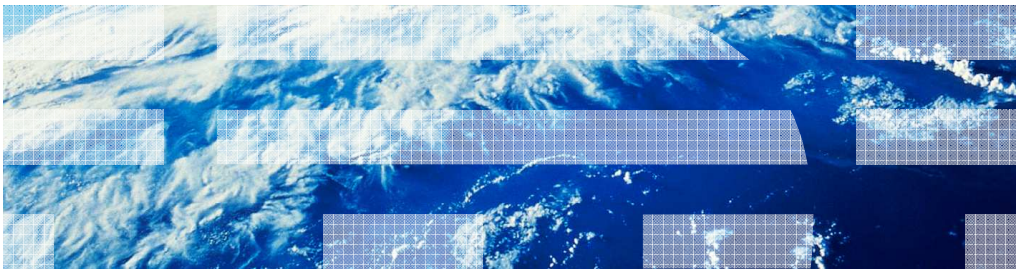
IBM WebSphere Application Server Feature Pack for OSGi Applications and Java Persistence API 2.0

OSGi Fundamentals



WebSphere software

This presentation explains the fundamentals of the IBM WebSphere® Application Server Feature Pack for OSGi Applications and Java™ Persistence API 2.0

Table of contents

There is information in this module that is beyond the feature pack because this module covers OSGi as a technology.

The purpose of this module is two-fold:

First, setting the background for other modules in this collection, which assume a basic amount of familiarity with OSGi concepts and terminology. T

Second, to cover the basic aspects of OSGi that customers need to understand to use the feature pack and pointing out possible pitfalls and gotchas

It will start with a very brief overview of OSGi. What is OSGi and why do people care?

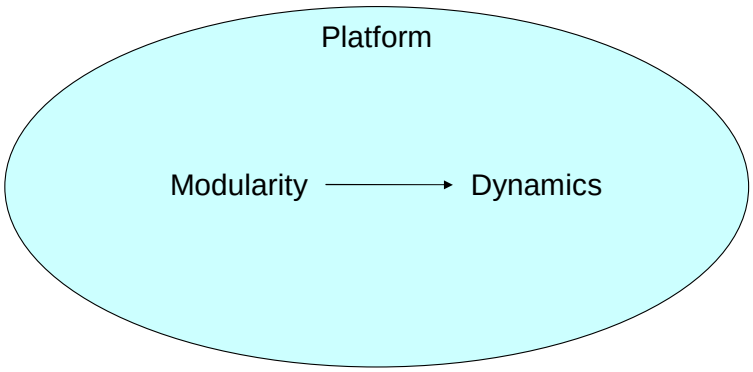
Then bundles are covered in some detail since they sit at the very heart of OSGi. Services and the service registry are covered in a more cursory manner afterwards.

Finally, a bit about problems that are commonly encountered when starting with OSGi.

What and why, OSGi?

OSGi

- Dynamic module system for Java



Officially, the OSGi service platform is known as the “dynamic module system for Java”.

Modularity in the Java language is restricted to just the modularity on the level of classes and their methods: a class can be visible or hidden, similarly a method can be visible or hidden (given some conditions). However, Java does not support modularity- that is encapsulation and information hiding - on any higher level than this (apart from writing independent applications).

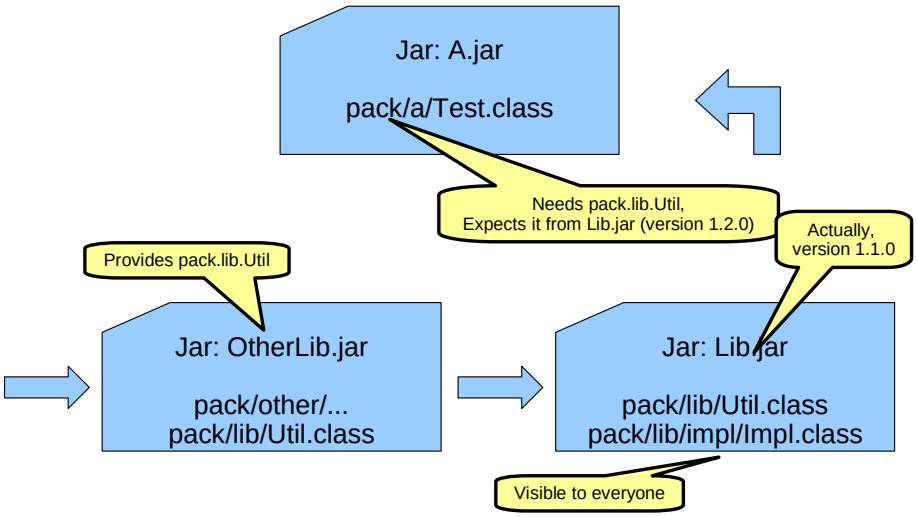
Why is this important? Because Java deployments on an application server typically pull in many different libraries at conflicting versions. Without encapsulation - and worse, with no dependency management, the best and safest way to use modules is to use a separate version in every application and make sure that all modules in the application use the same version of the library.

This is where OSGi comes in. OSGi provides modularity on the level of jar files (or modules). And it not only supports information hiding but also declarative dependencies. What this means is that an OSGi module (called a bundle) defines not only what packages it provides but also which ones it needs. Bundles and packages carry not only a name but also have specific versions.

Building on top of the modularity comes the second core aspect of OSGi dynamics. This is all about associating a life cycle with modules so that they can be activated as needed, stopped and updated, or replaced.

OSGi is not an orthogonal technology. It cannot be dropped in like a lot of Java libraries. It is a framework, and you must understand the rules and write your applications specifically in order to use it. It can be compared to a very light weight container. Your OSGi applications will always run in the OSGi framework like your EJB modules always run in an EJB container.

Java modularity



So what is the problem of modularity in Java?

Here is an example.

Here you have a jar file, A.jar. It has a class pack.a.Test needs pack.lib.Util.

During testing it was compiled against this class in Lib.jar at version 1.1.0.

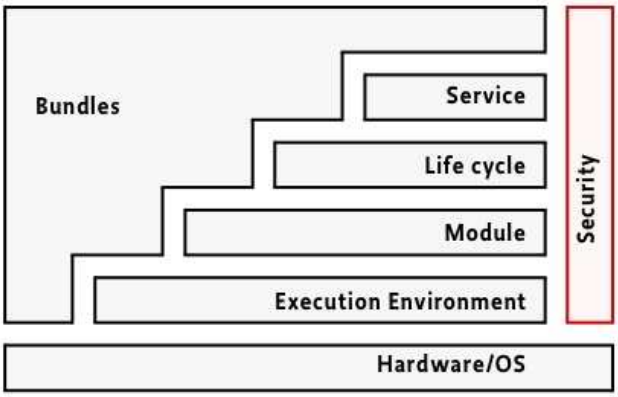
However, during deployment the class is provided by the first jar that contains pack/lib/Util.class. The class carries no version information, so there is no way to diagnose problems around using that class. The jar file also contains no version information, so even if the class came from Lib.jar it might be the wrong one.

In Java the typical hierarchical class loader structure can be linearized to a list of jar files that will be searched in order. Worse, parent class loaders are by default searched first, so even if Lib.jar were co-located with A.jar, pack.a.Test still might get pack.lib.Util from the wrong place.

This problem can be especially hard to diagnose when OtherLib only accidentally contains pack.lib.Util where its primary classes it is meant to provide are in another package, say pack.other.

Finally, jar files by themselves provide no visibility scoping. Public classes in implementation packages are visible to everyone not just classes in the jar file that contains the package.

OSGi components



This shows a schematic overview of the OSGi platform.

Leaving aside the hardware and operating system that of course sit at the bottom of the stack, the key piece of the pictures are bundles.

They are pervasive in the OSGi stack in that everything OSGi offers is in some way or other related to bundles.

To start with bundles define their execution environment (which defines minimum JDK/JVM level; typically this is only important for the embedded space).

Bundles are also the unit of modularity. They further have an associated life cycle and are the providers of services.

The OSGi service capability can be understood as essentially enabling a one-JVM Service-oriented Architecture. It helps to further decouple bundles and so enable scenarios like hot update. Essentially, the service registry is an integration layer for further decoupling bundles. Traditionally one obtains an implementation of a particular interface directly by instantiating the concrete class or using a factory (again directly). However, that means a client uses one particular implement out of one particular bundle's class space.

Instead for something like hot swapping, the client needs to be completely decoupled from any potential providers. The way to achieve this is the service registry. There are more details on this later in the module.

Finally, security is another pervasive aspect of OSGi. It builds on traditional Java 2 security but extends it to account for the dynamics inherent to the OSGi platform.

So why?

- Modularity
  - Encourages modularity in applications
  - Easy to administer, provision and build
  - Minimal coupling through service-based interaction
- Dynamics
  - Hot update [not supported on the feature pack as of release 1.0.0]

So why are people interested in OSGi?

For most use cases the main benefit of OSGi the enforced modularity, which helps the developer to structure applications more clearly and contain interdependencies. The declarative dependencies aspect also greatly simplifies building applications and provisioning them and administering them. Since modules declare their dependencies, these dependencies do not have to be included during application assembly since they can potentially be retrieved later.

Furthermore, updating modules can be accomplished on a per-module basis without having to isolate dependencies into their respective applications. The key messages here are: because of the version information, libraries can be shared without risk; and because of strong and semantic version information, libraries can be updated with reduced risk.

The dynamics aspects of OSGi are a lot less commonly used. One very interesting use case is that of hot update. With a properly structured application (probably built on a blueprint that will be introduced later) it is possible in OSGi to update parts of the application without any downtime. Note, though, that this is not currently supported in the feature pack.

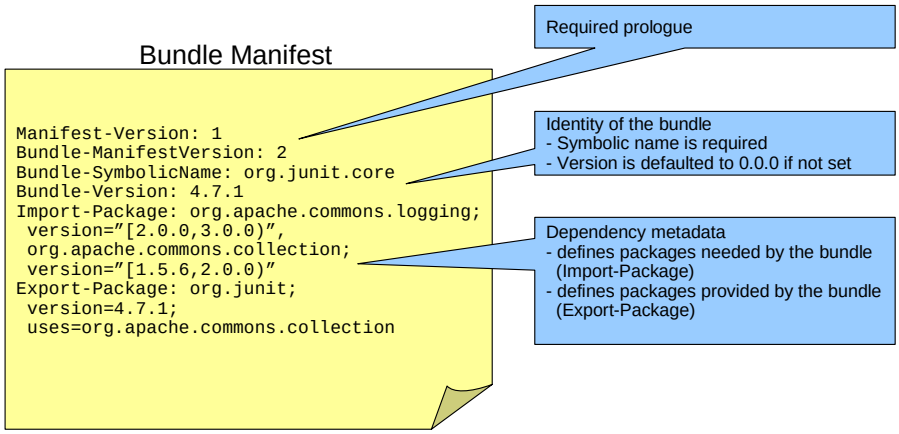
Section

OSGi basics

This section covers OSGi basics.

Bundles

- Jar files with additional metadata in META-INF/MANIFEST.MF



OSGi bundles at the very simplest are just jar files with a couple of additional entries in the normal jar manifest (called the bundle manifest in OSGi terminology).

What makes a bundle?

At the very least you need Bundle-ManifestVersion and Bundle-SymbolicName alongside the typical Manifest-Version (which is common to all manifests).

The top two tell the version of the manifest and the version of OSGi in use.

Then comes the identity of your bundle, the Bundle-SymbolicName and Bundle-Version. These lock down an identity for the content of the bundle.

After that come the packages, including versions that are imported and exported by this bundle. There will be more explanation of this later in this module.

OSGi versions

- Single version
  - for example Bundle-Version: 1.2.3.build_20100503
  - <major>.<minor>.<micro>.<qualifier>
    - *Major version change = breaking API change*
    - *Minor version change = compatible with earlier versions API change*
    - *Micro version change = bug fix, no API change*
    - *Qualifier = no semantics*
  - for example JPA 1 → 1.0.0, JPA 2 → 1.1.0 according to these semantics
- Version range
  - 1.0.0 = 1.0.0 or above
  - [1.0.0,2.0.0) = 1.0.0 up to but excluding 2.0.0
    - Square brackets are inclusive, round brackets are exclusive

A word on the versions in OSGi. The version scheme is similar to many other version schemes such as Mavens or jigsaws version handling.

An important piece of OSGi versions, which is not so commonly replicated in other schemes, is semantic version handling. A semantic version handling scheme associates particular meaning to individual versions and particular version increments. For OSGi a version comprises three meaningful parts, the major, minor, and micro parts and a free-form qualifier. The three first parts are numerical, whereas the qualifier can contain letters and certain punctuation symbols.

The semantics around the version parts are built around API changes where APIs are exported packages. A major version change in the version of a package denotes a breaking API change. Clients cannot typically support more than one different major version. A minor version change denotes API changes that are compatible with earlier versions. Finally, changes in the micro version represent bug fixes that include no change to the exported API.

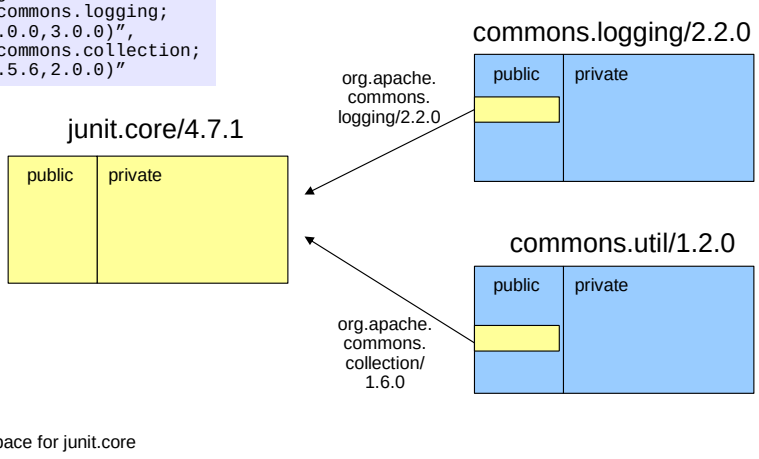
OSGi version schemes are much more strict than a lot other version handling schemes. In particular, JSR version does not obey the OSGi version handling rules. So as an example JPA 2 packages, which one might expect to have version 2.0.0, are actually versioned at 1.1.0 in OSGi environments because the JPA2 API is compatible with earlier versions.

In addition to individual versions, version ranges are an important concept in OSGi. OSGi supports opened ended and closed version ranges. Both ends of a closed range can be either exclusive (round bracket) or inclusive (square bracket)

Declarative dependencies (1 of 2)

- OSGi dependencies are primarily package-based

```
Import-Package:
 org.apache.commons.logging;
 version="[2.0.0,3.0.0)",
 org.apache.commons.collection;
 version="[1.5.6,2.0.0)"
```



OSGi dependencies are predominantly package based and best practice is to use only package based dependencies.

The Import-Package header contains a comma-separated list of packages to be imported. By default, a bundle can see nothing except java.* classes and javax.* classes that are part of the J2SE. Classes from any other package either need to be inside the bundle or imported. Otherwise, NoClassDefFoundErrors will occur at runtime.

Essentially, every bundle has its own class loader. As shown in this picture the OSGi dependency resolution constructs a class space, or class loader, for the junit.core bundle that contains exactly the classes it is allowed to see. Internal classes from the two other bundles and exported packages not needed by junit.core are not available to the junit.core bundle.

What this also means is that clients of the junit.core bundle are in general unaware of what imports junit.core used. So a bundle test.core can use a different version of org.apache.commons.logging without clash! So instead of a traditional hierarchical class loader structure like in JEE, OSGi bundles are linked by a networked class loader structure with tight visibility constraints.

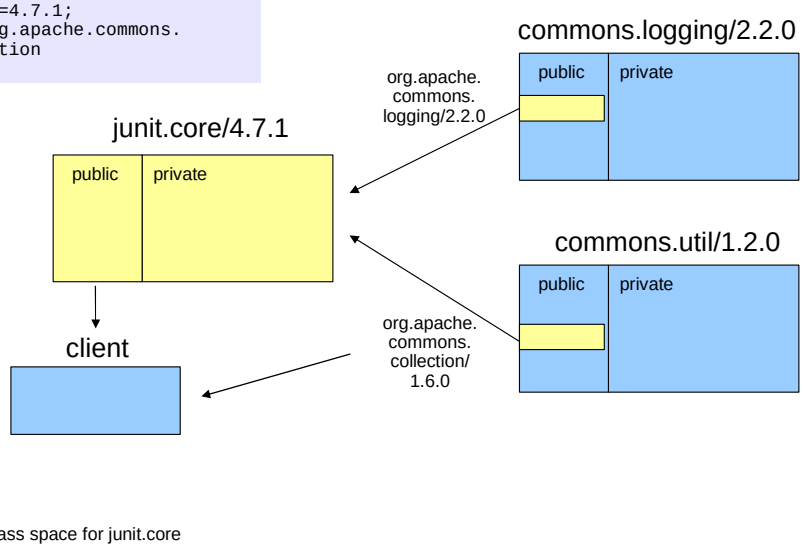
Also, note that package imports and package exports carry version information, including version ranges, of their own, which are independent of the bundle version as shown in the case commons.util bundle. These versions restrict the possible matches for import declarations. For example, the org.apache.commons.logging import would not be satisfied by an export of that bundle at version 1.0.0.

Note that if a package import has no associated version range, the implicit version range is “0.0.0” which means 0.0.0 or above.

OSGi allows for packages to be available optionally by specifying the directive “resolution:=optional” for a package. This would for example allow a bundle to optionally use different logging providers.

Declarative dependencies (2 of 2)

```
Export-Package: org.junit;
 version=4.7.1;
 uses=org.apache.commons.
  collection
```



Export-Package headers declare the packages that a bundle can supply to clients; that is, the bundle's public API.

The structure is very similar to the Import-Package header in that it contains a comma-separated list of packages each of which can have several attributes separated by semi-colons.

Most commonly a package export statement will contain a version, this gives the version of the package. Often this is the same as the version of the bundle, but that is not a requirement. If no version is specified – which is bad practice – OSGi defaults the version to "0.0.0" not the version of the bundle.

One slightly trickier concern about exporting packages is class space consistency. In order to guarantee a client can use the classes in a package, it needs to be guaranteed that the client also has the same view of any classes used by the former classes. So in this hypothetical example, since the classes in org.junit use org.apache.commons.collection in their visible API the uses attribute should be specified. This attribute forces the client to be wired to exactly the same package import for org.apache.commons.collection.

If this is not possible, say because the client expects org.apache.commons.collection in the range [1.7.0, 2.0.0), then the client can also not be wired to the org.junit package.

Other bundle manifest headers

- Bundle-Activator: org.junit.start.Activator
  - Call point to alert bundle of life cycle events
- Bundle-Classpath: WEB-INF/classes, WEB-INF/lib/spring.jar
  - Paths to load classes from inside the bundle, defaults to the root “.”
- Require-Bundle: commons.util;version=”[1.0.0,2.0.0)”
  - Declarative dependency for a whole bundle
- Fragment-Host: commons.util;version=”[1.0.0,2.0.0)”
  - Allows a bundle to append classes (and imports / exports) to another bundle as if they were part of it

There are four more commonly used manifest headers that bear mentioning.

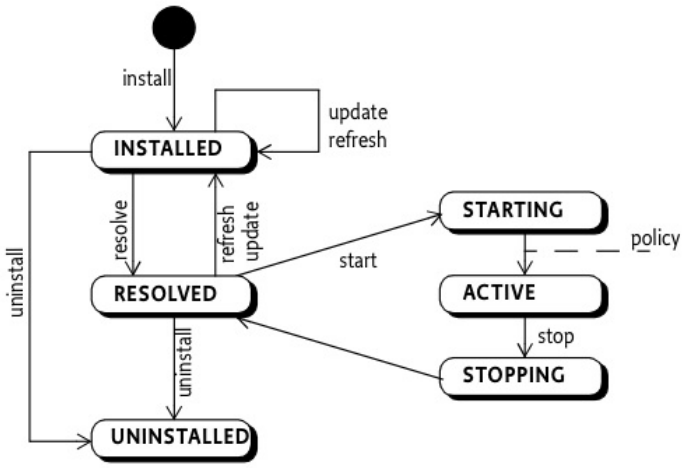
The Bundle-Activator defines a class in the bundle that is instantiated by the OSGi framework and receive “start” and “stop” life cycle events. This is essentially the only entry point to interact with the OSGi framework.

The Bundle-Classpath header defines the paths where classes can be found inside a bundle. This is important for bundles with non-standard class structure so as web bundles, which following WAR conventions have classpaths similar to the one shown. The classpath can include both directories and jar files inside the bundle.

The Require-Bundle header allows bundle-scoped dependencies to be declared. This breaks major parts of the OSGi dependency model and should only be used when package imports cannot work, such as split packages.

The Fragment-Host marks a bundle as a fragment. At runtime a fragment is attached to a non-fragment (host) bundle and acts essentially as if all the classes (or resources) in the fragment were contained inside the host bundle itself. A fragment can also add Import-Package and Export-Package entries to the host bundle, provided they do not conflict with the declarations of the host bundle. The main use case of fragments is to provide platform or environment classes and resources to a core bundle that is platform independent.

Bundle life cycle



In addition of modularity OSGi gives bundles a life of their own as opposed to the static nature of jar files. This means bundles are not permanently on the class path, they can come and go at any time.

Also, before bundles are ready to be used they go through several states:

In the INSTALLED state, a bundle is known to the OSGi framework

In the RESOLVED state, if all the dependency metadata for a bundle can be resolved it goes into resolved state

In the STARTING state, when a bundle is requested to start either explicitly or by a class loading request it first goes into starting state and from there into the active state (or stays in starting if the bundle is lazy and the start request allows laziness)

In the ACTIVE state, the bundle is running, by this time the bundle activator has been called

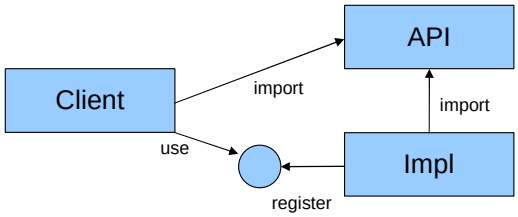
The STOPPING state, is similar to starting state when a bundle is to be stopped, it initially goes into stopping state and back to resolved

The UNINSTALLED state is the state where the bundle has been removed from the framework

In order to update a bundle, that is, to refresh the binary and the metadata, it has to be stopped and then refreshed - so the state transition is ACTIVE to STOPPING to RESOLVED to INSTALLED to RESOLVED and potentially then to STARTING and finally ACTIVE.

The service registry

- Additional layer of encapsulation and dynamics
- Service
  - An object associated with a list of classes (typically interfaces) it provides and properties
  - Dynamic (can come and go), framed by bundle life cycle
  - By default OSGi ensures class compatibility
  - Supports laziness



On top of the modularity afforded by declarative dependencies, OSGi provides another layer of encapsulation and dynamics, the service registry.

The basic idea is that bundles should be able to interact only through interfaces. In order to actually obtain a concrete implementation of an interface, a bundle goes to the service registry and looks up an interface either by the name of the interface, additional service properties, or a combination. OSGi will take care in that only services that use the same version of the interface's package are made available to the client bundle.

In addition to the decoupling provided by the service registry, services are dynamic - they can be registered and deregistered at any time. Also, a service is bound to the bundle that registered it; a bundle can only export services while it is in an active state. Those states are ACTIVE, STARTING and STOPPING.

What this means is that a client should obtain afresh, a service, whenever it needs to use it. Furthermore, the OSGi APIs allow clients to wait for services to become available. In total this allows a service provider bundle to be exported without stopping any other bundles and with only a temporary service outage.

This is the hot-swap scenario mentioned previously. It should be apparent that this is quite a complex scenario and imposes a great number of requirements on the bundles involved.

Section

Troubleshooting

This section covers troubleshooting.

Common OSGi problems

- No class def found
- Bundle does not start
  - Cause 1: bundle cannot be resolved
    - Dependency metadata is incorrect
  - Cause 2: start method on bundle activator failed
    - Probably still dependency metadata
  - **No free lunch** with package imports and export
- ClassCastException cannot convert <X> to <X>
  - The target class and the source object's class come from two different bundles, hence two different class loaders
- Service cannot be found
  - Service exporter and importer use different version of the package

There is a lot of documentation on getting started with OSGi, so this list claims by no means to be anywhere near exhaustive. However, these are some of the most common causes of problems that IBMs development and system test personnel have encountered while grappling with OSGi.

Probably the most common problem found is no class def found error which is typically caused by forgetting to import a needed package.

On the chart the first mentioned problem is bundles that do not start. This has two common causes. By far the most common is that the bundle cannot be resolved, for example it specifies “Import-Package: a” but there is no bundle that provides ‘a’. The bundle is not able to start as long as there is no one providing package ‘a’.

A less common problem, but all the more baffling, is a problem in the bundle activator. For example, the bundle activator class cannot be loaded because some classes it depends on are not visible, which is probably because the Import-Package statement is missing or incorrect. The important lesson here is that there is no free lunch as regards package imports. Everything other than java.* needs to be imported.

The second problem can occur when a package is provided and contained in multiple bundles. At some stage you might hit the seemingly nonsensical exception that X cannot be cast to X.

What this actually means that X coming out of class loader C1 cannot be cast to X coming out of class loader C2 because even though for all intents and purposes the class might be the same with the exact same byte code, classes are different if they are loaded by different class loaders, which happens by default in OSGi unless Export-Package and Import-Package are used.

The last problem is around OSGi services. Often enough a service that appears to be in the service registry, as inspected through some OSGi console, cannot be retrieved from a certain bundle. What this typically boils down to is a problem like the one just mentioned. The service exported has interface Y from bundle B1 whereas the service importer sees Y from bundle B2. Since those two incarnations of Y are not castable to each other, OSGi will by default (BundleContext#getServiceReference) hide the incompatible service. Hence, it is there but not there.

Further info

- OSGi specifications – in particular core specification
  - http://www.osgi.org/Download/Release4V42
- Two main implementations
  - http://www.eclipse.org/equinox
  - http://felix.apache.org
- Information center
  - http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/topic/com.ibm.websphere.osgifep.multiplatform.doc/topics/ca_intro.html
- Neil Bartlett's blog
  - http://njbartlett.name/blog/

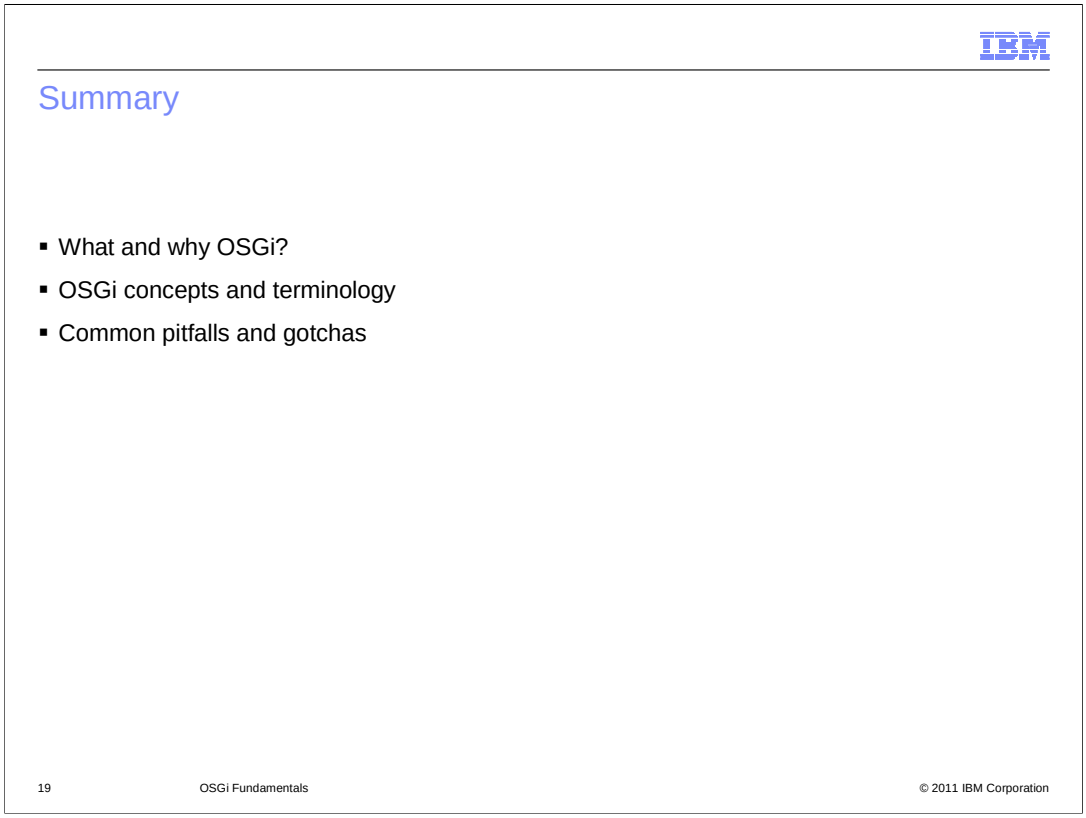
There are plenty of good information sources about OSGi on the web. There are four advisable for a start.

One is the OSGi home site. All the OSGi specifications are publicly accessible and of course there is no better authority.

The two open source implementations sites both have a wealth of interesting material around OSGi and of course tutorials.

The Information center for the OSGi applications feature pack contains a few introductory topics about OSGi, in particular OSGi in the enterprise, that are well worth reading.

Finally, in the OSGi community Neil Bartlett is known for his OSGi introductory lectures. On his blog, beside lots of interesting tidbits around OSGi, he has a very nice down to earth introductory book for OSGi (open source).



In summary, this module has presented an overview of what OSGi is and why customers are interested in it.

It presented basic concepts and terminology that are needed for understanding OSGi and are used in other modules on this topic. This included descriptions of bundles, manifests, dependencies, services, and the service registry.

Finally it has enumerated some of the common problems people can bump into when developing OSGi applications.

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_wasosgijpafep_OSGi_basics.ppt

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

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

Trademarks, disclaimer, and copyright information