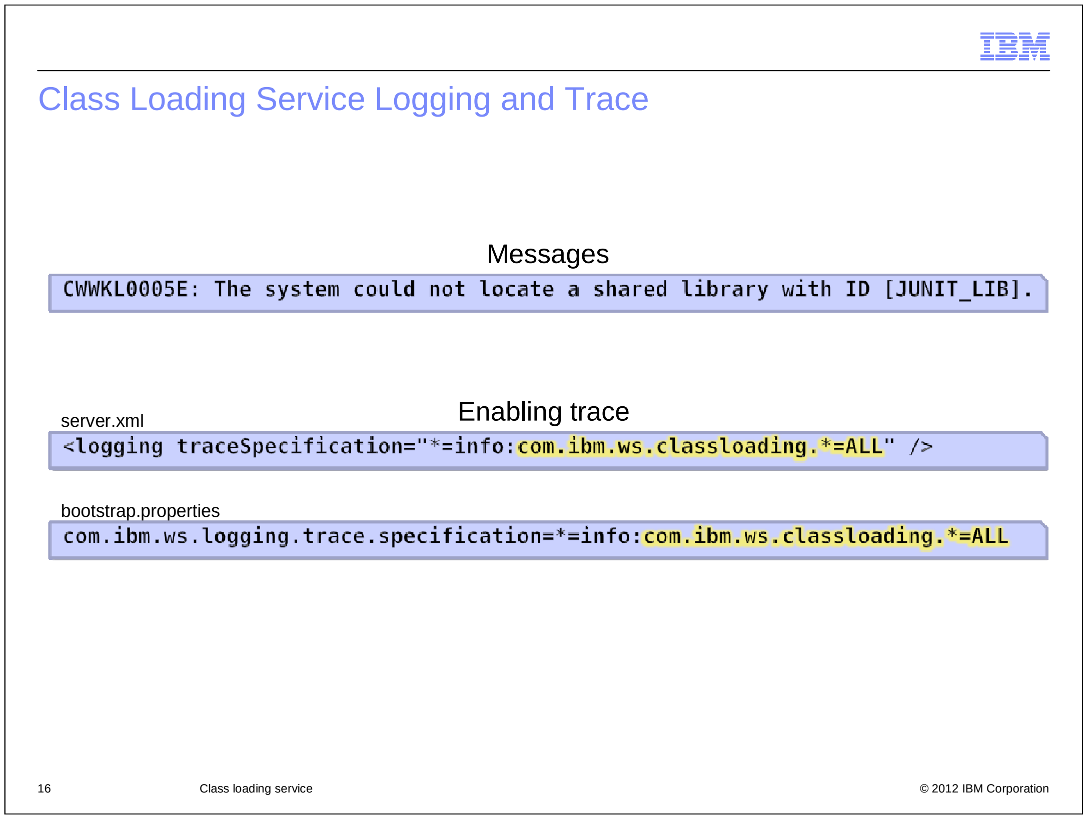
Since OSGi allows applications to specify the versions of packages to be imported, once again this coarser level of control is only needed for Java EE applications.

Section

Troubleshooting

15 Class loading service © 2012 IBM Corporation

In these slides, you will find out about the diagnostic tools available for understanding the behavior of the Class Loading Service.



Class Loading Service Logging and Trace

Messages

```
CWWKL0005E: The system could not locate a shared library with ID [JUNIT_LIB].
```

Enabling trace

server.xml

```
<logging traceSpecification="*=info:com.ibm.ws.classloading.*=ALL" />
```

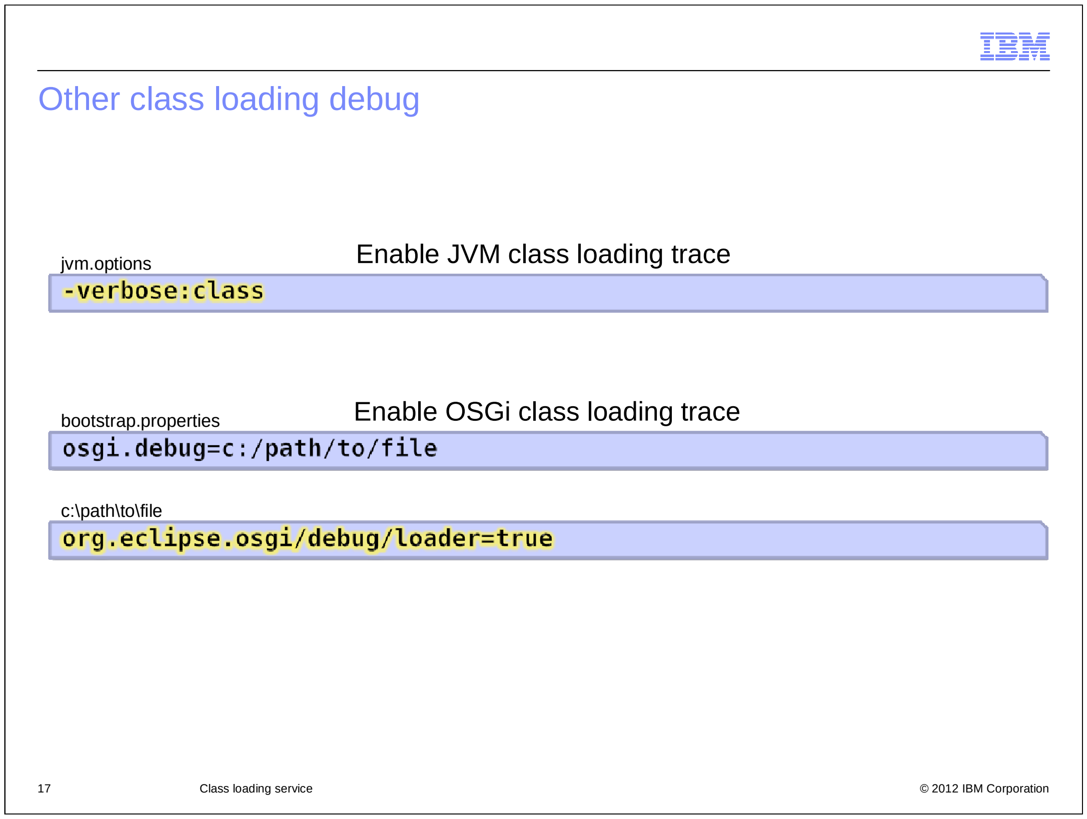
bootstrap.properties

```
com.ibm.ws.logging.trace.specification=*=info:com.ibm.ws.classloading.*=ALL
```

16 Class loading service © 2012 IBM Corporation

All the messages from the Class Loading Service begin with a key of CWWKL, followed by four numbers and another letter. These are visible in the terse console.log and the more verbose, time-stamped messages.log. The example here indicates a problem retrieving a library.

To enable trace for the Class Loading Service, you can add or amend the logging configuration element in the server.xml. The value of the traceSpecification is a colon-delimited list. The first entry in this example is the default trace — this must be made explicit if still required when a traceSpecification is provided. The second entry, highlighted in yellow, is the part that turns on the class loading trace. “ALL” is a very verbose setting, and it can suffice to set it to “FINE”. There is an option to turn on the trace in the bootstrap.properties as well, but remember that this is only read at start up. The server.xml approach allows trace settings to be modified in a running server instance.



Other class loading debug

Enable JVM class loading trace

jvm.options

```
-verbose:class
```

Enable OSGi class loading trace

bootstrap.properties

```
osgi.debug=c:/path/to/file
```

c:\path\to\file

```
org.eclipse.osgi/debug/loader=true
```

17 Class loading service © 2012 IBM Corporation

It is often helpful to enable JVM class loading trace, which tells you the class loader for every class loaded in the entire system. This can be done in the jvm.options file. Note that you can also get similar information from a javacore from an IBM virtual machine, or an hprof file from a Sun JVM.

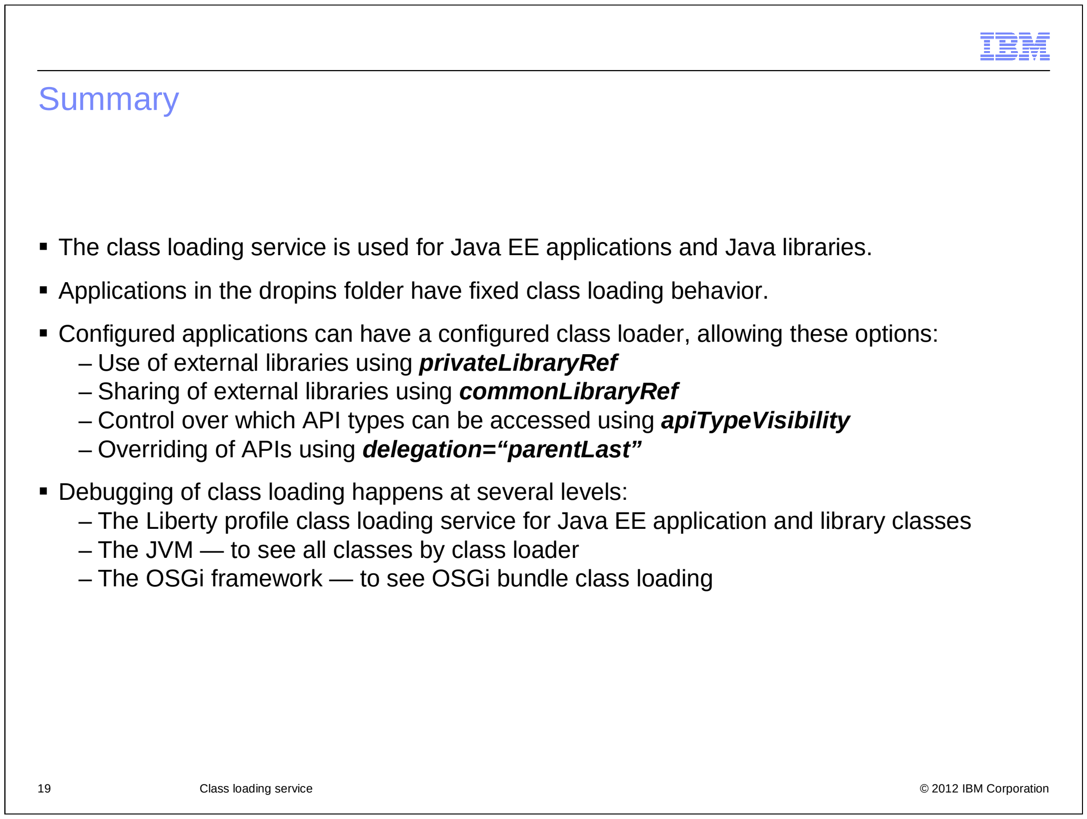
Sometimes you will want to understand what the OSGi framework is doing. This requires two files to be edited. The location of the OSGi debug properties file is supplied by way of the bootstrap.properties file. The contents of the OSGi debug properties file determine the tracing, and the highlighted yellow text shows how to enable OSGi class loading trace.

Section

Summary

18 Class loading service © 2012 IBM Corporation

To recap…



Summary

- The class loading service is used for Java EE applications and Java libraries.
- Applications in the dropins folder have fixed class loading behavior.
- Configured applications can have a configured class loader, allowing these options:
  – Use of external libraries using ***privateLibraryRef***
  – Sharing of external libraries using ***commonLibraryRef***
  – Control over which API types can be accessed using ***apiTypeVisibility***
  – Overriding of APIs using ***delegation="parentLast"***
- Debugging of class loading happens at several levels:
  – The Liberty profile class loading service for Java EE application and library classes
  – The JVM — to see all classes by class loader
  – The OSGi framework — to see OSGi bundle class loading

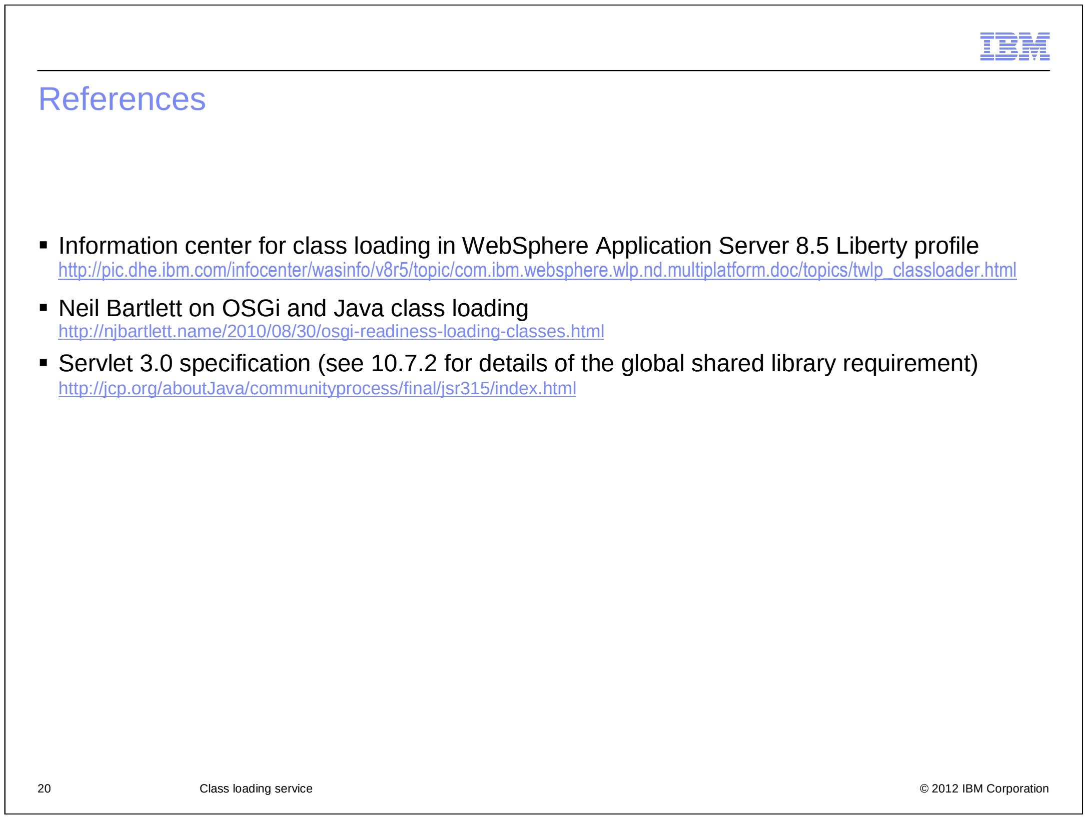
19 Class loading service © 2012 IBM Corporation

The Class Loading Service is used for non-OSGi applications and libraries.

Dropped-in apps cannot be configured and use the default class loading behavior.

Configured applications can have configured class loaders, allowing the use of shared libraries and tighter control over APIs.

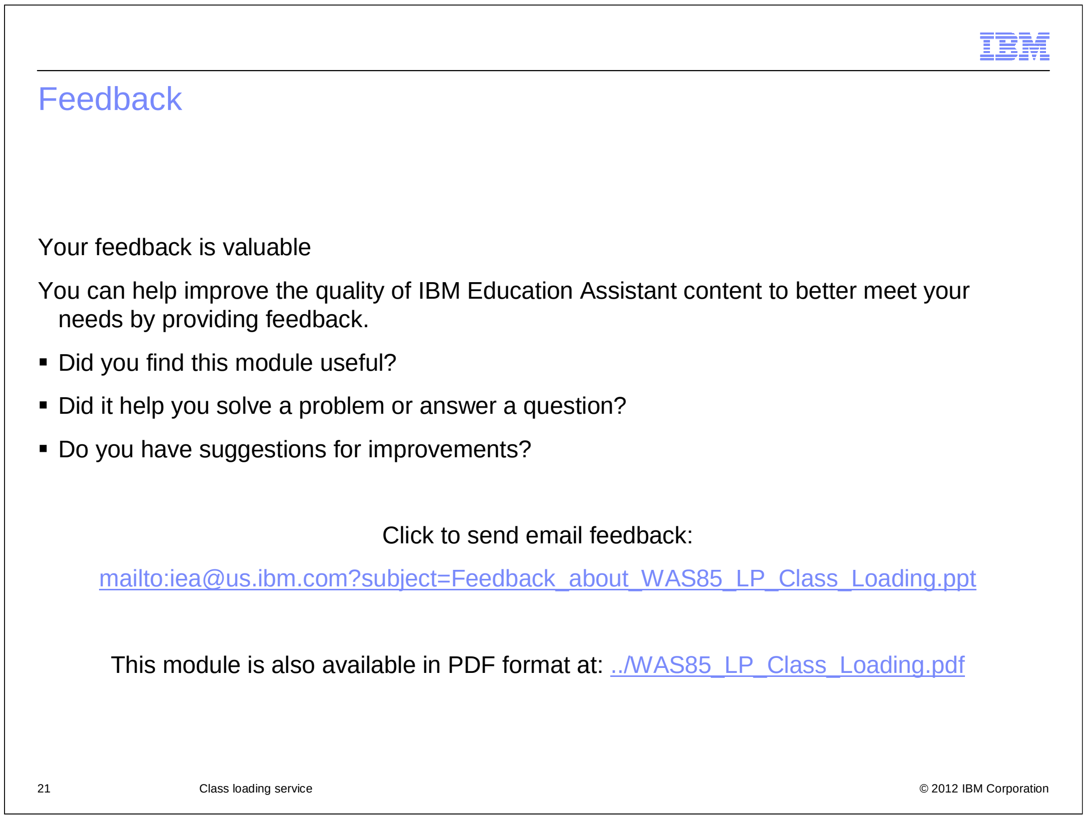
Debugging is possible at three levels: the Class Loading Service, the JVM, and the OSGi framework.


References

- Information center for class loading in WebSphere Application Server 8.5 Liberty profile
  http://pic.dhe.ibm.com/infocenter/wasinfo/v8r5/topic/com.ibm.websphere.wlp.nd.multiplatform.doc/topics/twlp_classloader.html
- Neil Bartlett on OSGi and Java class loading
  http://njbartlett.name/2010/08/30/osgi-readiness-loading-classes.html
- Servlet 3.0 specification (see 10.7.2 for details of the global shared library requirement)
  http://jcp.org/aboutJava/communityprocess/final/jsr315/index.html

20 Class loading service © 2012 IBM Corporation

See these references for additional background information about **the class loading service**



Feedback

Your feedback is valuable

You can help improve the quality of IBM Education Assistant content to better meet your needs by providing feedback.

- Did you find this module useful?
- Did it help you solve a problem or answer a question?
- Do you have suggestions for improvements?

Click to send email feedback:

mailto:iea@us.ibm.com?subject=Feedback_about_WAS85_LP_Class_Loading.ppt

This module is also available in PDF format at: ../WAS85_LP_Class_Loading.pdf

21 Class loading service © 2012 IBM Corporation

You can help improve the quality of IBM Education Assistant content by providing feedback.

Trademarks, disclaimer, and copyright information