Best practices for developing and working with OSGi applications

Graham Charters
Jeremy Hughes
Tim Mitchell
Alasdair Nottingham
Mark Nuttall
Zoe Slattery
Tim Ward

July 14, 2010

For over a decade, OSGi technology has solved application development modularity challenges around complexity, extensibility, and maintenance. With the introduction of the IBM® WebSphere® Application Server Feature Pack for OSGi Applications and JPA 2.0, enterprise Java™ applications consisting of OSGi bundles can now be developed and deployed to WebSphere Application Server V7. This article describes best practices for developing well constructed OSGi applications to help you benefit most effectively from this new capability.

Introduction

OSGi modularity provides standard mechanisms to address common challenges with Java applications. In 2007, the OSGi Alliance Enterprise Expert Group (EEG) was formed to bring OSGi infrastructure benefits to enterprise application developers in the form of an enterprise Java programming model. The support for OSGi applications works with the enterprise-level qualities of service of IBM WebSphere Application Server to give the most complete and robust enterprise server for modular Web applications. You can use the WebSphere Application Server Feature Pack for OSGi Applications and JPA 2.0 to deploy and manage Web applications as a set of versioned OSGi bundles. You can also configure one or more bundle repositories, as part of the provisioning infrastructure, to host common bundles that multiple applications use, and to simplify the deployment of applications that use those common bundles. The WebSphere Application Server V7 Feature Pack for SCA V1.0.1.5 update adds support for composing OSGi applications with heterogeneous assets in support of the concepts of service-oriented architecture (SOA). (See Related topics)

With any new technology there are some dos and don’ts, also known as best practices for architects, developers, and deployers. OSGi technology has been available for over a decade and in that time many best practices have emerged. This article describes the principal best practices
relevant to writing OSGi applications for the OSGi Applications feature of WebSphere Application Server and the integration with Service Component Architecture (SCA). Some of these items are general OSGi best practices, others are specific to the support provided in WebSphere Application Server; for clarity, those that fall in the latter group are indicated as such.

**Best practices**

The best practices described here are:

1. **Use Blueprint**
2. **Use Blueprint to enable service-based provisioning**
3. **Use Blueprint to enable SCA integration**
4. **Version that which is versionable**
5. **Separate API from implementation**
6. **Share services not implementations**
7. **Good bundles are like good classes: loosely coupled and highly cohesive**
8. **Avoid split packages and Require-Bundle**
9. **List contained content in the Application-Content header**
10. **Make WABs not WARs**
11. **Only Use-Bundle when you must**
12. **Use persistence bundles to share your persistence units**
13. **Make best use of the component models on offer**
14. **Let the container do the heavy lifting**

Details for each of these practices are provided in the sections that follow.

**1. Use Blueprint**

Using Blueprint is a good practice in general. Some of the benefits that Blueprint provides are support for a simple POJO development and testing model, simple component assembly, and the fact that it is based on open standards. Using Blueprint in WebSphere Application Server OSGi applications is additionally recommended because of enhanced support for container integration, Service Component Architecture (SCA) integration, and service-based provisioning.

**Here's why**

Blueprint is a simple component assembly model based on the Spring Framework. It was standardized by the OSGi Alliance in the Enterprise OSGi R4 V4.2 specification with contributions from SpringSource. As a standardized form of Spring, it supports the same dependency injection patterns, enabling simple Java components to be developed that do not use framework APIs and are therefore easy to unit test.

The article [Developing enterprise OSGi applications for WebSphere Application Server](#) outlines some of the reasons why WebSphere Application Server supports the Blueprint component model. These reasons are motivated by client requirements, and much of the OSGi application support in WebSphere Application Server relies on the use of Blueprint to enable certain capabilities, such
as service-based provisioning and SCA integration. Although using Blueprint is a good idea in general, there are even more reasons to do so when developing WebSphere Application Server OSGi applications.

2. Use Blueprint to enable service-based provisioning

(Specific to WebSphere Application Server)

*When developing and using OSGi services with Blueprint, the Blueprint definitions will be used to ensure service dependencies are satisfied when provisioning the application into WebSphere Application Server.*

*Here's why*

An OSGi bundle manifest describes the packages that a bundle imports from another bundle and the packages that it exports for use by other bundles. Therefore, given a particular bundle, it is possible to determine a set of bundles required in order to satisfy all package imports, including transitive dependencies. A best practice is to separate interface and implementation into different bundles (see *Separate API from implementation*). Another best practice is to use OSGi services for implementation dependencies (see *Share services not implementations*). A consequence of these two best practices is that provisioning package dependencies only satisfies interface requirements, but does not provide implementation bundles. The WebSphere Application Server OSGi applications feature solves this by using Blueprint definitions to determine the services that a bundle provides and requires. It then uses this information during application deployment to provision implementation bundles from the application archive (.eba) or internal bundle repository (a bundle repository configured and managed through WebSphere Application Server administration).

*Example*

**Figure 1. Service based provisioning example**

Figure 1 shows an API bundle, an implementation bundle providing a service, and a client bundle that uses the service from the implementation bundle. Both the client and implementation bundles have a package dependency on the API bundle that contains the Java interface for the service. The client bundle is part of an OSGi application (listed in its content), and the API and implementation bundles are shared and are to be provisioned from the internal bundle repository. The API bundle gets provisioned into WebSphere Application Server as a result of the package dependency, and as long as both the client and implementation bundles are implemented using
Blueprint, the implementation bundle will also be provisioned. If one or other of these two bundles does not use Blueprint, then the implementation will not be provisioned.

For example, if the implementation bundle uses Blueprint but the client does not, then the application will be provisioned but without the implementation bundle. This is because the provisioner is not aware of the missing service dependency from the client bundle. If the client bundle uses Blueprint but the implementation bundle does not, then the application will fail to deploy because the provisioner will not be able to satisfy the client bundle's service dependency.

3. Use Blueprint to enable SCA integration
(Specific to WebSphere Application Server)

When developing OSGi applications that are to be integrated with other technologies, it is a best practice (actually, essential) to use Blueprint to define the OSGi application's service implementations.

Here's why

In WebSphere Application Server, SCA is used to assemble OSGi applications with other application types (for example, Java EE). It is also used to expose OSGi application services via various transports and protocols (such as JMS, Web services, Atom) and to enable service dependencies to call out via those transports and protocols. To achieve this, SCA requires a description of the services provided by the OSGi application and the services required by the application. SCA re-uses the information described in the Blueprint XML to derive the information required by SCA. It is therefore necessary to use Blueprint when exposing OSGi applications' services to SCA. If an OSGi application contains services that are not defined using Blueprint, then a Blueprint facade must be created that describes these services to Blueprint and then forwards the requests to the non-Blueprint service implementations.

Figure 2. Blueprint support for SCA integration

Figure 2 shows an OSGi application with a single bundle that provides a service to be called from outside the application by SCA. This call can be from another SCA component (perhaps
another OSGi application or a Java EE application), or from a specified binding, such as a Web service or JMS invocation. The same bundle also requires a service that is to be provided outside the application by SCA. Again, this might be to call another SCA component, or to make an invocation over a specified binding. In the first diagram in Figure 2, Bundle A is implemented using Blueprint. SCA uses the Blueprint service definition to understand how to identify and invoke the target OSGi service. SCA does not use the Blueprint service reference definitions because it has sufficient information in the application manifest (the artifact used to define an OSGi application) to determine what services are valid targets. In the second diagram, the service and references are implemented in Bundle B using some other component model, such as declarative services (DS). SCA does not understand DS and therefore this does not work. In the third diagram, a Blueprint facade bundle, Bundle C, is used to describe the service to SCA and then forward the invocation to the DS service implementation in Bundle B.

4. Version that which is versionable

Provide semantic versions for bundles and packages to enable loose coupling between API clients and providers.

Here’s why

In OSGi, packages and bundles can be versioned. If no version is specified it defaults to zero. Packages and bundles should be semantically versioned so clients can protect themselves against API changes that might break them.

Semantic versioning defines a versioning scheme that enables the version to communicate compatibility information to clients. Semantic versioning uses a major.minor.micro numbering scheme. Major, minor, and micro are natural numbers (0, 1, 2, and so on).

- A change in the major version indicates a binary incompatible API change for clients of the API; that is, a client compiled against version 2 of an API would need to be recompiled to work against version 3. An example of a binary incompatible change is removing an interface or method.
- A change in the minor version indicates that the API has been enhanced, but existing clients do not need to be recompiled. An example of this kind of change is the addition of an interface, or of adding a method.
- A change to the micro version indicates a change has been made that does not change the API. An example of this is a bug fix removing a NullPointerException.

APIs versioned in this way enable a client to limit the version of an API to those it can work with. When importing a package or requiring a bundle, the client can express the versions it can work with. For example (1,2) permits the client to work with any version 1 package or bundle, but not with a version 2 package or bundle. This only works because the client knows that future changes indicate a binary compatible change using the major version.

If semantic versioning is used for bundles and packages, then it is possible to have multiple versions of an API in the runtime at once and have bundles use both at the same time.
So far, this best practice has described compatibility in terms of an API client. When following the **Separate API from implementation** best practice, the implementation of an API will be in a separate bundle from the API and it also gains advantages from semantic versioning. For an implementor, a change in either the major or minor version indicates a binary incompatible change, for example, they may be required to implement a new interface or method. (For additional advantages and considerations when using semantic versioning, see [OSGi Alliance Semantic Versioning Technical Whitepaper](#).)

**Examples**

Figure 3 shows two providers of an API where the API is the same version. In OSGi these two providers are considered equivalent. Both the client (to the right) and an implementor (to the left) express a dependency that will match either API provider.

**Figure 3. A client and an implementation can use either of two equivalently versioned packages**

Figure 4 shows two API providers at different versions. It shows how a change in the minor version affects clients and implementers differently. In this case, you have version 1.0 and version 1.1 of the API being provided. Client A can use either API provider; Client B, making use of new API methods, can only use the higher version 1.1 API. Implementation A can only use the API provided by the bundle API A, and Implementation B can only use the API provided by the bundle API B.

It is important to know that the service registry will limit the services visible to the requestor to those that implement the version of the API to which the bundle has been bound.

**Figure 4. How a client and implementation are affected differently by a minor API version change**

Figure 5 shows an API change that is incompatible from both the client's and the implementor's perspective. In this case, Implementation A and Client A are bound to API A. Implementation B and Client B are bound to API B.
Figure 5. How clients and implementations are similarly affected by a major API version change

5. Separate API from implementation

When building OSGi bundles, a best practice is to put any API classes into a separate bundle from the implementation classes.

Here’s why

Separating out the API classes from the implementation classes gives much greater flexibility. Having a separate API bundle enables a client to use any implementation provider; additionally, it enables more than one provider to be used simultaneously. Without this separation, the client bundle would need to be restarted in order to wire to a new provider bundle.

This best practice can also reduce the package dependencies of the API bundle, therefore reducing the possibility of cyclic dependencies between implementations of different APIs. This can easily occur when the two API implementations make use of each others API internally.

Once the API classes have been put into a separate bundle, it is important to ensure that no implementation packages are exported, and implementations of the API are published as OSGi services using the public API.

Examples

The scenario shown in Figure 6 shows a client bundle that imports both API and implementation classes from a provider bundle that has not separated these packages. The provider bundle also exports both the implementation and API packages. This is a bad practice because there is no separation and also because the implementation classes are exported and not exposed as services.
Figure 6. Badly designed provider bundle where the API and implementation classes are in the same bundle

The next scenario in Figure 7 shows a client bundle importing API packages from one bundle and implementation packages from another. Although the API and implementation packages have been separated, the provider still exports the implementation packages, rather than exposing the classes as services.

Figure 7. Badly designed provider bundles where the API and implementation classes have been separated

The final scenario in Figure 8 shows the best practice. The provider bundle has separated both the implementation and API packages. Additionally, the implementation classes are exposed as OSGi services.

Figure 8. Well designed provider bundles where the API and implementation classes have been separated
6. Share services not implementations

When developing OSGi bundles, a best practice is to use the OSGi service registry to achieve the factory pattern and to construct instances of classes exported by other bundles. Along with the Separate API from implementation best practice, this achieves a loose coupling of client, API, and API implementation.

Here's why

The practice of separating API and implementation abstracts multiple implementations from the clients that consume them. It makes it possible for one vendor’s implementation of an API to be replaced by another’s without needing to change the clients that use the API.

To be effective, the separation of API and implementation requires a mechanism for the client to obtain an instance of an implementation without knowledge of which implementation it is going to use. In non-OSGi systems, this is usually done via a static factory method on a class in the API. The static factory method chooses which implementation to instantiate, out of the implementations on the classpath. All of the implementations are visible because of the flat class space. The algorithm used to determine which implementation should be instantiated is specific to the API.

This mechanism will not work in OSGi because the bundle containing the API classes should not be able to see any of the bundles which contain implementations of the API.

OSGi provides a much neater solution in the form of the OSGi service registry. The bundle containing an implementation of the API either:

- Registers an instance of the API interface in the OSGi service registry. This has the effect of there being just one instance of the implementation for all client bundles to use.
- Registers an implementation of the OSGi ServiceFactory interface in the OSGi service registry. The ServiceFactory interface is used by OSGi when a service instance is requested. In this case, a ServiceFactory can choose the instance to return to each requesting client bundle. For example, it could return the same instance to all requesting client bundles, or it could return a different instance to each requesting client bundle, or anything in between.

The client is completely separated from the implementation, as in the static factory method pattern. In addition:

- The API is completely separated from all implementations, as the OSGi service layer handles the instance creation in collaboration with the implementation(s).
- The algorithm used to determine which implementation to return is consistent across all APIs rather than being specific to a single API, as is the case with the static factory method pattern used in non-OSGi systems.

The OSGi service registry is a dynamic mechanism for sharing services where services may come and go at run time. Blueprint helps API clients cope with this dynamism by injecting a required service into a bean in the client bundle when it becomes available. Should the service become unavailable, Blueprint can inject an alternative implementation, if there is one registered.
7. Good bundles are like good classes: loosely coupled and highly cohesive

When authoring bundles, limit their function to a specific task. Keep the number of tasks a bundle performs low, and try to have as few dependencies as possible.

Here's why

When authoring an OSGi application, it is very tempting to add more function to an existing bundle rather than to author a new bundle. This, however, quickly leads to large bundles with very low cohesion and usually causes bundles to import and export large numbers of packages. It is also a poor practice to use the Require-Bundle header to hide a large number of package imports, as this simply adds tight coupling between the bundles. As the dependency graph of a bundle grows, the number of bundles that must be downloaded and installed increases, often exponentially. This is sometimes called the "hairball effect." Poorly purposed bundles can require tens or even hundreds of megabytes to be downloaded and installed within your application just to access one or two simple functions, whereas a well-focussed bundle will have very few dependencies, a small dependency tree, and will be more readily reusable in other applications. It should be noted that this best practice mirrors the best practice for object oriented class design, and for the same reasons. An OSGi bundle should be well encapsulated, highly cohesive and loosely coupled -- just like a well written class -- to prevent it becoming tied to specific versions of other bundles or packages and to increase the opportunities for reuse.

Example

An example of a poorly purposed bundle would be a logging implementation that enables users to log to the file system, to a database via JPA, or to a messaging queue (Figure 9). In order to use the file system logger an application would need to load the messaging API, the JPA API, and so on. Thus the application would contain implementation classes for the loggers it isn't using. A much better solution would be to separate the logging interfaces and each of the implementations from one another (Figure 10). By doing this an application can choose to include the one logging bundle it uses, rather than having to include the extra implementation classes, and any future implementations, that are present in the logging bundle.

Figure 9. Poorly purposed system where a single bundle provides multiple implementations of the same API
Figure 10. A well purposed system where each implementation of an API is provided by a separate bundle

8. Avoid split packages and Require-Bundle

When developing OSGi bundles a best practice is to keep all of the classes from any one package in a single bundle. Splitting a package across multiple bundles leads to the use of Require-Bundle, compromising the extent to which systems using the bundles can be extended and maintained.

Here's why

Modular systems consist of loosely coupled, highly cohesive modules. Each module performs a coherent logical function and the boundaries between them are well defined. This design goal produces systems that are easier to maintain and extend. An OSGi bundle is a level of modularity above a package.

OSGi provides loose coupling of modules, which are bundles in OSGi terminology, by having each bundle declare its dependencies in terms of the Java packages it requires. These are satisfied at run time by packages that bundles provide. A bundle does not directly depend on a bundle that provides a dependency; the decision as to which bundle provides the dependency is left to the OSGi runtime and is based on which bundles are deployed to the framework. The effect is that a bundle providing a particular package can be substituted with a different bundle providing the same package, without having to change the bundles that depend on that package.

A split package occurs when a package is exported by two bundles at the same version and the set of classes provided by each bundle differs. Classes might be duplicated or, more typically, part of the package is in one bundle and the remainder of the package is in another.
Figure 11. Consumers of split packages need to use the Require-Bundle header

Figure 12. A complete package exported from a single bundle maintains high bundle cohesion

At run time, OSGi will satisfy a bundle's dependency specified by an Import-Package header with an export from another bundle. Even if there is more than one bundle that exports the package, only one of them is used to satisfy the dependency. The bundles are “wired” together. Classes required that only exist in the other bundle that exports the same package (the one that the importing bundle wasn't wired to) will not be visible to the importing bundle's classloader. If the importing bundle tries to use one of those classes at run time, a ClassNotFoundException will be thrown. Since the import/export package mechanism in OSGi only provides a way of wiring one bundle to another on a per package basis, another mechanism is needed to wire one bundle to multiple others to work around this split package situation. This can be achieved by using the Require-Bundle header and specifying the bundle symbolic names of the bundles exporting the required package.

However, this tightens the coupling between the bundles:

- The client now depends on the two (or more) bundles providing the package.
- All the packages exported by the providing bundles are now visible to the client, and as the client bundle is developed over time it might start to depend on these other packages.
- The bundle providing the package can no longer be replaced by a different one, with a different symbolic name, without making changes to all the client bundles that require it.

Splitting packages across multiple bundles and thereby forcing the use of the Require-Bundle header makes the application more tightly coupled and therefore more expensive to maintain and extend. If you have a package that makes sense to split between bundles, then you do not have high cohesion between those two parts and you should use two different packages.

9. List contained content in the Application-Content header

(Specific to WebSphere Application Server)
Bundles contained within an Enterprise Bundle Archive (EBA) file should be listed in the Application-Content header of that EBA's META-INF/APPLICATION.MF. Dependency bundles are best stored in a bundle repository, from which they are available to all provisioned applications.

**Here's why**

Bundles contained within an EBA can affect how that application is provisioned, but are invisible to other applications so far as their provisioning goes. If an EBA contains one or more dependency bundles, not listed in Application-Content, then these might end up being loaded into one or more server's shared bundle spaces as a result of the provisioning process. This gives rise to potential problems, because other applications could end up being wired to these dependency bundles even though they could not be provisioned against them. This can lead to unexpected changes in runtime behaviour which can be very difficult to detect.

Beyond the fact that contained content should be listed in the Application-Content header, you should also be aware that the ability to contain content at all is provided mainly as a convenience to you during development -- its use in production environments is not recommended. The bundles comprising the application content can be contained in the EBA, or a bundle repository. Production environments are expected to encourage -- and often insist -- on applications obtaining all their bundles from a bundle repository. This enables governance to be applied to the production bundle repositories as the sole sources of application code.

An APPLICATION.MF will be automatically generated for an EBA provided without one. The Application-Content will be interpreted as every bundle contained in the EBA's root directory. Each bundle's version range will be locked down to its exact version. In this case, none of the bundles can be shared with other applications, or upgraded after deployment. Developers without access to the IBM Rational® Application Developer integrated development environment, which provides more direct integration, might find it convenient to package applications with no APPLICATION.MF and to import a new application asset containing the updated content, rather than to install each new bundle into the internal bundle repository. An APPLICATION.MF should, of course, be created before the end of the development cycle. Contained content will generally become less common as you move from development through test to production environments. In a test environment it might be convenient to generate an EBA containing all its dependencies so that it can be installed regardless of the state of the target server's bundle repositories. This is unwise in production environments for the reasons discussed above.

Finally, note that WARs that are not WABs can only be contained; they are not bundles, and cannot be stored in a bundle repository. See the next best practice for more on this subject.

**10. Make WABs not WARs**

*When developing enterprise OSGi applications, create a Web application bundle (WAB) and do not rely on the WAR to WAB conversion.*

**Here's why**
The OSGi applications feature of WebSphere Application Server will automatically convert a WAR file into an OSGi bundle, known as a Web application bundle. This feature is very useful when getting started with OSGi, but the automatic conversion prevents certain features from being used. As a result, during development, WAR files should be converted to a bundle for deployment into a production system.

When using OSGi bundles, an application can indicate a range of bundle versions that can be supported. Having done this, individual bundles can be updated without redeploying the whole application. This capability depends on modules having an known identity, which is not the case with the auto conversion, where the identity is generated at deploy time.

When a WAR is converted to a WAB, the package imports that get generated are not versioned. As a result, the package could pick up binary incompatible versions of the required package. For example, if versions 2 and 3 of a package are available, it is not possible to prevent the WAR using version 3. This will result in run time errors if the WAR requires version 2 of the package and is incompatible with version 3.

### 11. Only Use-Bundle when you must

*(Specific to WebSphere Application Server)*

*When creating a WebSphere Application Server OSGi application, a best practice is to specify the Use-Bundle application manifest header only in a specific subset of bundle sharing scenarios.*

**Here's why**

The WebSphere Application Server Use-Bundle application manifest header provides a way to direct the application deployment process to prefer one bundle over any others that export the same package.

At run time, OSGi applications are isolated from each other, but their dependencies are shared. The bundles specified in the Application-Content header of an OSGi application run in their own isolated OSGi framework. The dependencies run in the server’s shared bundle space.

The dependencies themselves are determined during OSGi application deployment. During deployment, the packages imported by the application bundles are matched with packages exported by bundles either in the application or in the configured bundle repositories. If a bundle in a bundle repository is needed then it is provisioned to the server. Due to package version constraints, it is possible for two applications to be deployed and configured to run with different bundles providing the same package in the shared bundle space. This only becomes a problem if the two applications need to interact with each other using instances of classes from the package in question. If they do, then the Use-Bundle application manifest header should be used to ensure the two applications are wired to the same bundle for that package.

It isn't necessary to specify Use-Bundle to achieve the sharing of dependencies between applications. It is only necessary to specify Use-Bundle when two applications need to share the same version of a class to ensure they are wired to the same providing bundle.
Example

For example, an application containing Bundle X is deployed which imports a package org.hal.a from version 1.0 up to but not including version 2.0 (Figure 13). In accordance with OSGi semantic versioning, this means X can be wired to any bundle that exports org.hal.a with no breaking API changes, from the perspective of a consumer using the API. At this time, only one bundle, called API.A, exports org.hal.a and it does so at version 1.0. So, Bundle X is wired to the API.A. Bundle X therefore has visibility to implementations of org.hal.a version 1.0. Implementation.A is one such implementation.

**Figure 13. OSGi wires Bundle X to API.A to provide org.hal.a at a version acceptable to Bundle X**

Then, an application containing Bundle Y is deployed (Figure 14). Bundle Y imports the org.hal.a package from version 1.5 up to but not including version 2.0. API.B is also deployed, which exports package org.hal.a at version 1.5. Bundle Y cannot be wired to API.A because it imports a newer version of org.hal.a; it *can* be wired to API.B. The Implementation.B bundle provides an implementation of org.hal.a version 1.5 and imports org.hal.a from version 1.5 up but not including version 1.6. The [OSGi Alliance Semantic Versioning technical whitepaper](https://osgi.org/library/semver-whitepaper.pdf) describes the reasons behind this in more detail.

**Figure 14. OSGi wires Bundle Y to API.B to provide org.hal.a at a version acceptable to Bundle Y**

Then a third application is deployed, containing Bundle Z. Bundle Z imports the org.hal.a package with the same version range as Bundle X: from version 1.0 up to but not including 2.0 (Figure 15). The author of Bundle Z wants it to consume Implementation.A, but because API.B exports org.hal.a at version 1.5 (which is within Bundle Z’s imported range) Bundle Z can be wired to API.B and subsequently can only see Implementation.B.
Best practices for developing and working with OSGi applications

12. Use persistence bundles to share your persistence units

When using JPA in an application, package the bundle as an OSGi persistence bundle. If your clients are not to use the EntityManagerFactory directly, expose a data access service in the OSGi service registry and do not export the entity classes from the persistence bundle.

Here’s why

The WebSphere Application Server OSGi application feature has support for OSGi persistence bundles that conform to the OSGi JPA Service specification. By packaging JPA managed classes and persistence.xml in an OSGi persistence bundle, it is possible to create a reusable module with easily shareable JPA resources that can be used in the WebSphere Application Server OSGi runtime, and with unmanaged JPA Service implementations. Persistence bundles can also be
shared by a large number of OSGi based persistence clients, and by not including persistence client code are both highly cohesive and readily re-usable. Remember to specify the Meta-Persistence bundle manifest header, and that the persistence descriptor can be given any path within the bundle.

If a persistence unit is packaged inside a legacy WAR, it should be broken out as a separate OSGi bundle. Persistence units in WARs are managed by the Java EE container and are not shared within the OSGi framework. As such, they cannot be used by other OSGi bundles in the OSGi application. If removing the persistence unit from the WAR is not possible, remember that it cannot be accessed by the Blueprint container or via the OSGi service registry.

Another good reason for separating persistence bundles from other application code is explained by an earlier best practice. If several applications need to access data within a database, then it makes sense for them to reuse the same entity mappings. This will help to prevent data corruption within the database, as both applications share the same data mappings, and only one bundle needs to be updated for any future schema changes. If the persistence unit is packaged inside a larger bundle then the entities can no longer be easily shared between multiple applications, requiring duplication of code and maintenance.

13. Make best use of the component models on offer

(Specific to WebSphere Application Server)

Between them, OSGi, the WebSphere Application Server OSGi Applications feature, and SCA offer a range of increasingly coarse-grained component models:

- Use Blueprint to define fine-grained components within OSGi bundles.
- Define OSGi applications as collections of bundles providing a specific application function.
- Use SCA to expose services from one OSGi application to another, and to provide services to and from regular Java EE applications.

Here's why

Use Blueprint to manage your fine-grained components and interactions with other OSGi Bundles. Keep your bundles cohesive (typically small) and your dependencies loosely coupled. The principles of high cohesion and loose coupling continue to apply as bundles are grouped into enterprise bundle applications.

An OSGi application can be thought of in much the same way as a traditional Java EE enterprise application. In assembling an application, keep in mind that WebSphere Application Server deploys the bundles listed in Application-Content into their own isolated framework. All applications' dependency bundles are deployed into a single shared bundle space. Thus, the factoring of common code libraries and services into shared dependency bundles is encouraged. This saves memory, increases reuse, and helps make a system easier to govern. The bundles listed in a given OSGi application's Application-Content should comprise only the persistence, business, and presentation logic unique to that application.
There are two ways of driving work into an OSGi application. The first is via HTTP into a WAB. The second is to invoke a service listed in the Application-ExportService header; to do this requires the use of SCA.

SCA is the most coarse-grained of the available component models. It provides a distributed programming model and enforces pass-by-value semantics. Use SCA to expose services from one OSGi application to another, and to provide services to and from regular Java EE applications. Rational Application Developer 8 provides tools support for building and integrating OSGi applications as SCA components.

14. Let the container do the heavy lifting
(Specific to WebSphere Application Server)

Use container services to build applications and supply enterprise qualities of service rather than writing code to manage services.

Here's why

Application programmers often wish to solve complex problems in a robust, reliable manner and rely upon enterprise services, such as transactions and managed resources, to provide them with the qualities of service that they need. There is, however, often some reluctance to give complete control of these services to the container, as there will undoubtedly be a loss in flexibility associated with the loss of control. As a result, many applications choose to manage their own resources and lifecycles.

Inversion of control, and particularly dependency injection, are incredibly powerful tools for simplifying application development, but at their heart they rely upon the fact that the container can be responsible for lifecycle and resource management. Applying declarative transactions to all methods in a Blueprint bean requires one line of XML, whereas it would take dozens of lines of code in each method to accomplish the same result within the application. Similarly for resource management, using a Blueprint resource reference enables direct injection of a resource into an application, and also enables it to be configured with security credentials so that they do not have to be supplied in the application. If a developer chooses to locate his own resources, then there is no opportunity for the container to authenticate the application, thus the credentials are stored inside the application, and the application must be changed if the credentials expire.

Example

A bundle offers a JPA-based data access service, but does not use any container services. It must manually manage its transactions, EntityManager lifecycle, and service registration. This takes approximately 100 lines of code, and requires multiple try/finally blocks for clearing up resources. Clients must also use the OSGi API to look up the service, which requires more lifecycle management and try/finally blocks. Also, as Blueprint was not used to register and consume the service, the application must handle error cases where the service is not available. There is also no automatic service-based provisioning.
By using declarative transactions, Blueprint, and managed JPA the application reduces in size by several hundred lines of complex lifecycle and error management code. This reduces the number of potential defects in the application. The application, and all future applications that make use of the data access service, can also take advantage of service-based dependency provisioning.

**Conclusion**

This article described some best practices in constructing OSGi bundles and WebSphere Application Server OSGi applications to achieve high cohesion within bundles and loose coupling between bundles. By employing these best practices, you will be able to create more maintainable and extendable applications that use OSGi technology running in WebSphere Application Server V7 with the Feature Pack for OSGi applications and JPA 2.0.
Related topics

- WebSphere Application Server Information Center including documentation for the Feature Pack for OSGi Applications and JPA 2.0
- OSGi Alliance Semantic Versioning Technical Whitepaper (PDF)
- IBM WebSphere Application Server V7 for Developers
- IBM WebSphere Application Server V7 Feature Pack for OSGi Applications and Java Persistence API 2.0
- IBM Rational Development Tools for OSGi Applications
- IBM Rational Application Developer Beta for OSGi-based application tools

© Copyright IBM Corporation 2010
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)