Developing enterprise OSGi applications for WebSphere Application Server

An overview of modular applications

Skill Level: Intermediate

Dr. Ian Robinson (ian_robinson@uk.ibm.com)
Distinguished Engineer
IBM

Dr. Holly Cummins (cumminsh@uk.ibm.com)
Software Engineer
IBM

14 Jul 2010

Creating modular, extensible Web applications using standard Java™ EE deployment has its challenges, but can generally be accomplished with good design practices and discipline. Where it gets really hard, though, is when you want to separate out common modules to share between multiple enterprise applications, or use multiple versions of common libraries at the same time. OSGi is a Java modularity technology that has been used internally in IBM® WebSphere® Application Server and the Eclipse platform for many years, and was designed to enable the development and execution of dynamic, modular, extensible applications. The WebSphere Application Server V7 Feature Pack for OSGi Applications and JPA 2.0 enables modular enterprise applications to use OSGi directly to dramatically simplify their development, assembly, and deployment. The feature pack also provides an infrastructure in which modular design is no longer just a best practice but is the only practice.

Introduction

The vast majority of the cost associated with software development is not related to the initial design, development, and test of a new application -- although these can

be expensive -- but to the maintenance and evolution of the application thereafter.

Designing and building applications and suites of applications from coherent, versioned, reusable modules accessed only through well-defined interfaces reduces complexity and provides the greatest flexibility to maintain and evolve the software after its first release. A modular design, in which the modules offer well-defined services and interfaces, enables large-scale projects to be divided between teams who can focus on their own tasks without having to understand the details of the other teams' code beyond the agreed externals, reducing the breadth of complexity for each team and improving the time to delivery. In addition, the cost of application maintenance is reduced by minimizing the scope of the impact of any changes to the application -- assuming that scope can be determined. If a change to a module affects only its internals, with no impact to its external contract, then maintenance can be applied to a single isolated module and testing can be better targeted. This should all be self-evident, but we could benefit from some active help from our development tools and run time infrastructure to support such a modular design approach. For example, wouldn't it be nice to have an easier way to deploy and maintain common code that is used by multiple EARs?

First of all, what makes a good module?

- A module should provide a **coherent** logical function and should be sufficiently self-contained that it represents a practical "unit of reuse."
- A module should be **loosely-coupled** to other modules through well-defined external interfaces and dependencies.
- A module should **isolate** its internals from other modules so that changes to the internal behavior of a module have no impact on any other modules

Java classes are typically too fine-grained to form a logical unit of re-use between applications, their isolation focus being primarily limited to the encapsulation of instance data. Classes are usually packaged inside a JAR file that represents a coherent function. As the unit of deployment (and as the artifact that exists in the file system), the JAR is a very practical unit of reuse but it lacks some of the other characteristics of a good module.

Consider visibility: if you have a method in a class in package foo.bar that needs to be available to a class in package bar.foo, then you need to declare that method with a public access modifier. You have no way to indicate whether this method should or should not also be available to classes outside the JAR. You can apply conventions and best practices. You can organize your interfaces into external and internal packages using a package-naming convention. For example, WebSphere Application Server interfaces intended for use by applications are contained in sub-packages of com.ibm.websphere or com.ibm.wsspi, whereas internal interfaces not intended for application use are mostly in sub-packages of com.ibm.ws. If different teams working on different modules within a project are disciplined about

the definition and use of module externals, then these modules will remain coupled only through the defined externals. Of course, there's always that little bit of plus-plus function in your module that you couldn't live without, but isn't part of the module interface. So when another team finds out how much this will make their lives better too, what's the harm in letting them go ahead and use it? After all, you all work on the same overall project and the methods they'd use all have public Java accessibility, so it wouldn't need code changes to agree to let them use it -- if they bothered to ask. And now, without meaning to, you've broken the desired loose-coupling between bundles. The "scope of impact" of a change to your bundle's internals might now extend beyond your bundle, which makes it harder to figure out what might be affected and, therefore, what needs to be tested.

It is clear, then, that while the JAR is a very practical unit of reuse, it lacks the capacity to distinguish between externals and internals and has no means to isolate the latter. And it's actually worse than this: while its pretty easy for you to make this JAR (your unit of reuse) available in another environment, how do you figure out whether the new environment can satisfy all the dependencies it has? If you get this wrong, you have a potential time-bomb on your hands, meaning that it could be running happily for days and then need to load a class the new environment doesn't have and *boom* -- ClassNotFoundException. The problem here is that, in addition to providing no isolation, JARs also have no means to declare their own dependencies.

The JAR, then, is close to what you need for a good reusable module but it lacks some basic modularity characteristics. This is where OSGi bundles come in.

An OSGi bundle *is* a JAR but it has additional headers in the JAR manifest. In a plain JVM with no machinery to process this additional metadata, the bundle behaves like a normal JAR. In a JVM that includes an OSGi framework, the metadata is processed by the framework and additional modularity characteristics are applied. Listing 1 shows an example MANIFEST.MF.

Listing 1. OSGi headers in a bundle manifest

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: MyService bundle
Bundle-SymbolicName: com.sample.myservice
Bundle-Version: 1.0.0
Import-Package: com.something.i.need;version="[1.0,2.0)"
Export-Package: com.myservice.api;version=1.0.0
```

Some headers to note here are:

- **Export-Package** lists one or more packages, at specific versions, to

export from the bundle. Only exported packages are visible outside the bundle; any package not listed here is visible only inside the bundle.

- **Import-Package** lists one or more packages, at specific versions or ranges of versions, required by the bundle.

Without getting into the details of versioning (which will be discussed later) let's consider what these two headers are doing. Recall that earlier on we talked about the need to establish development best practices to define a module's external interfaces and to ensure that no internals were used by any client modules. With OSGi you now have a way for this best practice to be supported by enforcement in the runtime and to provide a module owner with a mechanism to properly encapsulate its internals. Furthermore, you have metadata in the bundle manifest that clearly identifies all the packages required by the bundle that need to be provided by other bundles. In a complex application consisting of many bundles, this gives you a much more deterministic means to identify the impact of a change to a bundle and to understand the likely impact on the rest of the system, reducing risk and cost in the software development lifecycle.

The processing of the metadata and the run time enforcement of the visibility rules are provided by the OSGi framework **resolver**. As each bundle is started, the resolver reconciles each of the bundle's imports against the exports declared by other bundles that have been installed into the framework and computes an individual classpath for each bundle. If a bundle has a dependency that cannot be resolved, the bundle will not start. The ticking time bomb mentioned earlier (for the case when a JAR is moved to a new environment which cannot satisfy all its dependencies) is dealt with by effectively removing the fuse. It's far easier to deal with an application that won't start than an application that works properly for some time and then fails unexpectedly with a ClassNotFoundException.

Nothing we've looked at so far is specific to an enterprise runtime -- so how does any of this relate to enterprise Java applications or enterprise applications servers?

Given an enterprise application server that is able to process and honor OSGi bundle metadata, then the first obvious benefit of OSGi is the proper modularization of complex enterprise applications consisting of large numbers of modules. But it goes much further than this and solves a number of additional problems common in enterprise environments. We touched briefly on one of these already: in many large-scale enterprise deployments where there are tens or hundreds of EARs deployed, it is usually the case that each EAR is self-contained to the extent that many common libraries used by the applications are packaged inside each EAR that needs them. You end up with an explosion of copies of these libraries on the file system or repository to which the EARs are deployed and in memory when the applications are started. While enterprise applications can often be deployed to various vendors' enterprise environments, with administrative dependencies configured for shared libraries installed independently from the EARs, these mechanisms vary from vendor to vendor and limit portability. Shared library

configuration is also typically dissociated from the deployment process itself, requiring separate, post-deployment administrative configuration, which increases the complexity of the end-to-end deployment process.

OSGi metadata and bundle repositories give you the opportunity to greatly simplify the deployment of suites of enterprise applications that share common libraries so that only application-specific modules need to be included in the application archive. The enterprise deployment process becomes much more powerful when it understands the OSGi metadata and can resolve bundle dependencies against the content of bundle repositories configured within the deployment environment. All common libraries can be managed in a centralized bundle repository which then becomes part of the enterprise (cell) configuration.

OSGi also gives you the opportunity to do a much better job of versioning in the enterprise environment. Today's enterprise applications often contain third-party frameworks or libraries with similar dependencies on common libraries. This can become a real headache if different frameworks require common libraries at different versions. These headaches can turn quite serious when you have everything working nicely until you need to update a vendor framework only to find you can't because it has a dependency on a later and incompatible version of a library already in use by another framework in the application.

OSGi versioning metadata and classloading eliminate this problem. In Listing 1, above, the Import-Package header indicates a dependency on the package com.something.i.need at a **version range** of "[1.0,2.0)". This means the dependency is satisfied by any version of the package in the range: 1.0 # version < 2.0. Therefore, version 1.0 or 1.5.0 or 1.9 would satisfy the dependency, but version 2.0 would not. OSGi's versioning mechanism enables package providers to indicate whether a new version of a package is backwardly compatible to the previous version, with a change in the major part of the version indicating an incompatible update. Package consumers can indicate the version, or range of versions, they are able to work with. (See this PDF whitepaper for more on OSGi versioning). Importantly, if two bundles within an application depend on different versions of the same package, then these dependencies can both be simultaneously satisfied because the OSGi resolver can calculate different classpaths for the two bundles.

The OSGi platform specifications, along with reference implementations and compliance tests, are produced by the OSGi Alliance and have been in common use for over ten years. In March 2010, the enterprise environment was embraced with the publication of the OSGi V4.2 Enterprise Specification. This defines the OSGi semantics of enterprise Java technologies such as transactions, persistence, and Web components. This important specification defines standard mechanisms to bring the Java EE and OSGi worlds together, including OSGi metadata for a bundle to declare that it is a Web bundle containing a web.xml file, a persistence bundle containing a persistence.xml file, or a Blueprint bundle containing a blueprint.xml file. Web and persistence bundles are just familiar Java EE modules with additional

OSGi manifest headers; Blueprint bundles are more like Spring modules, but with a standardized bean definition XML.

OSGi as a technology has been popular for many years in standalone applications and client-side application technologies, and is used internally within the implementation of many enterprise application servers, such as WebSphere Application Server. Direct leveraging of OSGi by the enterprise applications that run on these application server platforms has, until recently, been inhibited by the lack of OSGi standards for enterprise applications and by a lack of widely available dedicated tooling and enterprise runtime support. This has changed with the publication of the OSGi Enterprise Specification and the availability of the option to deploy applications as OSGi bundles in an increasing number of enterprise application servers.

The remainder of this article discusses the run time and tooling support for developing and deploying enterprise OSGi applications introduced in the WebSphere Application Server V7 Feature Pack for OSGi Applications and JPA 2.0.

The OSGi application feature pack

Support for OSGi applications in WebSphere Application Server is introduced in the WebSphere Application Server V7 Feature Pack for OSGi Applications and JPA 2.0. In common with other WebSphere Application Server feature packs, this is a freely available download that can be additively installed and uninstalled on top of an existing WebSphere Application Server V7.0.0.9 or later. The feature pack actually consists of two installable features: the **OSGi application feature** and the **JPA 2.0 feature**. These can be installed independently or together; 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 testing of Web applications. This article is focused only on the OSGi application feature.

The WebSphere Application Server OSGi application feature gives you the opportunity to develop and deploy enterprise applications in a modular fashion, introducing configurable OSGi bundle repositories into the WebSphere Application Server administrative process. This enables common bundles to be factored out of individual enterprise application archives and managed centrally in a bundle repository. Multiple versions of bundles can be managed in a bundle repository, and the appropriate version associated with individual enterprise applications can be specified in metadata for those applications.

Let’s look at what it means to be an OSGi application in WebSphere Application Server.

At the most basic level, an OSGi application can be the same collection of modules

deployed in a Java EE enterprise archive (EAR) but with additional OSGi metadata that enables the modules to be loaded as OSGi bundles. While there is no difference in the end result between running such an application as a standard Java EE application using Java EE classloaders or as an OSGi application using OSGi classloaders, there are a number of reasons you might choose to develop and deploy an OSGi application:

- Applications can be deployed in archives containing (if desired) just the application-specific content, along with metadata referencing any shared libraries (bundles). Application archives become smaller with only single copies of common libraries loaded into memory.
- Multiple versions of classes can be loaded simultaneously in the same application using standard OSGi mechanisms.
- Deployed applications can be administratively updated in a modular fashion, at the bundle-level.
- At development time, enterprise OSGi projects in IBM Rational® Application Developer enforce OSGi visibility rules so that projects can only access packages from other projects that explicitly declare them as part of the project externals, providing environmental support to development best practices.
- Applications can be designed for extensibility and dynamic update through the use of OSGi services.
- At run time, applications will only start successfully if all their dependencies can be resolved, reducing the occurrences of ClassNotFoundExceptions while an application is processing a workload.
- Applications can use their own versions of common utility classes distinct from the server runtime's own usage without needing to configure application Java EE classloader policies, such as PARENT_LAST mode.

The benefits of using OSGi become ever more apparent as your application grows in complexity, or the suite of deployed applications grows in size, increasing the challenge of managing updates to the applications' modules and the utility libraries they use.

Getting started with OSGi applications

There are a number of ways to get started with deploying your first OSGi application to WebSphere Application Server. Probably the most common way is to start with an existing Java EE Web application that you are already familiar with, or to start with one of the sample applications shipped as part of the OSGi application feature. In all cases, you need to package the OSGi application in a format that can be deployed.

OSGi applications are deployed to WebSphere Application Server through wsadmin or via the WebSphere Application Server administrative console just like any other application, but are packaged in a new type of archive called an **enterprise bundle archive** (EBA). This is similar to an EAR but its modules are deployed as bundles to the desired target servers. An EBA represents a single isolated OSGi application consisting of one of more application modules and is the unit of deployment for an enterprise OSGi application. Similar to an EAR file, an EBA can contain all the constituent modules or bundles that make up the application but can, instead, 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.MF file which 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, as well as the deployable content of the application.

Listing 2 shows a complete application manifest for a simple OSGi application. The Application-Content header describes the primary bundles that make up the application and which will be deployed when the application is deployed. The reason you need a "list of contents" for the application being deployed is because not all of this content needs to be included with the application manifest inside the EBA; the bundles certainly can be packaged inside the archive but some or all of them can equally be provisioned by the WebSphere Application Server deployment process from a configured bundle repository. If, for example, the com.example.common.audit bundle provides common audit services for all the applications managed by an IT organization, that bundle should be installed into a common bundle repository rather than deployed as part of each application EBA.

Listing 2. Simple APPLICATION.MF for an OSGi application

```
Application-Name: MyApp
Application-SymbolicName: com.example.app
Application-Version: 1.0
Application-Content: com.example.app.web;version=1.0.0,
com.example.app.persistence;version=1.0.0,
com.example.common.audit; version=1.0.0
```

With the OSGi application feature, WebSphere Application Server provides a built-in OSGi bundle repository whose contents can be managed through WebSphere Application Server administration, as shown in Figure 1. WebSphere Application Server also provides the option to use external OSGi bundle repositories that are accessed by a configured repository URL.

Figure 1. Internal OSGi bundle repository in admin console

During the deployment of an OSGi application, WebSphere Application Server administration calculates all package and OSGi service level dependencies for the application to ensure that the application can be fully provisioned from a combination of the EBA and the configured bundle repositories. The application manifest file itself can be authored by a developer or generated by tools, such as Rational Application Developer. It is important to understand that the application manifest needs to list, in the Application-Content, only the primary application bundles, and it need not enumerate all the package-providing and service-providing bundles that these depend upon. The WebSphere Application Server deployment process resolves all the primary bundles' dependencies to calculate a transitively-closed list of application content, and prevents application deployment if the resolution process detects missing dependencies. It is also important to understand that configuration is by-exception, so an application manifest is not required at all if the Application-Content is all contained within the EBA. Each module that forms a part of the Application-Content, along with all its calculated dependencies, is deployed as an OSGi bundle. If a Web module with no OSGi metadata is included in the Application-Content, then the WebSphere Application Server deployment process will automatically convert this to a **Web application bundle** (a WAB module) during deployment. The configuration-by-exception and automated detection and conversion of WAR files provides the quickest way to get started with OSGi applications -- you can take a Java EE EAR containing only Web modules and deploy as an OSGi application simply by renaming the archive's .ear file extension to .eba.

OSGi services and the Blueprint component model

Another feature OSGi brings is a standard extensibility model, through OSGi services. You can design your application to take advantage of OSGi's rich and

dynamic service-based model to consume services from the OSGi service registry. OSGi services then become extension points that you can design into your application, extending it in the future via service provider implementations introduced through separate bundles with no change to the original application code. OSGi defines Java APIs for registering and discovering OSGi services, but a declarative approach to OSGi services is much simpler. This is where the enterprise OSGi V4.2 Blueprint container comes in: the Blueprint container manages the lifecycle and dependency injection of POJO bean components and is configured through an application-level XML bean definition file, which is a standards-based evolution of the Spring bean definition XML. By standardizing the bean definition XML schema and bean lifecycle management semantics in the OSGi Alliance, it has become possible to re-factor the dependency injection container out of the application (where the Spring framework libraries are typically packaged) and into the middleware. The enterprise OSGi v4.2 Blueprint container implementation from the Apache Aries project is integrated and extended as part of the WebSphere Application Server runtime when the OSGi application feature is installed.

Blueprint provides a fine-grained bean assembly model for OSGi applications, as well as a simple declarative means to publish a service provided by a POJO bean component. For example, the bean definition snippet shown in Listing 3 defines a bloggingServiceComponent bean, implemented by the BloggingServiceImpl class, for which a BloggingService service is registered in the OSGi service registry.

Listing 3. Blueprint bean definition

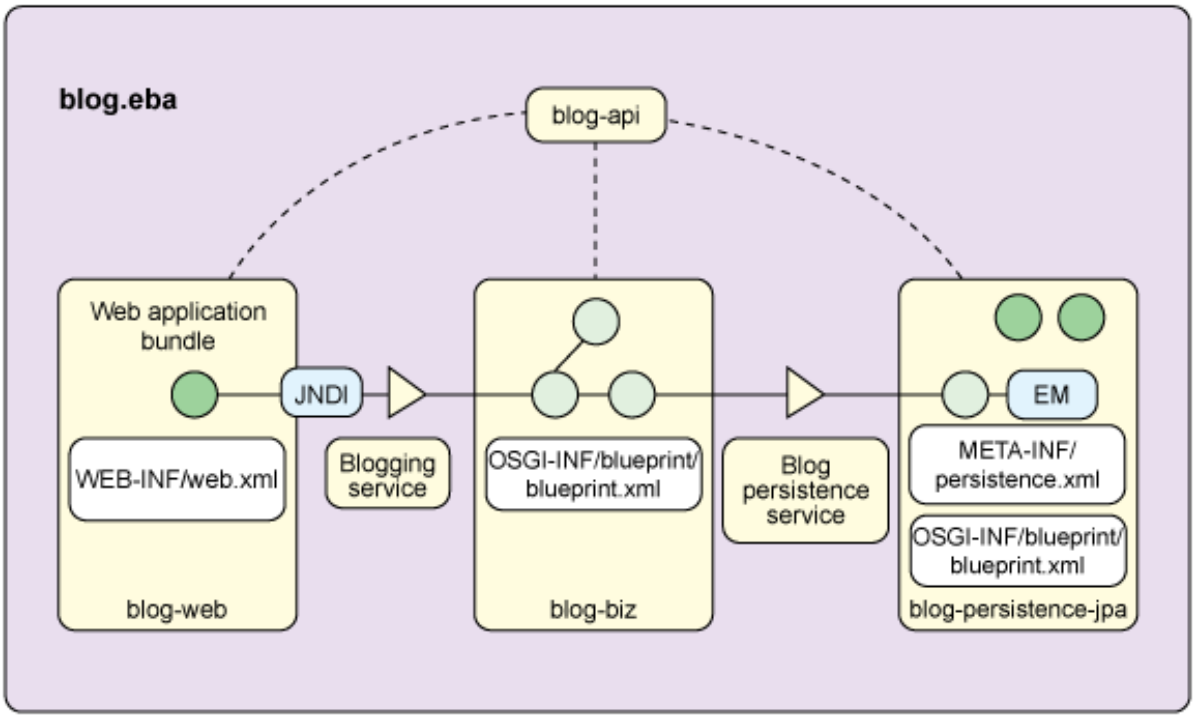
```
<blueprint>
  <bean id="bloggingServiceComponent"
       class="com.ibm.ws.eba.example.blog.BloggingServiceImpl">
<property name="blogEntryManager" ref="blogEntryManager"/>
<property name="blogAuthorManager" ref="blogAuthorManager"/>
<property name="blogCommentManager" ref="blogCommentManager"/>
</bean>
<service ref="bloggingServiceComponent"
interface="com.ibm.ws.eba.example.blog.api.BloggingService"/>
  ...
</blueprint>
```

In this example, three properties (which might be references to OSGi services or beans defined elsewhere in the bean definition blueprint.xml) are injected by the container into the bloggingServiceComponent bean when it is instantiated. Blueprint configured beans give OSGi applications a convenient way to encapsulate their business logic into POJO components that have their dependencies and configuration injected into them by the Blueprint container. Since the POJO bean 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.

A sample OSGi application

Let's look at one of the samples shipped as part of the OSGi application feature to illustrate how such applications are typically developed and deployed. The sample is a simple blog publishing application illustrating the combined use of Web, Blueprint, and persistence technologies within an OSGi application. (If you would like to explore the code in more depth, the source code for the blog application is shipped with the OSGi Application feature pack.)

Figure 2. Blog sample application



The blog sample demonstrates a typical architecture for an enterprise application. It consists of four loosely-coupled bundles: a Web layer, a business logic layer, a persistence layer, and an API bundle.

First of all, notice how the APIs are pulled into their own bundle. This is an OSGi best practice that keeps couplings loose. If necessary, an implementation can easily be swapped out for a different one at deploy time or even at run time.

The coupling between the bundles makes use of a fundamental OSGi construct, services (represented as triangles in Figure 2), which maintain the desirable loose-coupling between bundles and enable bundle implementations to be more easily replaced with minimal impact on the rest of the application. As described in OSGi services and the Blueprint component mModel, OSGi applications do not need to interact directly with the OSGi service registry, but can do so instead through declarative Blueprint configuration of simple POJO beans. Both the blog-biz and blog-persistence bundles use Blueprint-configured beans to encapsulate business logic and have their dependencies and configuration injected into them by the Blueprint container that manages their lifecycle. The Blueprint container wires the

beans together within the blog-biz bundle and also wires components in the blog-biz bundle to the Blog persistence service provided by the blog-persistence bundle.

The application's front end is a Web module using familiar Java EE servlet components. To illustrate how simple it is to combine Java EE programming styles with an OSGi service-based style, the sample blog-web bundle follows a pure Java EE programming model and uses JNDI to access the Blueprint-published OSGi service. The enterprise OSGi specification defines a standard mechanism for JNDI clients to obtain references to OSGi services, providing a natural bridge between two programming styles.

The blog-persistence-jpa bundle uses JPA as the persistence framework through which blog authors and entries are persisted to and retrieved from a database. It leverages the Blueprint container's ability to manage both persistence contexts and global transactions to ensure the business logic remains as simple as possible to develop and unit test.

Finally, the blog JARs are packaged together in an EBA and deployed to WebSphere Application Server.

Let’s look at each element of this application in more detail.

API bundle

Perhaps not surprisingly, the API bundle is the simplest component. It is a simple OSGi bundle that does not make use of any enterprise features. As mentioned above, it is the OSGi metadata in the JAR manifest that gives a JAR its "bundle characteristics." In the case of the blog API, the bundle declares that it exports the com.ibm.ws.eba.example.blog.api, com.ibm.ws.eba.example.blog.comment.persistence.api, and com.ibm.ws.eba.example.blog.persistence.api packages, at version 1.0.0.

Listing 4. Blog sample API bundle manifest

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: blog
Bundle-SymbolicName: com.ibm.ws.eba.example.blog.api
Bundle-Version: 1.0.0
Export-Package:
com.ibm.ws.eba.example.blog.api;version=1.0.0,
com.ibm.ws.eba.example.blog.comment.persistence.api;version=1.0.0,
com.ibm.ws.eba.example.blog.persistence.api;version=1.0.0
```

Web application bundle

In many respects, the Web bundle resembles a standard Java EE Web archive; it declares its servlet mappings in a web.xml file, and packages its code in WEB-INF/classes. However, there are some differences. As you would expect for an OSGi bundle, the manifest declares some imports -- in this case, the blog API exported by the API bundle. The Web-ContextPath is declared in the manifest, rather than in the EAR's application.xml file. The practical reason for this is that an OSGi application is not an EAR and so does not have an application.xml file. However, there is a more fundamental motivation. Having all the configuration information for the Web module within the Web bundle itself enables the Web archive to be much more modular and self-contained, in keeping with the spirit of OSGi.

Listing 5. Blog sample web bundle manifest

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-ClassPath: WEB-INF/classes
Bundle-Name: blog.web
Bundle-SymbolicName: com.ibm.ws.eba.example.blog.web
Web-ContextPath: blog
Bundle-Version: 1.0.0
Import-Package:
com.ibm.ws.eba.example.blog.api;version="[1.0.0,1.1.0)",
com.ibm.json.java;version="[1.0.0,2.0.0)"
```

The OSGi framework in WebSphere Application Server recognises the archive as a Web bundle and hands over to a Web container to manage the lifecycle of the servlets. Integration between the Java EE Web container and the OSGi container is achieved through federated JNDI lookup, a feature of the enterprise OSGi specification. All OSGi services are automatically registered in JNDI and can be accessed in a manner familiar to Java EE components. For example, the blog servlet accesses an implementation of the BloggingService, as shown in Listing 6.

Listing 6. Accessing OSGi services from Web components

```
InitialContext ic = new InitialContext();
return (BloggingService) ic.lookup("osgi:service/"
+ BloggingService.class.getName());
```

It could equally use an @Resource annotation to inject the reference as an

alternative to using the JNDI Context API.

Blueprint bundle

The business logic is implemented as a collection of POJOs with a declarative blueprint.xml configuration file to associate bean definitions and references with OSGi services. The Blueprint container takes care of the interactions with the OSGi service registry to manage the interactions with and the lifecycle of the services. The blueprint.xml snippet shown above in Listing 3 is from the blog sample business bundle and illustrates a typical pattern to declare a bean, inject its dependencies (which may be references to services or other beans), and publish the bean as a service.

Persistence bundle

The persistence bundle makes use of another standard feature of enterprise OSGi, JPA integration, to take advantage of managed persistence. The JPA persistence unit is configured as usual through a persistence.xml file. The bundle manifest of a persistence bundle requires a Meta-Persistence header to indicate that it is a persistence bundle, as shown in Listing 7.

Listing 7. Blog sample persistence manifest

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: blog.persistence
Bundle-SymbolicName: com.ibm.ws.eba.example.blog.persistence
Bundle-Version: 1.0.0
Meta-Persistence:
Import-Package:
com.ibm.ws.eba.example.blog.persistence.api;version="[1.0.0,1.1.0)",
javax.persistence;version=1.0.0
```

The persistence bundle actually gets a better deal than just managed JPA -- it gets *container* managed JPA since it is a Blueprint bundle (as well as a persistence bundle); the Blueprint container fully manages the JPA PersistenceContext and injects it into the managed bean (BlogPersistenceServiceImpl) annotated for the blogExample persistence unit, as shown in Listing 8.

Listing 8. Annotation-based injection of JPA EntityManager

```
private EntityManager em;
@PersistenceContext(unitName = "blogExample")
```

```
public void setEntityManager(EntityManager e) {

em = e;

}
```

As an alternative to annotating the bean Java code, the container-managed PersistenceContext could instead have been declared using the <jpa:context> element in the Blueprint bean definition, as shown in Listing 9.

Listing 9. Blueprint container-managed Transaction and Persistence configuration

```
<bean id="persistenceManager"
        class="com.ibm.ws.eba.example.blog.persistence.BlogPersistenceServiceImpl">
<tx:transaction method="*" value="Required"/>
<jpa:context property="em" unitname="blogExample"/>
</bean>
```

Listing 9 also illustrates Blueprint container-managed transactions. In this example, all the methods of the persistenceManager bean are run under a global transaction established (or joined) by the Blueprint container. The set of container-managed transaction values supported by the WebSphere Blueprint container is the same as that supported for the EJB container. (See Transactions and OSGi Applications for more details.)

Application assembly

The last piece of the sample blog application is the EBA. As described earlier, the EBA is the deployable unit, containing an application manifest which describes the application content. For the blog sample, the application manifest looks like Listing 10.

Listing 10. Blog Sample APPLICATION MANIFEST

```
Application-Name: 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, 2.0.0)",

com.ibm.ws.eba.example.blog.web;version=1.0.0,

com.ibm.ws.eba.example.blog;version=1.0.0

Use-Bundle: com.ibm.json.java;version=1.0.0
```

The basic format for this application-level metadata is described above. There are

two things to note in the blog sample manifest:

- The API, blog, and Web bundles deployed for the application must be at version 1.0.0 or later, but the persistence bundle must be a version in the range 1.0.0 up to (but not including) 2.0.0. What this means is that, while all the bundles can be at version 1.0.0 when the application is initially deployed, the application has been assembled to accommodate future updates of the bundles. If later, after the application has been deployed, a 1.1.0 version of the persistence bundle becomes available then the WebSphere Application Server administrator can (through wsadmin or the admin console) update that bundle within the application. The blog sample provides an additional version 1.1.0 of the blog persistence bundle to demonstrate this administrative update capability.
- The Use-Bundle header lists the com.ibm.json.java bundle separately from the Application-Content. This indicates that this bundle is shareable with other applications. All the bundles listed in the Application-Content will run in an OSGi framework instance that isolates these bundles from other OSGi applications in the same application server. In this way, one OSGi application cannot have unintended side-effects on another OSGi application just because it is deployed to the same target server. Because one of the goals of OSGi application support is to simplify module-sharing between applications in a fashion that is integrated with the deployment process, the Use-Bundle header provides a mechanism to identify modules which should be shared. As well as explicit Use-Bundle content, any bundles whose packages or services are implicitly resolved when the application is deployed are also considered to be providing shared content. When the application is started on the target server(s), any bundles that are identified as being shared are loaded into a server-wide parent OSGi framework, the contents of which are visible to each of the isolated application frameworks.

The blog sample is packaged to illustrate the use of the WebSphere Application Server bundle repository: one of the required bundles, the com.ibm.json.java bundle, is provided separately from the blog.eba archive. This shared bundle must be installed into the WebSphere Application Server bundle repository before the blog.eba is deployed. The deployment process calculates the package and service dependencies of all application bundles against the contents of the deployed EBA archive, the configured bundle repositories, and WebSphere Application Server provided API/SPI packages (such as Java EE packages and com.ibm.websphere packages). If all the dependencies are resolved, the application is successfully deployed to the configured target server(s). In the WebSphere Application Server admin console, installed OSGi applications appear in the same Business-level applications (BLA) collection view as Java EE and SCA applications. (See administrative tasks for deploying OSGi applications for details.)

SCA composition

The OSGi application feature can be used in conjunction with WebSphere Application Server's Service Component Architecture (SCA) support when the WebSphere Application Server V7 Feature Pack for SCA v1.0.1.5 or later is installed. SCA provides an assembly model for composing potentially heterogeneous components into coarse-grained composites that define external services and references for which a variety of different bindings can be configured. SCA and OSGi share some common concepts around the notions of assembling components into a coherent module with explicitly declared externals. It is quite natural to use these technologies together, although they do address different and unique aspects of service assembly.

On its own, an OSGi application can be assembled from a collection of OSGi bundles, each of which performs a coherent function within the application, like the Web, business, and persistence bundles of the blog sample application:

- **Within a bundle**, fine grained components can be implemented as POJO beans and wired together using a Blueprint bean definition; the internals of one bundle are not visible to other bundles.
- **Within an OSGi application**, bundles are wired together through OSGi services or external package dependencies; the internals of one OSGi application are not visible to other applications.

The above are two different levels of modularity. SCA provides the next:

- **Within an SCA composite**, SCA components are wired together through SCA services; the internals of one SCA composite are not visible to other composites.

An OSGi application can be assembled as an SCA component within a coarse-grained SCA composite and can selectively expose the OSGi services it implements, using SCA to provide remote bindings for these services. An SCA component implementation provided by an OSGi application has an SCA component type of *implementation.osgiapp*. It might remotely expose any of its Blueprint-configured OSGi services as SCA services and configure remote bindings for these services.

For example, suppose you wanted to compose the blog sample OSGi application as an SCA component, along with some other components with a different implementation type (for example, an EJB component) into an SCA composite, and then expose the bloggingServiceComponent defined in the snippet shown in Listing 3 as an external service with a default SCA binding. The first thing you would need to do is update the Blueprint service definition from which this snippet is taken to

indicate that the service is remoteable. This requires a standard OSGi property, service.exported.interfaces, to be added to the service definition (Listing 11).

Listing 11. Declare that a service is remotely available

```
<blueprint>
  <bean id="bloggingServiceComponent"
       class="com.ibm.ws.eba.example.blog.BloggingServiceImpl">
    ...
</bean>
<service ref="bloggingServiceComponent"
interface="com.ibm.ws.eba.example.blog.api.BloggingService">
<service-properties>
<entry key="service.exported.interfaces" value="*"/>
</service-properties>
</service>
  ...
</blueprint>
```

The application manifest shown in Listing 10 needs to indicate that this service should be visible outside the OSGi application. Do this by adding the Application-ExportService header to the application manifest (Listing 12)

Listing 12. Export the service from the OSGi application

```
Application-Name: Blog

Application-SymbolicName: com.ibm.ws.eba.example.blog.app

Application-Version: 1.0

...

Application-ExportService: com.ibm.ws.eba.example.blog.api.BloggingService
```

Now you can configure an SCA component whose implementation is an OSGi application and which provides an SCA service (Listing 13)

Listing 13. Configure an SCA component

```
<composite name="SocialMediaComposite">
<component name="BlogComponent">
<scafp:implementation.osgiapp
applicationSymbolicName="com.ibm.ws.eba.example.blog.app"
applicationVersion="1.0.0"/>
<service name="bloggingServiceComponent">
<binding.sca>
</service>
</component>
</composite>
```

In this example, a default SCA binding is shown but other bindings (such as binding.ws for Web services or binding.jms for JMS) could be specified, depending on how the service needs to be exposed.

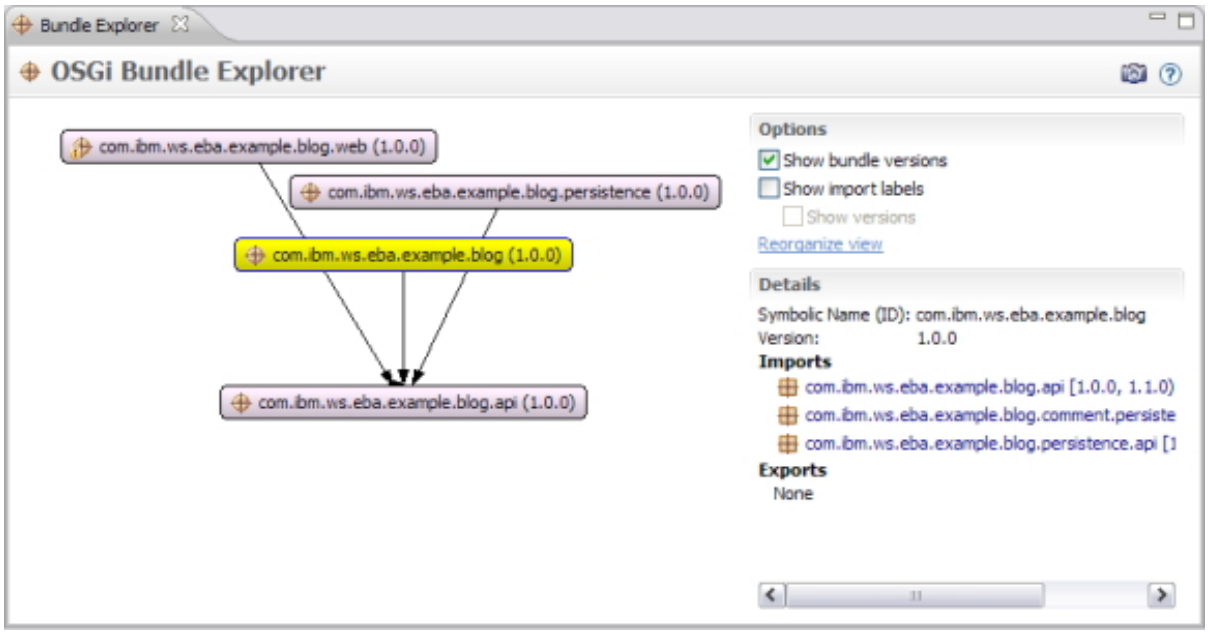
Similar to how it exports remote services, implementation.osgiapp SCA components

can import remote service references. A broader discussion of the scenarios and patterns for using OSGi and SCA together are beyond the scope of this article. (See documentation for assembling and deploying OSGi components in SCA composites for more.)

Development tools

Most of the development activities, and hence development tools, used to build enterprise OSGi applications are common with Java EE tools, but there are some new considerations. Primarily, these are around the compile-time classpaths, the authoring of the OSGi bundle and application manifests and, optionally, Blueprint bean definition files. Rational Application Developer adds support for developing OSGi application tools with the introduction of new projects types for OSGi bundle projects and OSGi application projects, with automated generation of manifests and forms-based editors to modify them. OSGi modularity semantics are honored in the project build paths so that only the packages explicitly imported and exported in a project's bundle manifest are shared between projects. Rational Application Developer's facet-based configuration enables OSGi projects to be configured as OSGi Web projects or OSGi JPA projects, and integrates tools for authoring web.xml, persistence.xml, and blueprint.xml. Rational Application Developer’s new Bundle Explorer can be used to visualize the relationships between the bundles in an OSGi application, as illustrated in Figure 3. OSGi application projects can be imported from or exported to .eba archives, and can be run from the Rational Application Developer workspace, either on the WebSphere Application Sever V7 optionally installed with Rational Application Developer or on a remote WebSphere Application Server V7 environment that includes the OSGi application feature.

Figure 3. Rational Application Developer Bundle Explorer



Beyond the integrated Rational Application Developer tooling, there are a number of open source tools to help generate OSGi bundle manifests. In addition, the EBA Maven Plugin developed by the Apache Aries community can generate an OSGi application manifest from a Maven pom configuration as part of a build.

Operational details

So far we've talked about developing, assembling, and deploying OSGi applications. This section takes a brief look at the result of a deployment and some of the administrative actions that can subsequently be taken.

A useful utility provided with the WebSphere Application Server OSGi application feature pack is the osgiApplicationConsole script installed in the install_dir/feature_packs/aries/bin/ directory. This is a wsadmin client application that provides a remote OSGi console for the specified application server. The command-line parameters are shown in Listing 14.

Listing 14. Using the OSGi application console utility

```
-h The host name of the target machine.
-o The port number of the SOAP port of the target server
-u The user ID, if the wsadmin connection is secured.
-p The password, if the wsadmin connection is secured.
example:
install_dir/feature_packs/aries/bin/osgiApplicationConsole -h server1.acme.com -o 8880
```

At the command prompt, you can type `help()` for a list of interactive commands. If you run this command after deploying and starting the blog sample application and then type `list()` at the command prompt, you'll see entries for two OSGi frameworks on the target server. One is for the OSGi framework into which the blog sample application is installed and one is the server-wide shared framework. To find out details of bundles, services, and packages for each framework, you need first to connect to the desired framework: from the command prompt, type: `connect(1)` to connect to the application framework. To list all the bundles installed into this framework, and see their bundle states, type `ss()`. You should see something like Listing 15.

Listing 15. Interactive OSGi application console

```
wsadmin>ss()
ID State Bundle
0 ACTIVE org.eclipse.osgi_3.5.2.R35x_v20100126
1 ACTIVE com.ibm.ws.eba.example.blog.app_1.0.0
2 ACTIVE com.ibm.ws.eba.example.blog.persistence_1.0.0
3 ACTIVE com.ibm.ws.eba.example.blog.web_1.0.0
4 ACTIVE com.ibm.ws.eba.example.blog.api_1.0.0
5 ACTIVE com.ibm.ws.eba.example.blog_1.0.0
```

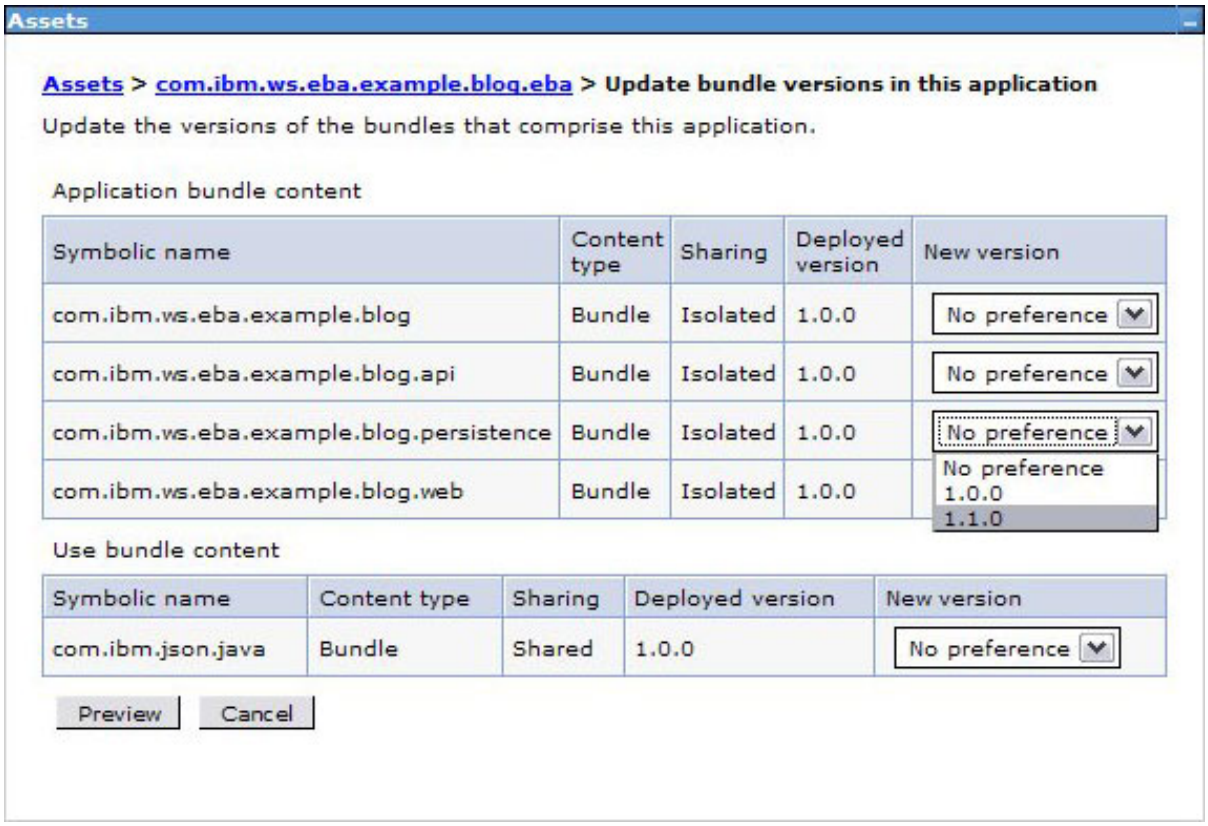
The WebSphere Application Server Information Center documents the interactive commands and filters you can use to get more information about running applications. Something this console helps you visualize very easily is the manner in which isolation is maintained for discrete OSGi applications so that the internal bundles and packages of one application are not visible to other applications. It is not necessary to deploy different applications on different servers just to isolate one application from another. Bundles that need to be shared between applications (for example, the Use-Bundle content described earlier (link to Application Assembly) are installed into the shared framework.

The last thing we'll look at is administrative update of a bundle in a deployed application.

After an application has been installed, you can navigate to the admin console's Assets view to see the installed versions of each bundle. If later versions of one or more of the bundles are available, in any of the configured bundle repositories, and those versions lie within the version ranges defined for the bundles in the OSGi application manifest, then you have the opportunity to administratively update some or all of those bundles to the desired version. By way of illustration, the blog sample provides an additional 1.1.0 version of its persistence bundle. If, after deploying the blog sample, you then add the 1.1.0 version of blog persistence to the WebSphere Application Server internal bundle repository, you can navigate via the Application

Assets view in the panel shown in Figure 4.

Figure 4. Administrative update of bundles



You can now choose the 1.1.0 version of the persistence bundle and select the **Preview** button to re-resolve the application and check for any inconsistencies. If the preview produces no errors, you can then select the **Commit** button to update the application and save the configuration.

Conclusion

The benefits of a modular approach to application design include reduced complexity, reduced time to delivery, and increased serviceability. While the benefits are well-understood and best practices are often put in place to encourage modular design, Java EE infrastructure on its own has limited ability to enforce or encourage modular design.

Enterprise OSGi combines the modularity principles and infrastructure of OSGi with a familiar Java EE programing model for enterprise applications and the server environments in which they run.

The OSGi application feature of the WebSphere Application Server V7 Feature Pack for OSGi Applications and JPA 2.0 provides comprehensive run time and integrated

administrative support for deploying OSGi applications to WebSphere Application Server. Application integrity is maintained by isolating applications from one another while enabling the sharing of specific bundles to be directed by application assembly metadata. The application deployment process is augmented to enable application content to be provisioned from a combination of application-specific archives and shared OSGi bundle repositories, reducing application archive size as well as disk and memory footprint. Spring-like declarative assembly and dependency injection, with its benefit of simplified unit test outside the application server, is provided through the Blueprint container, governed by OSGi standards and integrated into the server runtime. Assembly of OSGi applications into heterogeneous composites and remote binding to OSGi services is provided through a new SCA component implementation type for OSGi applications.

Tooling for developers of OSGi applications is provided in Rational Application Developer, including generators and editors for OSGi metadata, enforcement of OSGi modularity constraints in the development environment, import/export of enterprise bundle archives, and workspace-integrated capabilities to run and debug OSGi applications on a server.

Try it out!

Resources

Learn

- WebSphere Application Server Information Center including documentation for the featurepack
- B est practices for developing and working with OSGi applications
- Apache Aries project
- Eclipse Equinox project
- Administrative tasks for deploying OSGi applications
- assembling and deploying OSGi components in SCA composites
- IBM developerWorks WebSphere

Get products and technologies

- WebSphere Application Server V7 Feature Pack for OSGi Applications and Java Persistence API 2.0
- Rational Application Developer

Discuss

- OSGi Application feature discussion forum

About the authors

Dr. Ian Robinson
Dr. Ian Robinson is an IBM Distinguished Engineer and senior architect for the WebSphere Application Server, responsible for the strategy and development of OSGi technologies in WebSphere and the transaction processing capabilities of the WebSphere platform.

Dr. Holly Cummins
Dr. Holly Cummins is an IBM software engineer. She is a developer for the WebSphere feature packs and is also committer on the Apache Aries project. She has been with IBM for nine years and holds a DPhil in quantum computation and an MSc in software engineering.