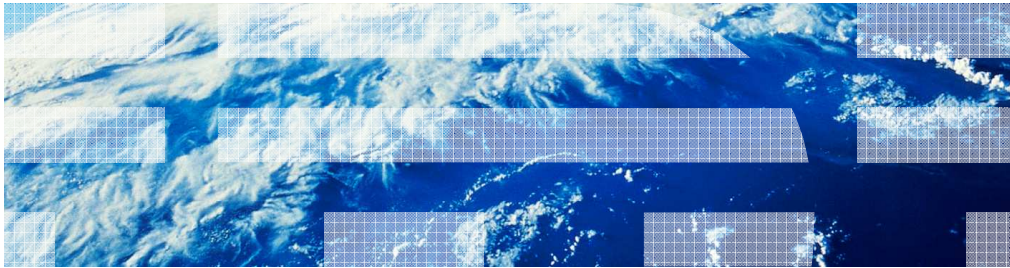


## IBM WebSphere Application Server Feature Pack for OSGi Applications and Java Persistence API 2.0



IBM WebSphere Application Server Feature Pack for OSGi Applications and Java Persistence API 2.0 OSGi applications in detail.

## Check out the information center

- This talk attempts to summarize and condense many new concepts.
- Check out the information center – available internally at <https://ut-ilnx-r4.hursley.ibm.com/websphere/aries/draft/index.jsp>
  - ▶ Feature Pack for OSGi Applications and JPA 2.0 (All operating systems)
  - ▶ “[Working with OSGi Applications](#)”
- See in particular
  - ▶ “[Learn about enterprise OSGi applications](#)”
    - “An introduction to OSGi Applications”
    - “OSGi bundles and bundle archives”
    - “Provisioning for OSGi applications”

The team has put a lot of work into making the Information Center articles educational and easy to use for this feature pack. Take advantage of their work. In addition to the kinds of information one normally finds in the feature pack is also a helloWorld sample which takes one through the creation and deployment of an OSGi application. The OSGi lab IBM Education Assistant module found at the same site as this module will extend that sample to take you through updating a bundle.

## Agenda (1 of 2)

- The OSGi application programming model
  - OSGi applications and the .eba file format
  - OSGi applications are isolated from each other, but their dependencies are shared
  - Provisioning an OSGi application
  - Updating an OSGi application
  - Migrating from .war and .ear formats
  - Advanced topics: Composite Bundles, Use-Bundle

Other modules have presented what OSGi is, what blueprint is and what a bundle is. This presentation will cover the OSGi application programming model. An OSGi application is a collection of bundles. What the feature pack gives you is the ability to define these collections of bundles and manage them as a collection.

You will cover the basic format of how these applications are structured and concepts of the programming model. Each application is isolated in that it's content is protected and applications do not overlap but applications can share dependencies so to take advantage of the OSGi capabilities for code reuse.

The provisioning process is how you get from a set of ranges of bundles to which specific bundles get into the runtime.

You will learn about updating applications, migration and some advanced topics, particularly information about class loading, the semantics of OSGi and some of the class loading things that OSGi lets you do that we have not been able to do without OSGi.

Finally some troubleshooting topics are covered.

## Agenda (2 of 2)

- The OSGi application programming model
  - **OSGi applications and the .eba file format**
  - OSGi applications are isolated from each other, but their dependencies are shared
  - Provisioning an OSGi application
  - Updating an OSGi application
  - Migrating from .war and .ear formats
  - Advanced topics: Composite Bundles, Use-Bundle

This section will discuss the OSGi application format.

## The Enterprise Bundle Application (EBA)

- An OSGi application, “Aries application” or “.eba” is a collection of OSGi bundles.
- Packaged as a zip-formatted file with a '.eba' suffix.
- Bundles can be 'contained' or 'by reference'.
- META-INF/APPLICATION.MF defines content and structure.
- OSGi applications are isolated from each other

The Feature Pack for OSGi applications introduces the concept of an OSGi application, being a collection of bundles. These applications are also known as “Aries applications”, after the Apache Aries project, or “EBAs”.

An EBA is packaged as a zip-formatted file with a '.eba' suffix.

Application bundles can be contained with the .eba, or provided 'by reference' – to be obtained from a bundle repository.

An application's content and structure is defined in its case-sensitive 'application manifest,' META-INF/APPLICATION.MF.

If this file is missing, the application's content is taken to be all the bundles contained in the .EBA's root directory.

OSGi applications are isolated from each other.

They cannot share packages.

Each runs in its own OSGi framework, with its own service registry.

## META-INF/APPLICATION.MF can contain version ranges

- Here's a sample application manifest from the information center:

```
Manifest-Version: 1.0
Application-ManifestVersion: 1.0
Application-Name: Example Blog
Application-SymbolicName: com.ibm.ws.eba.example.blog.app
Application-Version: 1.0
Application-Content:
  com.ibm.ws.eba.example.blog.api;version=1.0.0,
  com.ibm.ws.eba.example.blog.persistence;version="[1.0.0,1.1.0]",
  com.ibm.ws.eba.example.blog.web;version="[1.2.0,1.2.5)",
  com.ibm.ws.eba.example.blog;version="(1.2.0,2.0.0)"
Use-Bundle: com.ibm.json.java;version="[1.0.0,2.0.0)"
```

- Version range syntax is defined in the [OSGi Service Platform Core Specification version 4.2](#) section 3.2.6. For a version  $x$ ,
  - "1.0.0" means "1.0.0  $\leq$   $x$ "
  - "[1.0.0,1.0.0]" means "1.0.0 =  $x$ "
  - "[1.0.0,1.1.0]" means "1.0.0  $\leq$   $x$   $\leq$  1.1.0"
  - "[1.0.0,2.0.0)" means "1.0.0  $\leq$   $x$  < 2.0.0"
- By OSGi convention, the first digit is incremented on the introduction of an incompatible change

An OSGi application is defined as a collection of bundles, each of which can be at a specific version, or fall within a defined version range. Versions and version ranges were introduced in the OSGi fundamentals module.

All of these bundles can be installed at a range of versions. What this means is that the developer has set a range of versions for the bundles within which the administrator may replace the bundles with another that is within the specified range.

In this case the developer is saying that the blog API bundle needs to be at version 1 or higher but any version higher. The persistence bundle can be anything between 1.0 and 1.1.

The OSGi application developer uses an application manifest to define the broad structure and content of their application. Remember that bundles will often have three or four digit version numbers. A minor bug fix would increment the last digit, whereas a major, incompatible change would increment the first. The developer uses version ranges to indicate the extent to which a given bundle can safely be patched in the field, by an administrator, without further recourse to the development team.

In this example the application consists of five bundles, four of which are unique to this application and a fifth which may be shared among many applications.

## More on META-INF/APPLICATION.MF

- Key stanzas include:
  - ▶ Application-SymbolicName
  - ▶ Application-Content
  - ▶ Use-Bundle
- Mandatory headers are mandatory: Manifest-Version, Application-ManifestVersion, Application-SymbolicName, Application-Version, Application-Content
- Optional headers: Application-Name, Use-Bundle
- Optional headers covered in the talk on SCA integration: Application-ImportService, Application-ExportService

This file's stanzas are covered in detail in the information center. Key stanzas include `Application-SymbolicName` which is a unique name, `Application-Content` which is a comma separated list of bundles, and optionally versions or version ranges, that comprise the application's core content. An application must consist of at least one unique bundle.

`Use-Bundle` is a list of bundles that are to be used to satisfy package dependencies of bundles in `Application-Content`. This is only required in certain circumstances. `Use-Bundle` is discussed further later.

The slide also lists which headers are mandatory, and which optional. Two optional headers, `Application-ImportService` and `Application-ExportService` are covered in another module, on SCA integration. They are only used when the application interacts with the outside world.

## Turning version ranges into versions

- Application-Content can specify a range of versions against a given bundle's symbolic name.
- An application's content can change over time within those ranges.
- for example:
  - `Application-Content: my.bundle;version="[1.0.0,2.0.0]"`
- When an application is installed, the 'provisioning' process determines the exact bundle version to deploy for each version range specified in Application-Content

You have seen that the Application-Content header can specify a range of versions against a given bundle's symbolic name.

The application manifest's author can use version ranges to create an application whose content can change over time within those ranges.

For example, consider `Application-Content: my.bundle;version="[1.0.0,2.0.0]"`

The developer is here stating that an administrator can upgrade the version of my.bundle inside each installed copy of this application with another my.bundle whose version v is in the range  $1.0.0 \leq v < 2.0.0$ .

An EBA is installed as a WebSphere Application Server application asset and then added to one or more Business Level Applications.

When an asset is installed, specific versions of the bundles must be installed. The 'provisioning' process determines the exact bundle version to deploy for each version range specified in Application-Content.



## Bundle Repositories and Provisioning

- Application bundles 'may be obtained from a bundle repository.'
- A bundle repository records metadata about bundles, and can store the bundles themselves.
- Bundle metadata permits dependency analysis, an essential part of the programming model and provisioning process

It has been stated that application bundles 'may be obtained from a bundle repository.' A bundle repository is a web application that can serve up metadata about OSGi bundles. It can also be able to store and retrieve the bundles themselves. The OSGi bundle repository ("OBR") and its Java-based client API is defined in OSGi RFC 112.

A bundle repository must contain some XML which contains metadata about bundles, for the most part stripped out of the bundles' manifests. It records a bundle's name, version, imports and exports and where the bundle itself is.

It may also contain and quite likely does contain the bundles themselves.

OBR metadata allows you to record a bundle's "capabilities" - what it offers, and its "requirements" - what it needs. For example, each entry in the Export-Package header of a bundle's META-INF/MANIFEST.MF represents a "package capability" and each entry in the Import-Package header, a "package requirement."

A bundle may import a package from a bundle which imports two more packages from two other bundles, and so on. Such "dependency graphs" may be quite long.

OBR's Resolver interface allows you to ask the question, "given these starting bundle requirements (Application-Content's versions and version ranges), what bundles are needed, and at what versions, in order to have everything work at runtime?"

This dependency analysis, or "resolution" is vital to the operation of the provisioning process.

## Internal and external bundle repositories

- The feature pack provides a single 'internal' bundle repository.
  - Automated generation of metadata
- An administrator can define any number of 'external' bundle repositories.
  - Manage their own bundles and generate their own metadata
- Resolution is performed against all bundle repositories, plus any bundles contained within the .eba

The feature pack provides a single 'internal' bundle repository.

This allows an administrator to add and remove bundles. Bundles are modeled and the associated xml metadata generated automatically.

An administrator can define any number of 'external' bundle repositories. These are responsible for managing their own bundles and generating their own RFC 112-compliant xml metadata. It is not expected that many users will configure external bundle repositories. An external bundle repository is defined by providing the URL from which its xml metadata can be obtained.

Resolution for a given application is always performed against the combined set of all bundle repositories, internal and external, and any bundles contained within the .eba in question. .EBA's by-value bundles are only available to itself: other applications cannot resolve against them.

Every application is resolved against every repository defined, internal and external.

## Introducing DEPLOYMENT.MF

- A list of all the required bundles, and their exact versions necessary for the application to work.
- Defines the bundles to be loaded into the runtime each time the application starts.
- Important stanzas:
  - ▶ Deployed-Content
  - ▶ Provision-Bundle
  - ▶ Deployed-UseBundle

When an eba asset is imported, its Application-Content is resolved against any by-value bundles within the .eba, and the contents of the internal, and any external, bundle repositories.

If resolution succeeds, a list is generated of all the required bundles, and their exact versions necessary for the application to work. This information is stored in the DEPLOYMENT.MF file.

The bundles listed in DEPLOYMENT.MF is loaded into the runtime each time the application is started. This provides consistency from start to start rather than re-provisioning the application every time it is started.

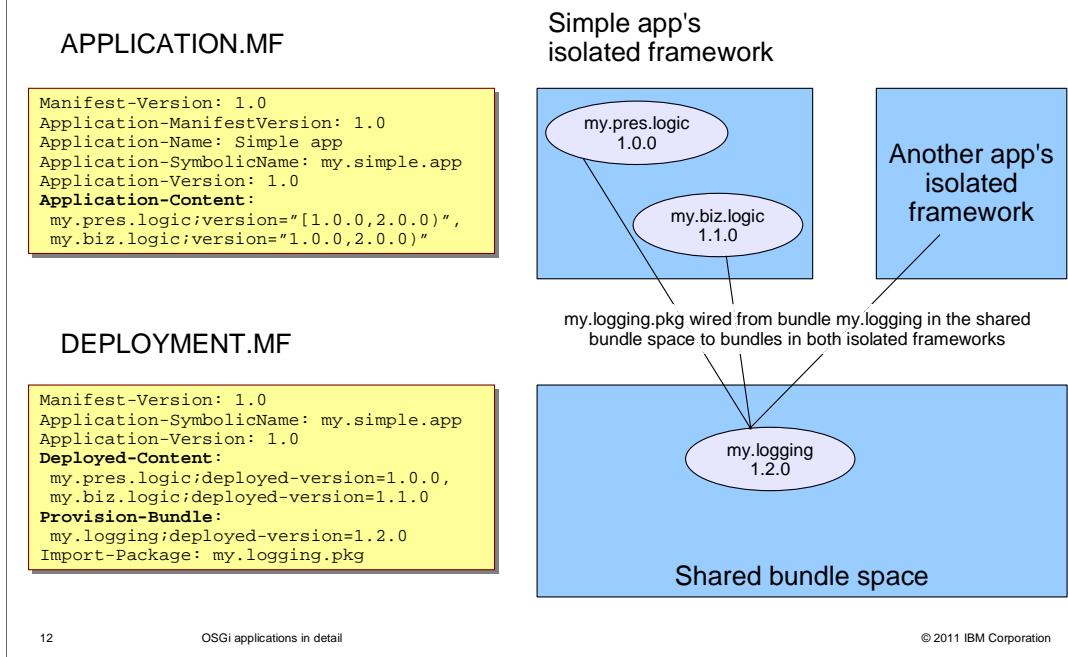
There are three important stanzas in this file.

The Deployed-Content stanza lists exact bundles and versions corresponding to core application content

The Provision-Bundle stanza lists bundles brought in by dependency analysis.

The Deployed-UseBundle stanza lists bundles called for by way of the Use-Bundle header in APPLICATION.MF.

## Core content is isolated. Dependencies are shared



12

OSGi applications in detail

© 2011 IBM Corporation

This picture illustrates several important concepts in the EBA programming model.

You can see here the isolation of core material and the sharing of dependencies, the definition of bundle version ranges and the provisioning of specific bundles into the runtime

For a given OSGi application, the bundles matching its core content are listed in the Application-Content stanza in APPLICATION.MF as ranges, and the Deployed-Content header of DEPLOYMENT.MF. For each OSGi application, these bundles run in their own isolated framework.

Every isolated application framework is a child of the single “shared bundle space” framework. There is one shared bundle space per application server and all the bundles in there are shared by all of the OSGi applications installed on that server. Every application's dependency bundles run in the shared bundle space. Applications can consume packages and services from the shared bundle space but cannot provide packages or services back to it.

## Agenda

- The OSGi application programming model
  - OSGi applications and the .eba file format
  - OSGi applications are isolated from each other, but their dependencies are shared
  - **Provisioning an OSGi application**
  - Updating an OSGi application
  - Migrating from .war and .ear formats
  - Advanced topics: Composite Bundles, Use-Bundle

This section will cover provisioning and OSGi application

## Provisioning an OSGi application

- APPLICATION.MF provides a list of starting requirements – one or more bundles, each with a given version or version range.
- Provisioning is the process of finding a set of bundles that fully meet those requirements, and the requirements of all the dependency bundles
- Requirements are calculated from:
  - Import-Package and Require-Bundle
  - <reference> and <reference-list>
- Matching capabilities are calculated from:
  - Bundle-SymbolicName and Bundle-Version
  - Export-Package
  - <service>
- Isolated bundles cannot provide bundles, packages or services to shared bundles.

APPLICATION.MF lists a set of starting requirements: one or more bundles, each possibly with a given version or range.

Provisioning is the process of finding a set of bundles that fully meet those requirements, and the requirements of all the dependency bundles.

Requirements are calculated from:

Import-Package and Require-Bundle statements in bundles' MANIFEST.MFs

<reference> and <reference-list> stanzas within bundles' blueprint xml files



Matching capabilities are calculated from:

Bundle-SymbolicName and Bundle-Version, and Export-Package statements in bundles' MANIFEST.MFs

<service> stanzas within bundles' blueprint xml files

Isolated bundles cannot provide bundles, packages or services to shared bundles. Any such 'circular dependency' will cause resolution to fail.

## Isolated/Shared is orthogonal to By Value / By Reference

Bundles are...	Provided by value - "Directly contained"	Provided by reference
<b>Isolated</b>	Contained in .eba; listed in Application-Content	Loaded from bundle repository. Listed in Application-Content
<b>Shared</b>	Contained in .eba. Not listed in Application-Content. Loaded into shared bundle space. <b>Other applications cannot resolve against these bundles, but may be affected by them at runtime.</b>  	Loaded from bundle repository. Not listed in Application-Content.

15

OSGi applications in detail

© 2011 IBM Corporation

This table illustrates the difference between (a) whether a bundle is contained in an .eba or obtained from a bundle repository, and (b) whether a bundle ends up being shared or isolated in the runtime.

Whether a bundle ends up being isolated or shared has nothing to do with how it was packaged and provided by the administrator. That is to say, whether it is provided directly in the EBA file or pulled in from a bundle repository has nothing to do with into which framework it ends up residing. It will end up being isolated if it is listed in the Application-Content stanza. They will end up in the shared framework if they are not in the Application-Content stanza.

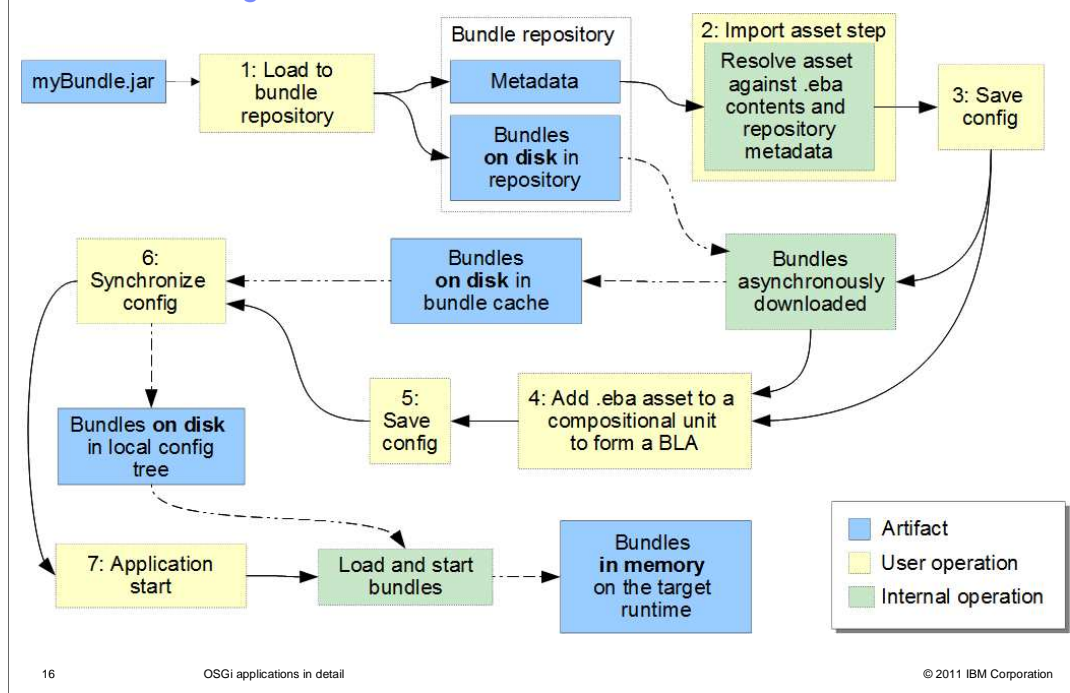
There's arguably another line to this table called "Not used." Bundles can be included in a .eba but not listed in Application-Content. They are 'Shared' if the Resolver determines them to be needed, that is to say that they satisfy one or more implicit dependencies. They are 'Not used' if the Resolver does not determine them to be needed. In this case they are not loaded into any runtime, but will be available to an administrator wanting to update the eba.

The point in the bold red text with warning symbols is that it can lead to unexpected behavior. For example, imagine that an application works correctly on a test system. Its DEPLOYMENT.MF can be exported, and the application imported into another, perhaps production environment, where it no longer behaves as expected. It is possible that the application was working correctly only because it was using a bundle 'smuggled into' the first runtime in this manner. Had the bundle come by way of a bundle repository, the application would have resolved against it, and carried that dependency correctly into the second environment.

Conversely, an application may fail to work correctly by picking up such a bundle – but its DEPLOYMENT.MF would not reflect the bundles to which it was being wired. One would have to inspect the runtime itself to understand what was happening.

Sometimes it may be convenient to include all a bundle's dependency within the .eba so that it can be guaranteed to work in the absence of a fully configured bundle repository. Such convenience carries this risk of 'polluting' the target runtime, so this approach should be used with caution.

## How bundles get to the runtime



This slide follows the progress of an application bundle from the internal bundle repository to being loaded into the runtime.

First, the bundle is loaded into the internal bundle repository, which stores both the bundle as a file on disk, and metadata about the bundle in the internal repository xml file.

Secondly, a .eba application asset is imported into WAS. The application is resolved against the original bundle's metadata, stored in the internal bundle repository. The bundle is determined to be required. This is when provisioning happens.

Thirdly, the administrator saves the newly imported asset. This triggers the download manager to download a copy of the bundle from the internal bundle repository to the 'bundle cache' which is on the dmgr.

Then the fourth step, once all the required bundles have been downloaded into the bundle cache, the administrator is able to add the .eba asset to a business level application, targeted to one or more WebSphere Application Server servers in the cell.

Fifth, the administrator saves the WAS configuration.

Sixth, once the WAS configuration is saved and all bundles have been downloaded into the dmgr's bundle cache, WebSphere Application Server configuration must be synchronized so that the bundle cache and newly created BLA is copied onto the target node. As a result of this, the original bundle now exists as a file in the configuration tree of the target server.

Finally, number seven shows that the administrator starts the business level application. This causes the local copy of the original bundle to be loaded into memory as a bundle running on the target server runtime.



## Agenda

- The OSGi application programming model
  - OSGi applications and the .eba file format
  - OSGi applications are isolated from each other, but their dependencies are shared
  - Provisioning an OSGi application
  - **Updating an OSGi application**
  - Migrating from .war and .ear formats
  - Advanced topics: Composite Bundles, Use-Bundle

This section covers updating an OSGi application.

## Updating an EBA

### Sample APPLICATION.MF

```
Manifest-Version: 1.0
Application-ManifestVersion: 1.0
Application-Name: Simple app
Application-SymbolicName: my.simple.app
Application-Version: 1.0
Application-Content:
my.pres.logic;version="[1.0.0,2.0.0)",
my.biz.logic;version="[1.0.0,2.0.0)"
Use-Bundle:
my.use.bundle;version="[1.0.0,1.1.0)"
```

- Application developer defines version ranges.
- Administrator can make changes within those ranges.
- Replacement bundles can come from the internal, or an external bundle repository, or those provided 'by value' within the original .eba.
- An updated application must be restarted for changes to take effect

### Sample DEPLOYMENT.MF

```
Manifest-Version: 1.0
Application-SymbolicName: my.simple.app
Application-Version: 1.0
Deployed-Content:
my.pres.logic;deployed-version=1.0.0,
my.biz.logic;deployed-version=1.1.0
Deployed-UseBundle:
my.use.bundle;deployed-version=1.0.1,
Provision-Bundle:
my.logging;deployed-version=1.2.0
Import-Package: my.logging.pkg,
my.pkg.from.use.bundle;version="1.0.1";b
undle-symbolic-
name=my.use.bundle;bundle-
version="1.0.1"
```

As stated earlier, “The application manifest's author can use version ranges to create an application whose content can change over time within those ranges.”

An administrator can replace a bundle listed in Deployed-Content or Deployed-UseBundle with another that falls within the ranges set in the corresponding entries in APPLICATION.MF.

They can choose replacement bundles from the internal, or an external bundle repository, or those provided 'by value' within the original .eba.

If the asset still resolves, each BLA that the asset is deployed into must be restarted for the changes to take effect.

Bundles listed in Provision-Bundle cannot be updated, and are not displayed to administrators, since they cannot be guaranteed to be used (sometimes called “wired”) at runtime.

## Agenda

- The OSGi application programming model
  - OSGi applications and the .eba file format
  - OSGi applications are isolated from each other, but their dependencies are shared
  - Provisioning an OSGi application
  - Updating an OSGi application
  - **Migrating from .war and .ear formats**
  - Advanced topics: Composite Bundles, Use-Bundle

This section will speak briefly about migration of non OSGi applications to become OSGi applications.

## Migration scenarios

- The Feature Pack provides [limited support](#) for the conversion of JEE applications.
- An .ear file can be renamed .eba and imported as before.
  - ▶ WARs are converted to WABs
  - ▶ Utility jars are not converted into bundles.
  - ▶ EJB jars are not converted: not supported in OSGi.
  - ▶ Security settings are migrated.
  - ▶ The application content will be taken to be the set of converted .war files.
- See the information center [article](#) on Spring migration

The Feature Pack provides limited support for the conversion of JEE applications.

.ear file can be renamed .eba and imported as before if and only if it contains WAR files and nothing else. If it contains anything other than WAR files there are either some manual steps or it cannot possibly be done.

Any WAR files are converted into OSGi-compliant web application bundles, WABs. (A WAB is a .war file with one or more additional manifest headers, much like a regular bundle is a .jar. A WAB remains a valid .war file.)

Utility jars are not converted into bundles.

EJB jars are not converted: EJBs do not work in the OSGi environment.

Any Java2 was.policy file is converted into a permissions.perm file, and all permissions promoted to the application level.

The application content will be taken to be the set of converted .war files.

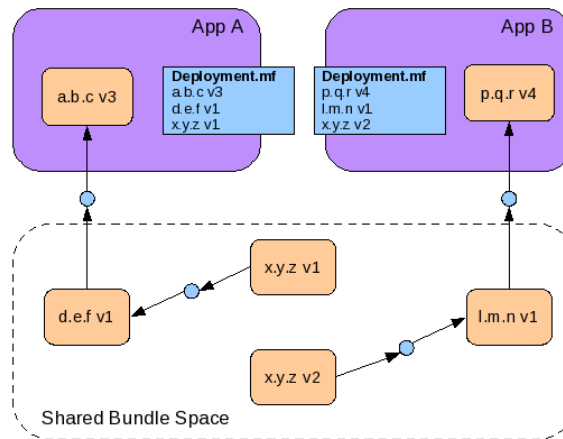
The information center contains an article that lays out the type of manual steps required to convert a Spring application into one that works with Blueprint.

## Agenda

- The OSGi application programming model
  - OSGi applications and the .eba file format
  - OSGi applications are isolated from each other, but their dependencies are shared
  - Provisioning an OSGi application
  - Updating an OSGi application
  - Migrating from .war and .ear formats
  - **Advanced topics: Composite Bundles, Use-Bundle**

Now for some advanced topics, mainly class loading, namely composite bundles and Use-Bundles.

## Package wiring and the need for composites



This is what one might expect to happen

Before covering what a composite bundle is, it must be understood why one is necessary.

A DEPLOYMENT.MF is supposed to insure that what happens tomorrow is what happened today unless you have upgraded your application.

This slide shows two applications each consists of one core bundle and two dependency bundles.

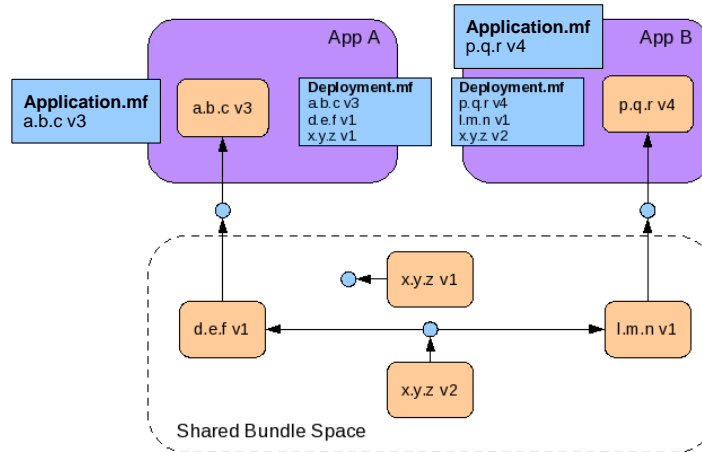
Application A is installed first and works as expected.

Bundle d.e.f contains an Import-Package: x.y.z;version="1.0.0" that is. 1.0.0 or higher.

A second application B is then installed, which causes bundle x.y.z v2 to be installed into the shared bundle space. This bundle exports package x.y.z at version 2.

Although you might expect d.e.f to be wired to x.y.z at version 1, that's not what happens in practice, as shown on the next slide.

## Bundles listed in Provision-Bundle get installed into the shared bundle space but cannot be wired at runtime



This is what will happen instead: d.e.f v1 gets wired to x.y.z v2 instead of x.y.z v1 as its DEPLOYMENT.MF can lead one to expect

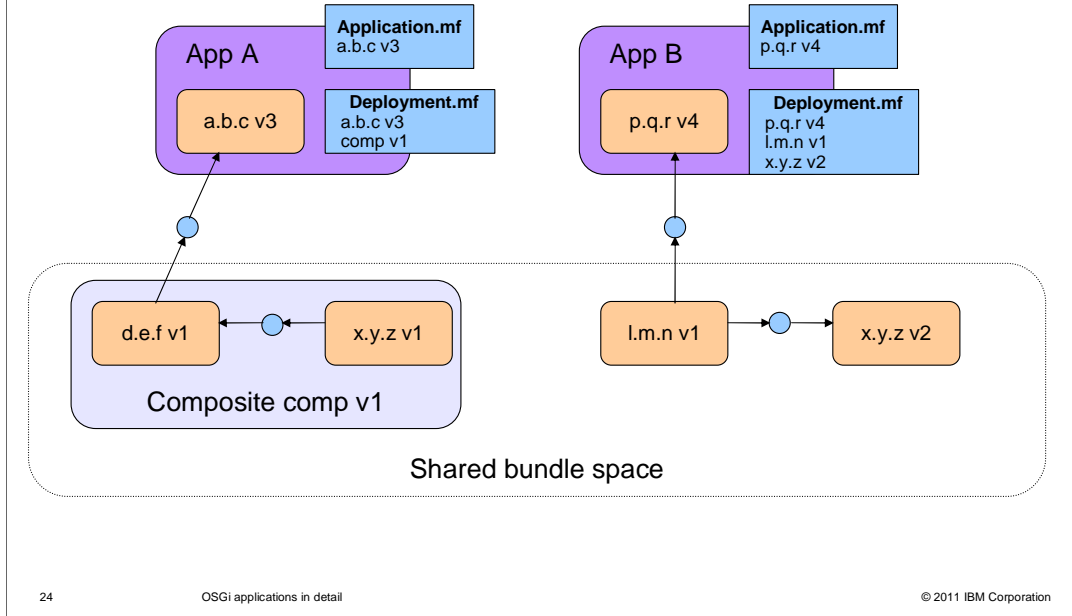
Because d.e.f v1 uses an unbounded import, it ends up perhaps unexpectedly wired to a different version of x.y.z. This can result in the application A behaving differently after application B is installed.

Due to the way OSGi works because the x.y.z d.e.f and l.m.n bundles are all loaded into the Shared Bundle Space which is a single framework. What happens is that once application B is installed x.y.z v2 is introduced into the Shared Bundle Space, since the DEPLOYMENT.MF of application A doesn't control what happens, application A ends up using dependencies introduced by application B. The introduction of application B here has disturbed application A and it can have different behavior as a result. You certainly do not want this.

In many cases this situation can be avoided by more carefully version handling d.e.f v1's Import-Package. But it can for some reason be necessary to consume d.e.f v1 'as is' – perhaps it comes from an external team or project, and it's not possible or desirable to edit its bundle manifest.

Composite bundles provide a mechanism to solve this type of problem.

An administrator can define a composite bundle to ensure the required package wiring



24

OSGi applications in detail

© 2011 IBM Corporation

In this example the administrator has defined a composite bundle named 'comp' which contains two regular bundles, `d.e.f v1` and `x.y.z v1`.

The composite, `comp`, is defined as exporting only one package, `d.e.f` at `v1`.

When application A is resolved, `comp` is added to its `DEPLOYMENT.MF`.

At runtime, `d.e.f v1` will only be able to see `x.y.z v1`, because the bundles making up a composite bundle run in their own isolated framework at runtime. In essence bundle `x.y.z` at version 1 has been forcibly wired to bundle `d.e.f v1` which is wired back to bundle `a.b.c`. In this way, application A is now undisturbed by the installation of application B.



## Composite Bundle Archives, CBAs

- Packaged as a zip-formatted file with a .cba extension.
- Content is defined in META-INF/COMPOSITEBUNDLE.MF.
- Comprises one or more bundles with exact versions.
- Unlike an EBA, a CBA can import and export packages.
- CBAs can have dependencies
  - these are resolved when an EBA that uses the CBA is provisioned.
- The provisioning process has a preference for composites

A CBA is packaged as a zip-formatted file with a .cba extension.

A CBA must contain a manifest, META-INF/COMPOSITEBUNDLE.MF.

A CBA is defined as containing one or more bundles with exact versions. All of these bundles must be present in the internal repository, or within the CBA, when the CBA is installed.

Unlike an EBA, a CBA can import and export packages.

A CBA is not resolved when installed: it is factored into the resolution process as any EBA that uses it.

All things being equal, the Resolver has a preference for composites. Taking the trouble to define a composite indicates that it should be used where possible.

## Sample META-INF/COMPOSITEBUNDLE.MF

```
Manifest-Version: 1.0
CompositeBundle-ManifestVersion: 1
Bundle-Name: Sample composite bundle
Bundle-SymbolicName: comp
Bundle-Version: 1.0
CompositeBundle-Content:
  d.e.f;version="[1.0.0,1.0.0]",
  x.y.z;version="[1.0.0,1.0.0]"
Export-Package: d.e.f;version=1.0.0
```

- Note how exact versions are specified.
- Author must specify both package imports and exports, and service import and export filters. for example
  - CompositeBundle-ExportService: my.exported.Service
  - CompositeBundle-ImportService: my.imported.Service;filter="( &(k=v) (k2=v2) )"

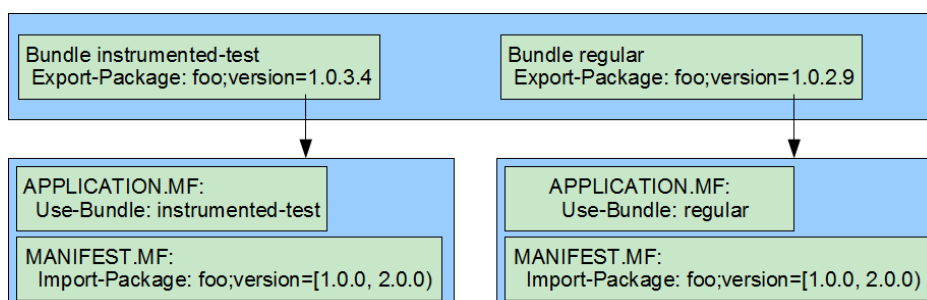
This slide shows a sample composite bundle manifest. Note that exact bundle versions must be specified as exact ranges, unlike the manner in which exact versions are specified in DEPLOYMENT.MF.

This composite comprises two bundles: d.e.f at version 1.0.0 and x.y.z, also at version 1.0.0. It exports one package, d.e.f, at version 1.0.0.

Service import filters are generated automatically for DEPLOYMENT.MF files but must be authored by hand for COMPOSITEBUNDLE.MF.

## Use-Bundle: generally not required, sometimes vital

- Used to enforce the particular shared bundle from which one or more required packages are imported into an isolated application.
- Useful when incrementally rolling out a fix to a shared library.
- Vital if a package is exported at a given version by more than one shared bundle



27

OSGi applications in detail

© 2011 IBM Corporation

It can sometimes occur that a package is exported at a given version or within a given range by more than one bundle in the shared bundle space. In these cases, it can be important for applications to influence how they are provisioned.

The Use-Bundle header in APPLICATION.MF permits an application to list bundles to be installed into the shared bundle space, from which packages are to be imported.

Entries are only written into the corresponding Deployed-ImportBundle header in DEPLOYMENT.MF if as a result of provisioning they are determined to provide at least one package to at least one bundle in Deployed-Content.

This example shows two versions of a bundle in a mixed environment: 'instrumented-test' is going through testing: it logs at a high level, and so runs slowly. A previous version, 'regular' is also in use. The teams have opted to leave the package import at [1.0.0, 2.0.0) and use the Use-Bundle stanza to control the bundle from which the package is consumed.

## Summary

- The OSGi application programming model
  - OSGi applications and the .eba file format
  - OSGi applications are isolated from each other, but their dependencies are shared
  - Provisioning an OSGi application
  - Updating an OSGi application
  - Migrating from .war and .ear formats
  - Moving an OSGi application from test to production
  - Advanced topics: Composite Bundles, Use-Bundle

This presentation has covered the OSGi application programming model, including how applications are structured - isolated from one another, even when sharing dependencies. You saw how the provisioning process gets a set of ranges from bundles, and which bundles get into the runtime. You learned about migrating from .war and .ear formats and finished up with some troubleshooting information.



## 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 apps in detail.ppt>

This module is also available in PDF format at: [../wasosgijpafep\\_OSGi\\_apps\\_in\\_detail.pdf](http://wasosgijpafep_OSGi_apps_in_detail.pdf)

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



## Trademarks, disclaimer, and copyright information

IBM, the IBM logo, ibm.com, and WebSphere are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of other IBM trademarks is available on the web at "[Copyright and trademark information](http://www.ibm.com/legal/copytrade.shtml)" at <http://www.ibm.com/legal/copytrade.shtml>

THE INFORMATION CONTAINED IN THIS PRESENTATION IS PROVIDED FOR INFORMATIONAL PURPOSES ONLY. in the United States, other countries, or both.

THE INFORMATION CONTAINED IN THIS PRESENTATION IS PROVIDED FOR INFORMATIONAL PURPOSES ONLY. WHILE EFFORTS WERE MADE TO VERIFY THE COMPLETENESS AND ACCURACY OF THE INFORMATION CONTAINED IN THIS PRESENTATION, IT IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED. IN ADDITION, THIS INFORMATION IS BASED ON IBM'S CURRENT PRODUCT PLANS AND STRATEGY, WHICH ARE SUBJECT TO CHANGE BY IBM WITHOUT NOTICE. IBM SHALL NOT BE RESPONSIBLE FOR ANY DAMAGES ARISING OUT OF THE USE OF, OR OTHERWISE RELATED TO, THIS PRESENTATION OR ANY OTHER DOCUMENTATION. NOTHING CONTAINED IN THIS PRESENTATION IS INTENDED TO, NOR SHALL HAVE THE EFFECT OF, CREATING ANY WARRANTIES OR REPRESENTATIONS FROM IBM (OR ITS SUPPLIERS OR LICENSORS), OR ALTERING THE TERMS AND CONDITIONS OF ANY AGREEMENT OR LICENSE GOVERNING THE USE OF IBM PRODUCTS OR SOFTWARE.

© Copyright International Business Machines Corporation 2010. All rights reserved.