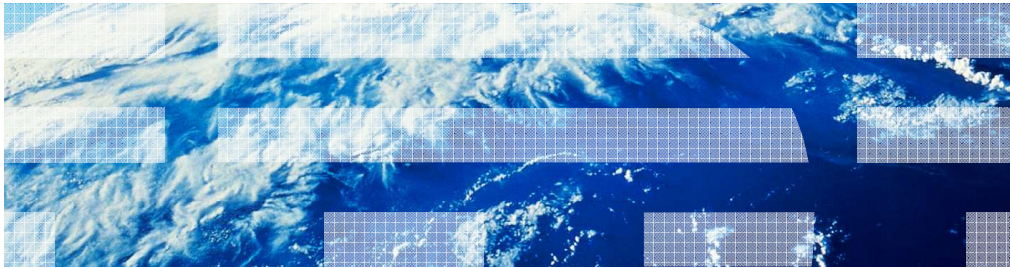


IBM WebSphere Application Server Feature Pack for OSGi Applications and Java Persistence API 2.0

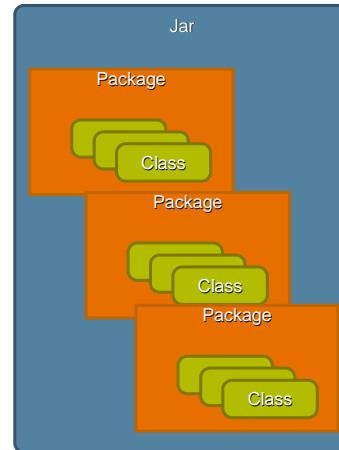
OSGi feature overview



This presentation is an introduction to the OSGi Applications feature of the IBM WebSphere® Applications Server Feature Pack for OSGi Applications and Java™ Persistence API 2.0.

Modularization in Java – Problems with jars

- Java platform modularity
 - Classes encapsulate data
 - Packages contain classes
 - Jars contain packages
- Class visibility:
 - private, package private, protected, public
- No “jar scoped” access modifiers
- No means for a jar to declare its dependencies
- No version handling
- Jars have no modularization characteristics
 - At runtime there is just a collection of classes on a global class path



In complex software engineering projects a properly modularized system enables: parallel development of modules by teams who need no understanding of the internals of other modules, reuse of modules by different applications, and maintenance of one module without affecting others.

Properly modularized systems are easier to maintain and extend.

Object oriented languages like Java help but their focus is on encapsulation of instance variables, they only help at the object and class level, and they offer no higher forms of modularity.

In particular JARs, as the module of deployment and therefore the ideal granularity at which to consider module reuse, have no modularity characteristics.

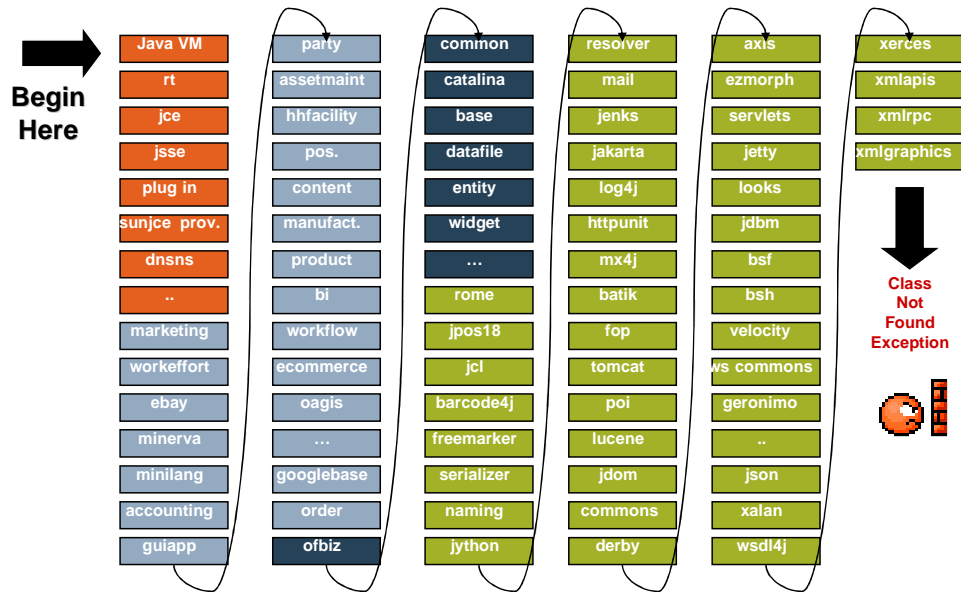
There is no “jar scoped” access modifier alongside public, protected and private.

Most JARs consist of multiple packages and, if the JAR represents a cohesive function, there is typically a need for classes in one package to access classes in another which then requires **public** accessibility. Immediately that makes these classes visible to classes in any other JAR. JARs provide no level of visibility control. Even well-behaved applications that only use the classes a jar-provider expects to be used externally are at the mercy of the global Java class path because the required class could be contained by multiple jars and the one loaded is the first on the global class path.

Not only do JARs lack the capacity to scope the visibility of what they contain, they also lack the capacity to declare their own dependencies. Many JARs have implicit dependencies on other JARs that means these JARs cannot be installed or moved around independently. If they are installed without dependencies being present then the first time there is any indication of a problem is at runtime.

Another problem with the global Java class path is its inability to accommodate multiple versions of a class. There can be multiple versions available on the class path but only the first will ever be loaded.

Problems with global Java class path



3

OSGi feature overview

© 2010 IBM Corporation

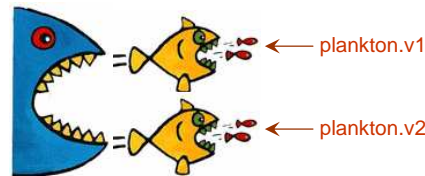
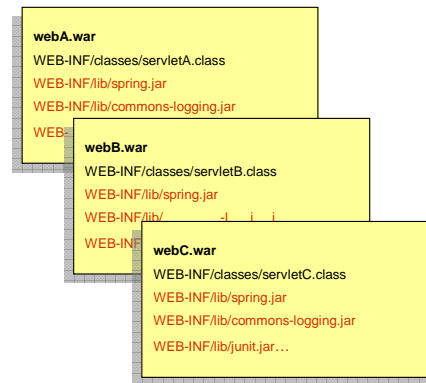
These last three problems are a consequence of the global Java classpath.

Java class-loading works by scanning the Java classpath and looking inside each jar in turn to find the required class. The choice of which jar a class is loaded from, and which version of a class is loaded, is entirely dependent on classpath order.

Classes whose dependencies cannot be resolved from the classpath are a potential problem; you do not find out until the “Class not found” exception at run time.

Problems with EARs/WARs

- Enterprise applications have isolated class paths but...
- *Across applications* - each archive typically contains all the libraries required by the application
 - Common libraries/frameworks get installed with each application
 - Multiple copies of libraries in memory
- *Within applications* – third party libraries consume other third party libraries leading to version conflicts



Java EE helps a little by isolating each enterprise application with its own class path. But even here there are some problems experienced in many Java EE deployments.

Enterprise applications often make use of third party Java libraries, either from open source or from an ISVs who provides the application. Common example are Apache Commons libraries, Spring, Hibernate and so on. The simplest way to ensure the coherency of each application is to include all the libraries each application needs in each EAR. While this makes it easier for EARs to be moved around from one system to another, it also makes for big ears and multiple copies of the same library in memory for each application.

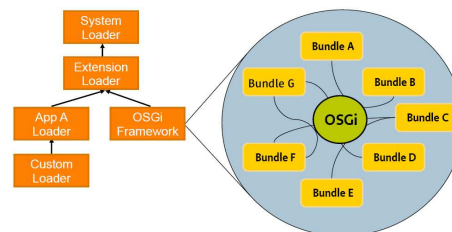
Within an application you can still only have one version of each class, which can easily become problematic when multiple third party libraries have dependencies on incompatible versions of some common utility class. “ObjectWeb ASM” is a good example of this kind of problem. ASM is a Java byte code manipulation and analysis framework used by many Java frameworks and has made non-compatible changes during its history. The effect of this is that if you have a web application using two frameworks both needing different versions of ASM then your application will not run even though it makes no direct use itself of ASM.

What is needed is a module system that can be used with Java to address these shortcomings and make it easier for Application Developers and Systems Administrators to build, deploy, and manage suites of applications consisting of reusable, versioned, modules. Which leads in to OSGi.

OSGi bundles and class loading

- OSGi bundle – A jar containing:
 - Classes and resources.
 - OSGi bundle manifest.
- What is in the manifest:
 - Bundle-Version: Multiple versions of bundles can live concurrently.
 - Import-Package: What packages from other bundles does this bundle depend upon?
 - Export-Package: What packages from this bundle are visible and reusable outside of the bundle?
- Class loading
 - Each bundle has its own loader.
 - No flat or monolithic classpath.
 - Class sharing and visibility decided by declarative dependencies, not by class loader hierarchies.
 - OSGi framework works out the dependencies including versions.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: MyService bundle
Bundle-SymbolicName: com.sample.myservice
Bundle-Version: 1.0.0
Bundle-Activator: com.sample.myservice.Activator
Import-Package: com.something.needed;version="1.1.2"
Export-Package: com.myservice.api;version="1.0.0"
```



5

OSGi feature overview

© 2010 IBM Corporation

OSGi defines a dynamic module for Java. It introduces some simple and yet powerful concepts to Java which eliminate each of the shortcomings we just discussed.

The key notion introduced is the “bundle” as the modular unit and the platform architecture is based upon bundles as the unit of deployment.

A bundle is just a JAR archive with a JAR manifest but the manifest contains additional OSGi metadata that is processed by the OSGi module layer. This metadata describes all the modularity aspects of the bundle.

Some notable metadata in the manifest are: the Bundle-Version: This is used to qualify the version of the bundle and enables multiple versions of the bundle to be concurrently active. The Import-Package header declares the external dependencies of the bundle that are used by the OSGi framework for bundle resolution. Specific versions or version ranges can be declared here. In the example, the imported package is required at version 1.1.2 or later. The Export-Package header declares the packages that are visible outside the bundle. Any package not declared here is only visible to classes inside the bundle.

Eclipse and Rational® Application Developer tools that are in a beta at this writing, as you will see later, provides a convenient GUI editor for this manifest.

How is this metadata exploited? It is used by the OSGi class loader. There is no global class path in OSGi – when bundles are installed into the OSGi framework their metadata is processed by the framework resolver and their declared external dependencies are reconciled against the versioned exports declared by the other installed bundles. The OSGi framework works out all the dependencies and calculates the independent class path for each bundle.

Each of the shortcomings of plain Java class loading are eliminated:

Firstly, only declared exports are visible outside the bundle

Secondly, dependencies are resolved to specific versions and multiple versions of packages can be available concurrently for different client bundles.

Finally, dependencies are explicit so that bundles will not start if all dependencies cannot be resolved

OSGi enterprise specification

- Released 22 March 2010
 - The product of the OSGi Enterprise Expert Group (EEG)
- Brings Enterprise technologies and OSGi together
- Using existing Java SE/EE specifications:
 - JTA, JPA, JNDI, JMX, web applications...
- Adds Spring-derived *Blueprint* component model and DI container
- Java EE provides the core enterprise application programming model
- Deploying modules as OSGi bundles simplifies reuse between applications, provides version handling, encourages (and enforces) modular design and enables dynamic module updates.

Having seen how OSGi can solve some of the common problems in Java and provide a modularity system for applications, we need to look at how this can be of practical benefit to applications developed for commercial enterprise Java runtimes like WebSphere. An obvious question to ask is – how does an enterprise Java application take advantage of any of these benefits? We have years of investment in Java EE with tools, runtimes and administrative processes to support it.

And this has been the primary concern of the OSGi Alliance Enterprise Expert Group, since 2007, and which released the first OSGi Enterprise Specification in March 2010. The Enterprise Expert Group is made up of platform and software vendors who have significant investment in Java EE, including IBM and Oracle in addition to SpringSource, Redhat and others. The resulting specification describes how Java SE/EE technologies like JTA, JPA, JNDI, JMX, and web applications run in an OSGi environment. There is no significant invention of new programming models, only the adaptation of what is already familiar into a more modular and dynamic runtime environment. The one extension beyond pure Java EE is the specification of the Blueprint component model and dependency injection container, an evolution of the Spring framework as an OSGi standard which we'll talk more about later.

Enterprise OSGi in Open Source

- Apache “Aries” created as a new Apache incubator project in Sep 2009:
 - to provide enterprise OSGi spec implementations
<http://incubator.apache.org/aries/>
 - to provide an environment to collaborate and experiment with new technologies to inform further EEG standardization.
 - In particular the programming model aspects of OSGi applications in an enterprise environment such as the Blueprint container and multi-bundle composites.
 - to build a broad development community to encourage implementation and adoption of EEG specs
- Aries components supporting an enterprise OSGi programming model are being integrated into both Geronimo and WebSphere Application Server.
 - Including Apache Felix Karaf, JBossOSGi, and others



Open source activities can often be a barometer on the success of new technologies and there are several open source projects with a focus on Enterprise OSGi. The most complete is the Apache Aries project, formed in September 2009. The objectives of Apache Aries are; to provide free, open source implementation of the enterprise OSGi technologies, to provide an environment to collaborate and experiment with new technologies to inform EEG standardization, in particular around those technologies that affect the application programming model such as the Blueprint container and multi-bundle composites, and to establish a broad and open community with an interest in enterprise OSGi to encourage implementation and adoption of OSGi in enterprise applications.

In seven months the Apache Aries project has grown to 43 contributors from companies including IBM, Progress, SAP, Redhat, Ericsson, LinkedIn, and others, including individual contributors.

Aries does not intend to provide a server runtime environment for enterprise OSGi but rather components that can be used in such an environment. The Apache Aries project provides implementations of Blueprint container, JPA integration, JTA integration, JMX integration, JNDI integration, Application assembly and deployment, Samples, documentation and an integrator's guide.

These have been integrated into Apache Geronimo and WebSphere Application Server as well as a number of other projects and products including Apache Felix Karaf, and JBossOSGi.

Application exploitation of OSGi in WebSphere

- OSGi has been used internally in WebSphere Application Server since V6.1 and in Eclipse since R3.
- Application-level exploitation is introduced in the **WebSphere Application Server Feature Pack for OSGi Applications and Java Persistence API (JPA) 2.0**
 - <http://www-01.ibm.com/software/webservers/appserv/was/featurepacks/>
 - Generally available May 2010
- Early Program available since Nov 2009
 - <https://www14.software.ibm.com/iwm/web/cc/earlyprograms/websphere/wasfposgiajp>
 - More downloads in a shorter period of time than any previous WebSphere Application Server V7 feature pack open beta program
- Two installable features:
 - OSGi Application feature simplifies the development, assembly, and deploy of enterprise applications
 - JPA 2.0 feature introduces Java EE 6 JPA 2.0 enhancements to object-relational persistence to simplify data access and optimize performance

WebSphere Application Server 6.1 introduced OSGi internally in 2005 and shipped in May 2006. IBM is now making it available for customer applications in a feature pack being released for Version 7.

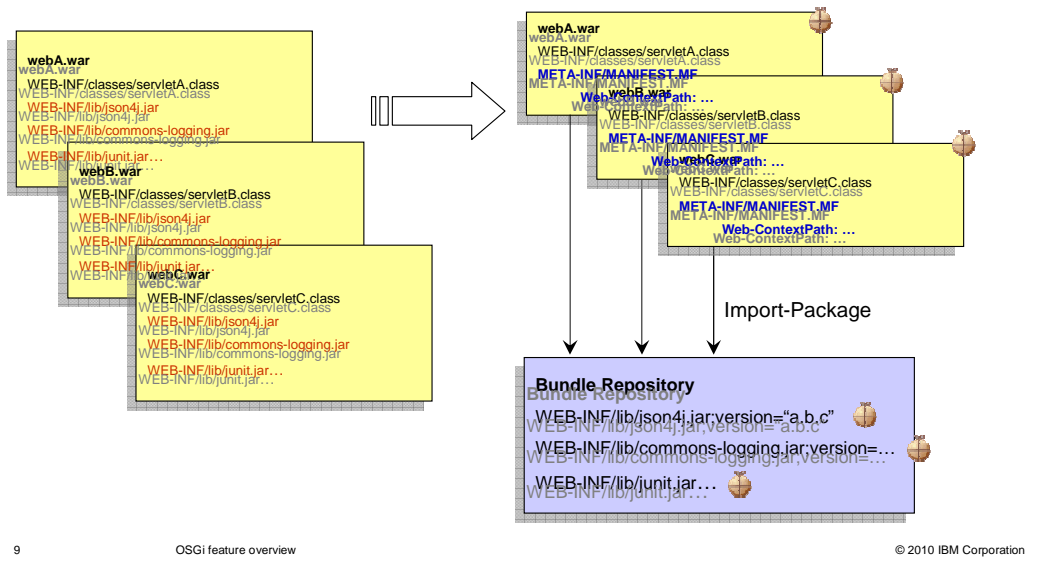
The feature pack integrates the Apache Aries and OpenJPA projects with the WebSphere Application Server and extends the Application Server to provide an end-to-end development, deployment and administrative integration for OSGi Applications.

The feature pack consists of two installable features that can be used separately or together. One introduces JPA 2.0 capability to WebSphere to further simplify and improve the performance of data access. The other feature is the OSGi Application feature which includes the Blueprint component model for POJO-based component assembly and deployment of applications as OSGi bundles. When used together, these features provide a simplified POJO-based component model, high-performance persistence framework and modular deployment system that simplifies the development and unit test of web applications. The OSGi application deployment model also greatly simplifies module reuse across applications.

Getting started: Bundlizing vanilla JEE

No Java code changes; war modules -> bundles

Common library jars may be easily factored out of the WARs and used at specific versions



9

OSGi feature overview

© 2010 IBM Corporation

Getting started with OSGi application support in WebSphere Application Server is made very simple with no need to make any changes to Web application implementation.

WAR modules can be deployed to WebSphere Application Server as web application bundles with no change of runtime behavior. On its own this is not all that interesting but it becomes interesting when you have multiple applications that use common libraries. What we can do now is place versioned, common libraries in an OSGi bundle repository so that each application using these libraries delivers only their unique modules.

Remember- a bundle is just a jar with additional OSGi metadata and a class loader which respects that metadata. Throughout this presentation a bundle symbol is used to indicate a jar that is actually a bundle. In the illustration here we can take three web application archives which include several common libraries and refactor these as three web application bundles containing only the unique content with the common libraries installed once into an OSGi bundle repository. It is the presence of the Web-ContextPath manifest header that causes the bundle to be recognized as a web application bundle.

New: Bundle repository configuration in WebSphere Application Server

The screenshot displays the 'Internal bundle repository' configuration page in the WebSphere Integrated Solutions Console. The left-hand navigation pane shows the following structure:

- View: All tasks
- Welcome
- Guided Activities
- Servers
- Applications
- Services
- Resources
- Security
- Environment
 - Virtual hosts
 - Update global Web server plus configuration
 - WebSphere variables
 - Shared libraries
 - Replication domains
 - OSGi bundle repositories** (highlighted with a red circle)
 - External bundle repository
 - Internal bundle repository
- System administration
- Users and Groups
- Monitoring and Tuning
- Troubleshooting
- Service integration
- UDDI

The main content area shows the configuration for the 'Internal bundle repository' for the bundle 'com.ibm.json.java'. The configuration includes the following fields:

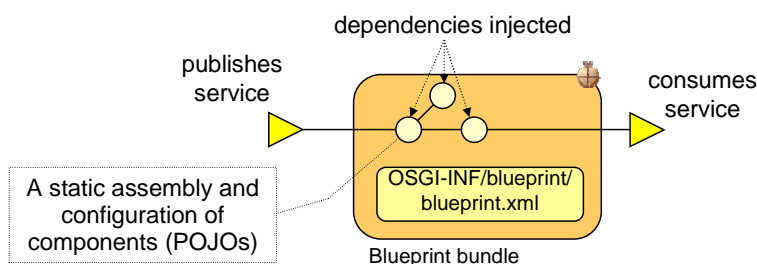
- Bundle symbolic name:** com.ibm.json.java
- Bundle version:** 1.0.0
- Bundle name:** JSON4J
- Bundle description:** (empty field)
- Imported packages:** (empty field)
- Exported packages:** com.ibm.json.java;version="1.0.0"

At the bottom of the page, there is a footer with the number '10', the text 'OSGi feature overview', and the copyright notice '© 2010 IBM Corporation'.

The WebSphere Application Server Feature Pack for OSGi Applications and Java Persistence API 2.0 introduces administrative support for using OSGi bundle repositories to simplify the deployment of applications which use common libraries. WebSphere Application Server can be configured either with the locations of external bundle repositories or can use the new internal bundle repository included with the feature pack. External bundle repositories provide their own tools for populating them and maintaining their content; the WebSphere Application Server internal OSGi bundle repository is managed by a WebSphere administrator using the administrative console or wsadmin scripting.

Common bundles can be installed once into the configured OSGi Bundle repository and used by many applications, reducing both disk usage and memory footprint.

Blueprint components and services



- Specifies a Dependency Injection container, standardizing established Spring conventions
- Configuration and dependencies declared in XML "module blueprint", which is a standardization of Spring "application context" XML.
 - Extended for OSGi: publishes and consumes components as OSGi services
- Simplifies unit test outside either Java EE or OSGi r/t.
- The Blueprint DI container is a part of the server runtime (compared to the Spring container which is part of the application.)

11

OSGi feature overview

© 2010 IBM Corporation

One of the significant features of the WebSphere OSGi Application feature is its introduction of the Blueprint component model. This is the result of the standardization activity around the Spring framework in the OSGi Alliance.

A sizeable portion of web applications use the Spring framework today for its POJO component model and dependency injection container. Spring provides a convenient way for business logic to be encapsulated into POJO components, which have all their dependencies injected into them by the Spring container. Since the POJO components have no Java dependencies on the application server it is very simple to unit test the business logic in a plain Java SE or Eclipse environment.

The Spring framework is a container that is packaged as a library with the application. In a Java EE environment, the Spring container delegates to the underlying application server for the management of resources such as database connections and for the application of qualities of service such as transactions and security. In a Java EE environment, Spring is essentially a proxy container to the native web container provided by the application server and can add a significant amount of path length.

By standardizing the Spring XML configuration format in the OSGi Alliance and delivering the container as an OSGi bundle, it has become possible to pull the dependency injection container out of the application and into the middleware. The standards-based evolution of the DI container is called the Blueprint container and the WebSphere OSGi feature pack integrates this container as part of the Application sever.

The Blueprint XML configuration file has the same structure as the Spring XML configuration file but in an OSGi namespace. The Blueprint XML is a bean definition file for all the beans provided by a single bundle. In addition to the bean definitions that are familiar to Spring developers, the Blueprint model adds new service and reference elements as part of the integration with the OSGi environment. Service elements direct the Blueprint container to expose a service interface for a component outside the bundle and a reference element directs the Blueprint container to locate a service that can be consumed from outside the bundle.

The yellow arrows in the figure indicate OSGi services that are published to and discovered from the OSGi service registry by the blueprint container. The OSGi service registry is a standard part of OSGi and provides a mechanism akin to JNDI for the publication of OSGi services, although the underlying use of the service registry is abstracted from application developers by the Blueprint container.

Ultimately, the Blueprint container manages the life cycle and dependencies of the POJO beans that contain the application logic in addition to the services and references each bundle provides, and ensures references are wired to available services.

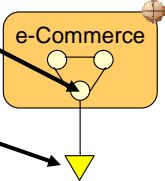
Exploiting blueprint components and Services (1 of 2)

e-Commerce bundle

```
<blueprint>
  <bean id="shop" class="org.example.ecomm.ShopImpl">
    <property name="billingService" ref="billingService" />
  </bean>
  <reference id="billingService"
    interface="org.example.bill.BillingService" />
</blueprint>
```

```
public class ShopImpl {
  private BillingService billingService;
  void setBillingService(BillingService srv) {
    billingService = srv;
  }

  void process(Order o) {
    billingService.bill(o);
  }
}
```

- 
- injected service reference
 - service can change over time
 - can be temporarily absent without the bundle caring
 - managed by Blueprint container

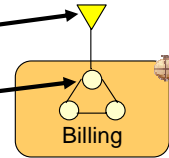
Here is a simple example of an eCommerce bundle which contains a “shop” bean which uses a BillingService service provided by another bundle. The Blueprint container locates the provider of BillingService and injects it into the shop bean at runtime.

OSGi services are dynamic so if the service provider can be changed then the blueprint container is able to dynamically rewire the reference to a new service without impacting the shop bean instance.

Exploiting blueprint components and Services (2 of 2)

Billing service bundle

```
<blueprint>
  <service ref="service" interface =
    "org.example.bill.BillingService" />
  <bean id="service" scope="prototype"
    class="org.example.bill.impl.BillingServiceImpl" />
</blueprint>
```



```
public interface BillingService {
    void bill(Order o);
}
```

- “prototype” scope indicates a new instance is created by the container for each use.
- “singleton” scope is the default.

On the provider side, the `BillingServiceImpl` is another POJO implementing a Java interface for which a service is registered by the Blueprint container when the Billing bundle is started.

By default beans are created with “singleton” scope which means only a single instance is created by the container. If the bean maintains any state then it can be declared with “prototype” scope so that the container creates a new instance each time it needs to inject the dependency into a client.

Blueprint persistence and transactions

- OpenJPA is default persistence provider in WebSphere
- Container managed JPA support integrated into Blueprint container:
 - @PersistenceUnit or @PersistenceContext (managed)
 - or <jpa:unit>, <jpa:context> bean property injection
 - Familiar development experience for JPA developers
 - Load-time enhancement of Entity classes
- Same container managed transaction attributes as EJBs:
 - Required, RequiresNew, Mandatory, NotSupported, Supports, Never

```
<blueprint>
  <bean id="shop" class="org.example.ecomm.ShopImpl">
    <jpa:context property="em" unitname="myUnit"/>
    <tx:transaction method="*" value="Required"/>
  </bean>
</blueprint>
```

The WebSphere Blueprint container does a lot more than the spring framework for managing JPA contexts. It understands standard JPA annotations in the blueprint components it manages and will inject an EntityManagerFactory or EntityManager into annotated components or components whose bean definition contains a “jpa” element as illustrated in the example here. Moreover, for the managed JPA case, the Blueprint container will manage the association between the EntityManager and the transaction context so the application does not have to. The WebSphere Blueprint container also provides full declarative transaction support using the same container-managed transaction attributes as EJBs.

You do not *need* to use Blueprint components in your OSGi applications just like you do not *need* Spring in Java EE, but the Blueprint container model provides significant ease of use benefits to developers, is based on an OSGi industry standard, is well-integrated with the server runtime, and has development tool support built into Rational Application Developer.

OSGi service registry and JNDI

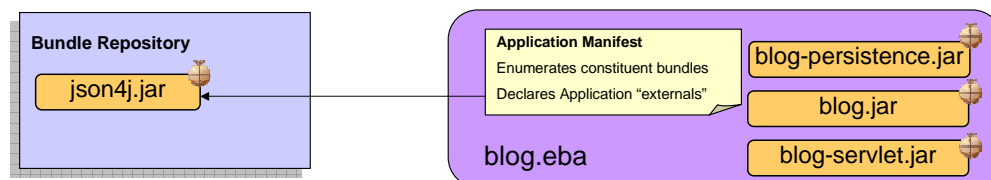
- OSGi services are published to and looked up from OSGi service registry.
 - From declarations in Blueprint XML
- Simplify integrating with existing JEE components:
 - OSGi Services registered in the OSGi Service Registry are also available in JNDI using the osgi:service URL scheme
 - Administered resources bound to JNDI are also published as services in the OSGi the Service Registry. The JNDI name is published as a service property called "osgi.jndi.service.name"

The OSGi Service Registry is a standard part of OSGi and is where services are registered by service providers for consumption by other bundles and how Blueprint services and reference use this mechanism. Existing web components are not aware of the OSGi service registry and use JNDI for service lookup. To enable existing Java EE mechanisms to interact with OSGi services, and vice versa, the Enterprise OSGi integration of JNDI defines a mechanism for OSGi services to be made available through JNDI and vice versa.

In the WebSphere OSGi Application feature, services published in the service registry are available to JNDI clients using the osgi:service URL scheme for the lookup. This is the primary method by which web applications discover Blueprint services. Equally, administered resources bound to JNDI are also published as services in the OSGi service registry with the JNDI name contained in a service property called osgi.jndi.service.name.

New: "Enterprise bundle archive" (EBA)

- An isolated, cohesive application consisting of a collection of bundles, is deployed as a logical unit in a ".eba" archive
 - An "OSGi Application"
- Constituent bundles may be contained ("by-value") or referenced from a bundle repository
- Services provided by the application are isolated to the application unless explicitly exposed through EBA-level application manifest
- Configuration by exception - absence of APPLICATION.MF means:
 - application content is the set of bundles contained by-value plus any repository-hosted dependencies identified during deployment



16

OSGi feature overview

© 2010 IBM Corporation

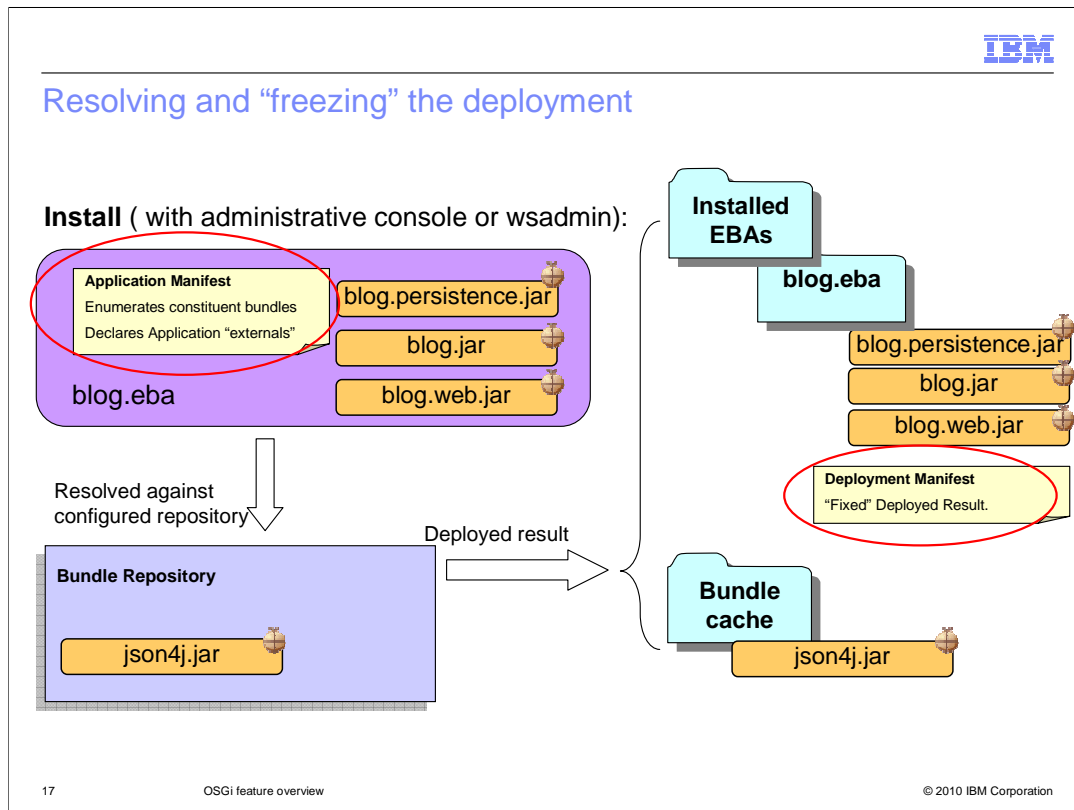
OSGi applications are deployed to WebSphere Application Server through wsadmin or the administrative console just like any other application, but are packaged in a new type of archive called an "enterprise bundle archive" or "EBA" archive. This is similar to an EAR but its modules are deployed as bundles to the required target servers.

An EBA archive represents a single isolated OSGi application consisting of one or more modules and is the unit of deployment for an enterprise OSGi application. Like an EAR file, an EBA archive may contain all the constituent modules/bundles that make up the application but it may just contain the metadata required to locate those bundles from a configured bundle repository. The metadata is in the form of an EBA-level "APPLICATION MANIFEST" file that describes the content of the application and whether the application exposes any external services and references. Just like a bundle manifest describes the modularity characteristics of a bundle, the application manifest describes the modularity characteristics of the application and the deployable content of the application.

The example here shows an OSGi Application packaged in a blog.eba archive, which contains three application bundles, and an APPLICATION MANIFEST, which refers to a fourth "json4j.jar" bundle. When the application is deployed, the json4j.jar is obtained from the configured bundle repository and does not have to be packaged inside the EBA.

Configuration is by exception – the absence of an APPLICATION MANIFEST indicates that all application content is contained within the archive and the application exposes no services or references externally.

Resolving and “freezing” the deployment



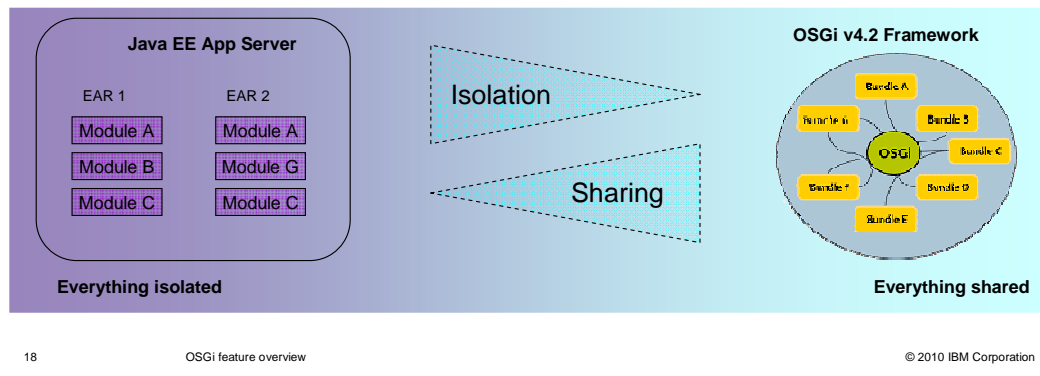
When present, the Application.mf is authored by the assembler (for example using Rational Application Developer). It does not need to contain a transitively-closed list of all the bundle dependencies determined for the application – these are figured out at deployment time and any missing package dependencies are reported as errors. Any resolution errors are caught early at deployment time – if the application deploys successfully then it should always be able to fully resolve and should never fail at runtime with a ClassNotFoundException.

Once an OSGi application has been successfully deployed then all its constituent bundles are pushed out to the appropriate target servers and the application can be administratively started. Starting the application causes its constituent bundles to go through the OSGi life cycle states “installed”, “resolved” and “active”.

Since other applications, deployed at a later date to the same servers could contribute shared bundles whose Java packages influence the dependency resolution that occurs when the application is started, WebSphere generates a “deployment manifest” that “freezes” the deployment of an application. This means that each time the application is restarted, the resolution process always calculates the same result regardless of other applications. Unlike the authored application manifest, the generated deployment manifest does contain the transitively closed content – the result of the deploy-time resolution. The deployment manifest is the description of the deployed application. This can be exported from one deployment to another to ensure that an application moving from a test system to a production system continues to resolve in exactly the same way it did during testing.

Details: Isolated and shared Bundles (1 of 2)

- In Java EE, modules are isolated within an application and applications are isolated from one another.
 - Makes sharing modules difficult
- OSGi 4.2 all bundles have shared visibility to the externals of all others bundles within an OSGi framework (JVM)
 - Makes isolating applications difficult

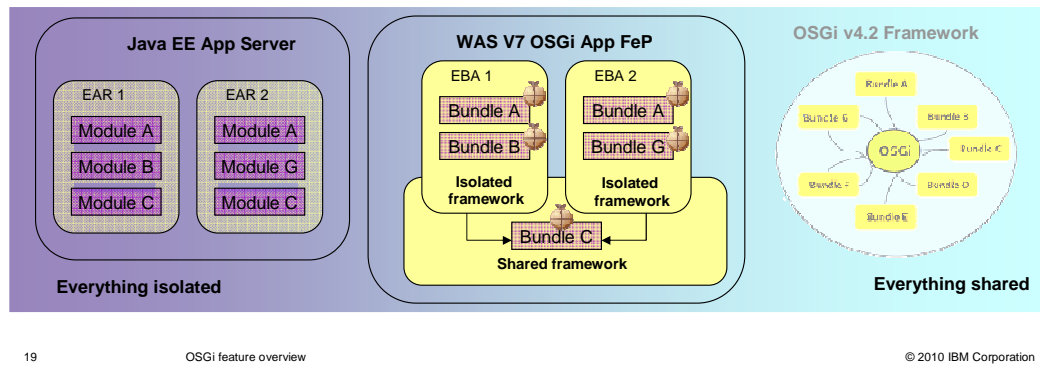


Application isolation is an important consideration. At one extreme, Java EE provides no portable notion of module sharing between enterprise applications – everything is isolated and sharing libraries is difficult. At the other extreme, OSGi bundles have shared visibility to the externals of all other bundles within an OSGi framework, which typically means within a JVM. This makes isolating applications difficult.

Something in between is required.

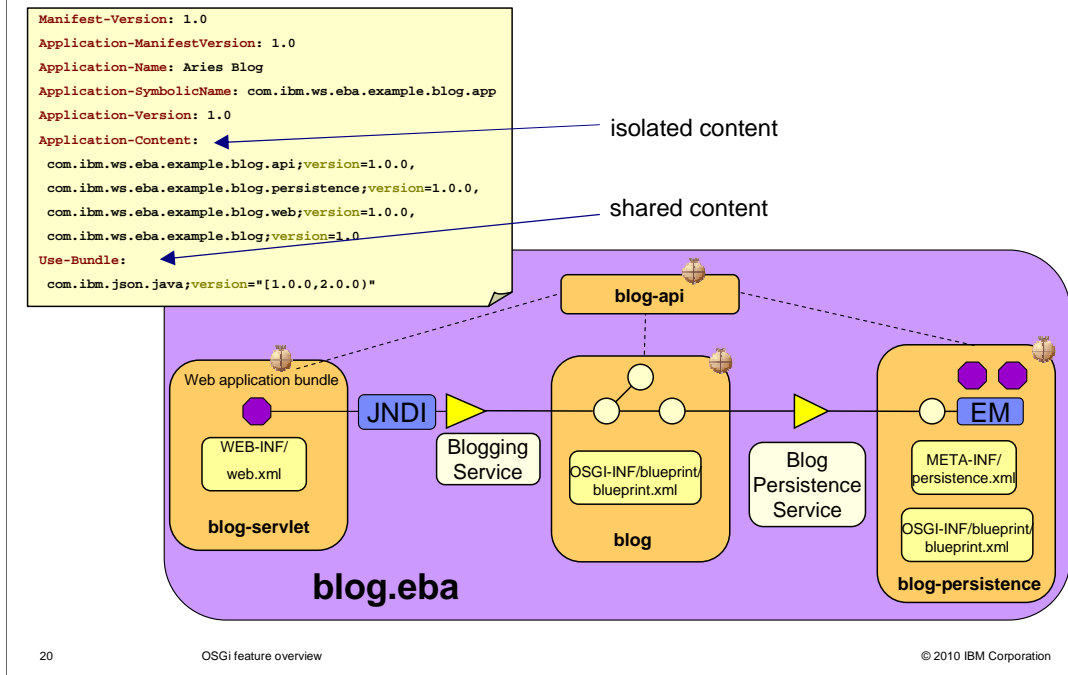
Details: Isolated and shared Bundles (2 of 2)

- Equinox 3.5 “nested framework” support enables “composite bundles” to run in isolated child frameworks
 - WebSphere installs each OSGi Application into an isolated child framework
 - Shared bundles are installed into the (single) parent framework



The current version of Equinox, which is used inside WebSphere Application Server and is the reference implementation of OSGi 4.2, supports the notion of nested frameworks whereby multiple peer frameworks are isolated from one another but may share a common parent framework. The WebSphere OSGi Application feature exploits this by deploying the isolated content of each OSGi Application into its own isolated child framework. Any libraries deployed from the OSGi bundle repository that are intended to be shared between applications are deployed into the single parent framework.

Example "Blog" application architecture



The APPLICATION MANIFEST here shows how isolated content is defined by the Application-Content header and how shared content can be defined by the Use-Bundle header.

The figure describes one of the sample applications shipped with the OSGi Application feature pack. It is a web application that provides a Blog.

The sample application consists of four bundles.

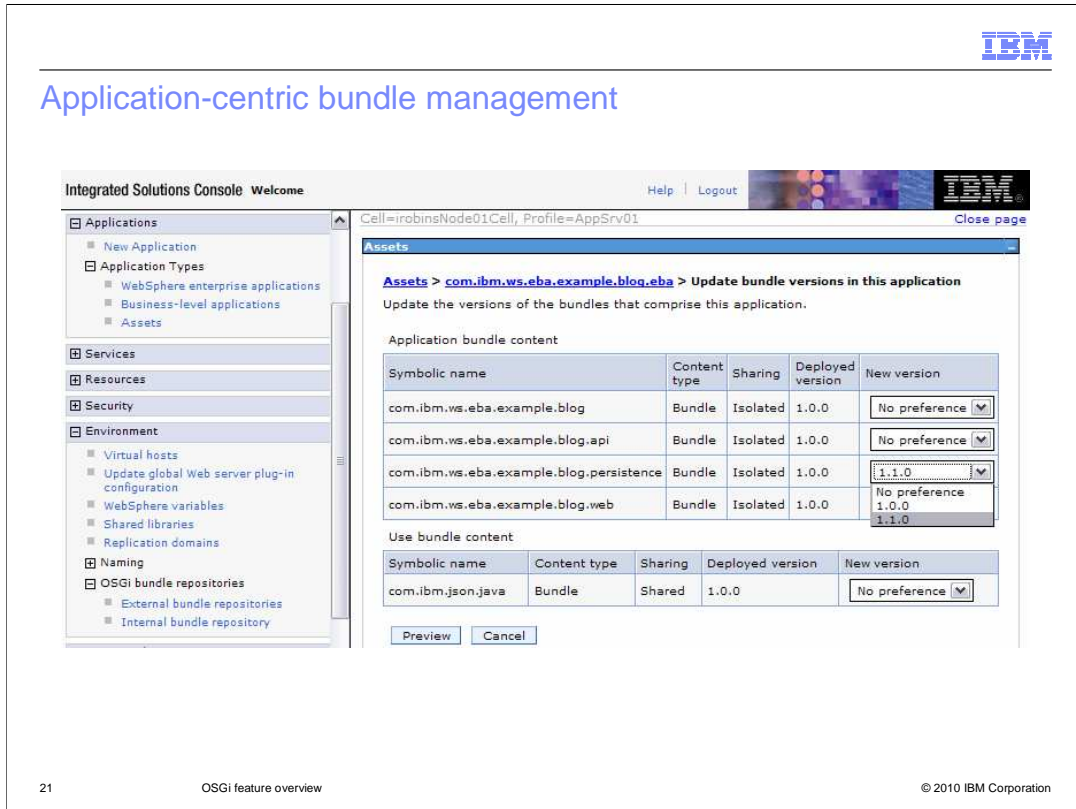
It has a web bundle to provide the user interface using standard servlets and dojo elements.

It has a blueprint bundle containing three beans which encapsulate the business logic. The entry point is a Bloggng Service which is accessed through JNDI by the web application.

It has a persistence bundle containing a standard persistence.xml and entities representing the persistent data.

Finally, it has a database where blog entries and author information are read from and written to through JPA.

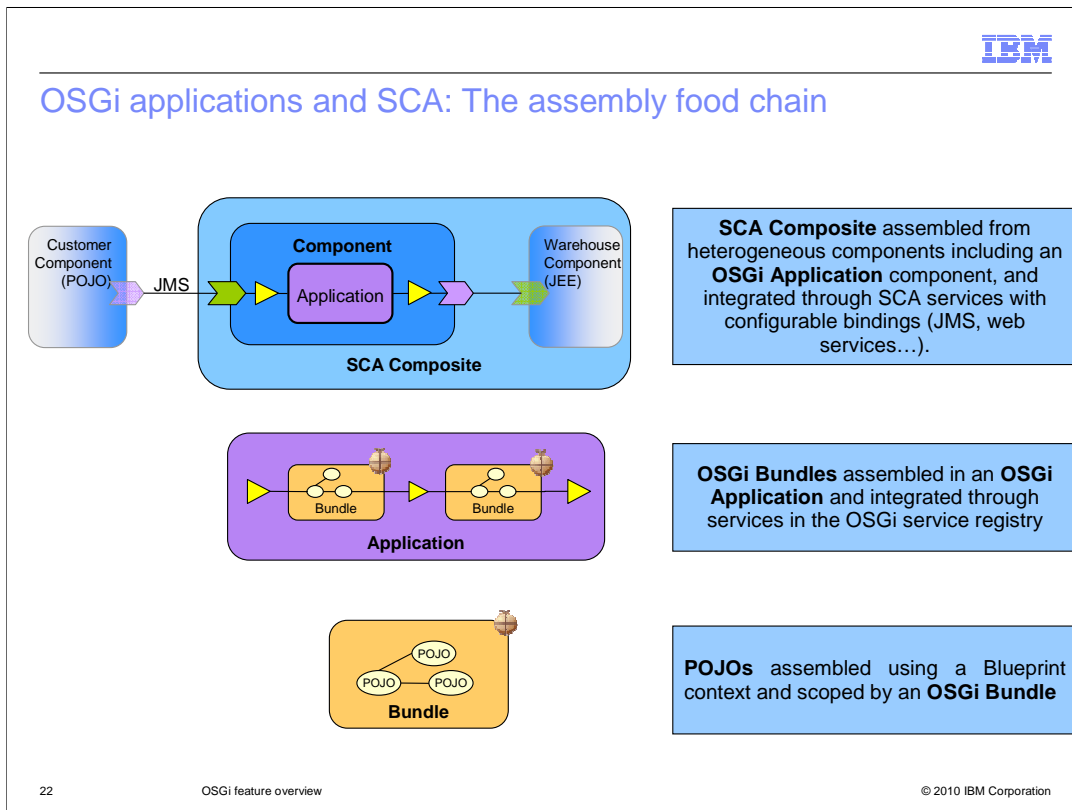
The brief demonstration shows how this OSGi application is deployed as a set of bundles to WebSphere Application Server. It also illustrates the isolated and shared frameworks and the placement of bundles in each.



In the WebSphere Application Server administrative console, you can see the version of each bundle used by the application. Initially these are the only versions of the bundles available.

To update one of the bundles in this application, replace the persistence bundle with a 1.1 version. You can do this by adding version 1.1 of the bundle to the bundle repository. The administrator who looks at the available bundles will now see that the persistence bundle is available at versions 1.0 and 1.1. If the administrator wants to move to version 1.1 he can preview the changes to make sure the new version would still enable the application to be fully resolved. In this case it is a safe change to make, so the administrator goes ahead and commits the change. The next time the application is restarted it will use version 1.1 of the persistence bundle.

OSGi applications and SCA: The assembly food chain

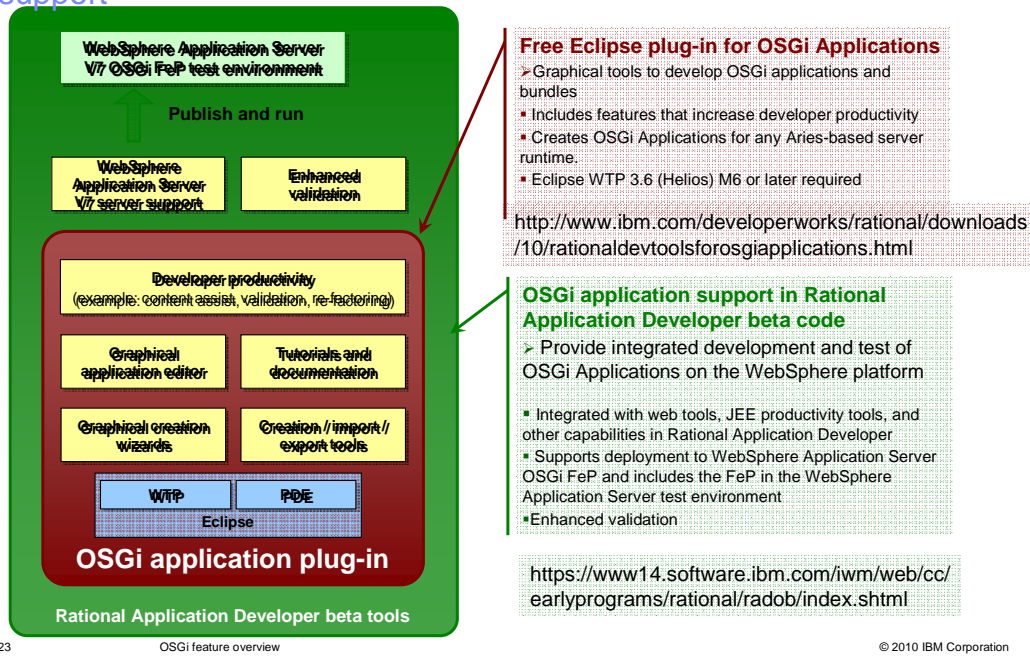


You have seen how POJO beans can be assembled and configured in a blueprint bundle and how multiple bundles - including web and persistence bundles - can be assembled into an isolated OSGi Application.

There is a further level of assembly available to OSGi applications into an SCA composite to provide an SOA abstraction. Within an SCA composite the OSGi Application is a component that can be wired to other components with different implementation types. For example, an SCA composite could contain an osgi-application component, a JEE component containing EJBs, a BPEL component, and so on. Each component within an SCA composite declares abstract services and references to which concrete bindings can be applied and it is through these services and references that the components of an SCA composite are wired together. The OSGi Application architecture was designed with this form of assembly in mind so that the services and references declared in a blueprint XML configuration can be exposed through the application manifest to be visible outside the application. Such exposed services and references can then be mapped to SCA services and references with the full range of available SCA bindings applied to them.

This enables OSGi applications to participate in two new scenarios: the assembly into heterogeneous composites of OSGi and non-OSGi components and remote use of OSGi application services through SCA services with a variety of bindings including JMS, SOAP/HTTP, IIOP and JSON-RPC.

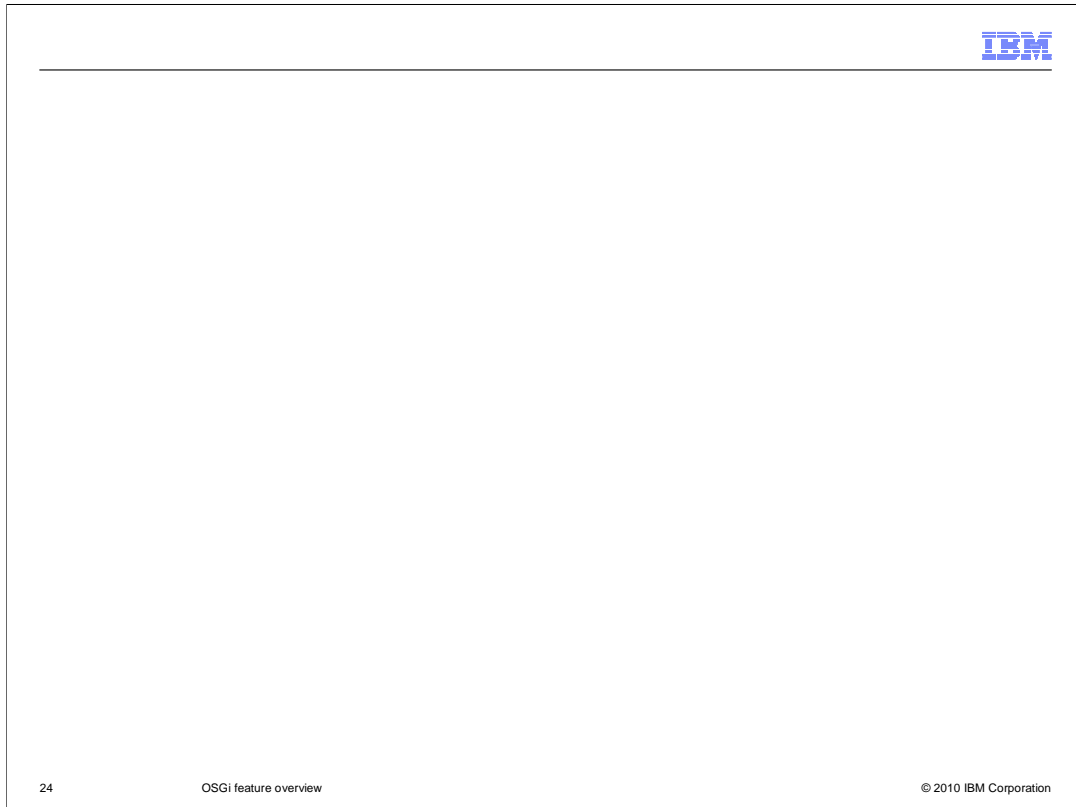
Rational Application Developer beta OSGi application project support



Tool support for OSGi Applications is available in the Rational Application Developer beta code that was published on April 16, 2010. The new tools are structured so that server-independent development and assembly tools can be installed as a plug-in into any Eclipse WTP 3.6 environment. While this is pre-integrated in the Rational Application Developer beta code, the availability of the new tools in Eclipse WTP configurations other than Rational Application Developer better enables these common tools to be used to develop OSGi Applications for deployment to Geronimo and, in the future, other non-WebSphere Application Server servers that integrate the Apache Aries runtime components.

The common development tools include new project type for OSGi Applications, the ability to import and export .eba archives, form-based editors for bundle manifests, application manifests and Blueprint configuration files in addition to tutorials and documentation.

Additionally integrated into the Rational Application Developer beta code are WebSphere deployment tools, a WebSphere Application Server test environment augmented with the OSGi application pack, enhanced validation tools, and integration with web and JEE productivity tools.



In summary you have seen some of the common problems experienced with Enterprise Java.

You have seen how the design, development, and maintenance of complex applications can be improved and simplified by a model that enforced real modularity at the level of the unit of deployment. When the coherent, reusable module is the thing that is actually deployed, rather than something within it like a class, then the barriers to module reuse are lowered as far as they can go, improving the likelihood of really achieving reuse and therefore reducing cost. The feature pack delivers support for modular OSGi applications, provides a fully integrated administrative model to support their deployment and management and, with the Rational Application Developer beta code, provides tools to improve developer productivity.

You have seen how sharing common libraries between isolated Java EE applications is greatly simplified by the integration of a bundle repository for common libraries with the application deployment process.

You have seen how multiple versions of classes can be accommodated within and across applications through OSGi bundles versioning.

You have also seen how the OSGi application feature pack supports a standardized evolution of Spring bean configuration and tightly integrates the DI bean container with the runtime to ensure the proper management of JPA and JTA contexts

Finally, you have seen how OSGi applications can be composed into larger SCA composites with services exposed remotely over a standard set of protocol bindings.

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 e-mail feedback:

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

This module is also available in PDF format at: [./wasosgijpafep_OSGi_overview.pdf](http://wasosgijpafep_OSGi_overview.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, Rational, 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.