## 1 – OSGi headers

We have so far touched on OSGi headers, the special entries found in Java Archive Manifest files that make a JAR into a bundle. There exists a large collection of these headers, and different organizations add additional ones to assist in specific application domains. As a quick reference guide, we have prepared a table of OSGi headers and their purpose.

| Header | Purpose |
| --- | --- |
| Bundle-ActivationPolicy | This tells runtime if the bundle should be loaded lazily, meaning start is not called until the first access of the class occurs. |
| Bundle-Activator | It specifies the class implementing the org.osgi.framework.BundleActivator interface. |
| Bundle-Category | A comma-separated list of category names. |
| Bundle-Classpath | This is a manual control of where to load classes from within the bundle. The default is "." or the root of the bundle; however, one may specify embedded jars into the classpath. |
| Bundle-ContactAddress | This specifies where to find more information on the bundle. Typically a website URL, organization, or project maintainer. |
| Bundle-Copyright | Indicates copyright holder of the bundle. |
| Bundle-Description | A brief text description of the bundle's purpose. |
| Bundle-DocURL | URL to find more information in a bundle. |
| Bundle-Icon | A list of icon URLs that can be used to represent the bundle. Icon files may be internal to the bundle or an absolute web address. No specific format is required. |
| Bundle-License | This describes which license(s) the bundle is available under. |
| Bundle-Localization | This references a set of property files, which may be used to localize the bundle. It is typically used for language support, currency, and units of measurement. |
| Bundle-ManifestVersion | This is an optional header, and defaults to Version 1. As of OSGi R4, the value 2 was introduced. Most bundles have Version 2 specified. |
| Bundle-Name | Text identifier for the bundle. |
| Bundle-NativeCode | This is used to provide information about native libraries that should be loaded by the bundle. |
| Bundle-RequiredExecutionEnvironment | This is a list of execution environments (JVM versions) that must be present for the bundle to be installed. |
| Bundle-SymbolicName | Along with Bundle-Version, these headers are used to uniquely identify a bundle at runtime. This header is mandatory. |
| Bundle-UpdateLocation | This indicates where the bundle should look for updates to the OSGi runtime bundle.update() call. |
| Bundle-Vendor | This is the vendor of the bundle. |
| Bundle-Version | This is the version of the bundle in the Major.Minor.Micro.Qualifier format. The first three fields are numeric, while the qualifier can be any text value. |
| DynamicImport-Package | Wires to packages that may not be known in advance. Using this feature is expensive as the framework must search all exported packages instead of using a calculation. |
| Embed-Dependency | This acts as a macro to include resources and updates the Bundle-Classpath header to pick up these resources at runtime. |
| Export-Package | This makes a comma-separated list of packages available to other bundles. |
| Export-Service | Deprecated – formerly used to make static exports, services are now consumed dynamically. |
| Fragment-Host | This identifies the parent bundle this fragment should attach to. |
| Import-Package | This bundle's package level dependencies. At runtime, the OSGi framework will be the bundle to any compatible bundle that provides the required package. |
| Import-Service | Deprecated – formerly used to make static imports, services are now consumed dynamically. |
| Include-Resource | BND tool header – copies resource into JAR. |
| Private-Package | BND tool header – specifies packages that are not exported. |
| Provide-Capability | This is part of the generic requirements/capabilities model for bundle wiring. Export package statements are translated into capabilities. |
| Require-Bundle | This declares a dependency with a bundle-symbolic name instead of a package name. Import-Package is the preferred mechanism. |
| Require-Capability | Bundle requires capabilities provided by another bundle. The imported package and required bundle are translated into requirements, but they can be anything we want as a requirement. |

While the preceding table is helpful in quickly understanding some of the most commonly seen headers in the core specification, out of which we have listed a few custom headers, several more exist for various tools such as BND, Eclipse, and Spring. We feel that we should look a little closer at Bundle-SymbolicName, Bundle-Version,Import-Package, and Export-Package.

### Tip

**Apache Karaf command**

To view headers of installed bundles quickly, issue the osgi:headers (bundle:headers on Karaf 3.x) command as follows:

karaf@root> osgi:headers BundleID

### Bundle-SymbolicName

The SymbolicName of a bundle is the only mandatory header that an OSGi bundle must contain. This header supports a directive to indicate if the bundle should be treated as a singleton, or that only one bundle of this name should exist in the framework:

Bundle-SymbolicName: com.packt.osgi.starter;singleton:=true

### Tip

The singleton directive lets the OSGi environment know that there should only be one bundle in the system with this name at the same time.

### Bundle-Version

Along with Bundle-SymbolicName, this attribute uniquely identifies a bundle in the framework. Additional attention needs to be paid to OSGi versioning as the OSGi environment will pay strict attention to version requirements as defined by the bundle's import and export packages.

The generally adhered practice for versioning is as follows:

Major.Minor.Micro.Qualifier

#### Major

A major number indicates incompatibilities in bundle use between versions, that is, incompatible changes in APIs.

#### Minor

A minor number indicates that this is a backward compatible build, and users should be safe updating to a higher version (of course you should always verify through testing). However, if the version applies to an API package, implementers of the package will not work. For instance, adding a method to an API is a minor version change.

#### Micro

A micro version update indicates that an internal change has occurred, but it does not alter its APIs. Normally, this is used when bug fixes are made to the bundle.

#### Qualifier

A qualifier is used to indicate small internal changes, or to communicate to the users' bundle status, that is, a milestone build.

Bundle-Version: 1.0.0.milestone1

Bundle-Version: 1.0.0.SNAPSHOT

### Import-Package

A bundle declares its dependencies through this header. The semantics surrounding the version attribute merits additional attention; the use of (' and ') denotes exclusive values, and [' and '] denotes inclusive values. These braces are used to denote version ranges. There are five cases we must examine:

#### Inclusive Minimum, Exclusive Maximum:

The version is denoted with "[minimum, maximum)", which translates to "wire to package with at least minimum version, up to but not including the maximum version". This version range is commonly seen when your wiring bundles to us up to the next major version of a library.

Import-Package: com.packt.some.package;version="[1.2.3, 2.0.0)"

#### Inclusive Minimum, Inclusive Maximum:

The version is denoted with "[minimum, maximum]", which translates to "wire to package with at least minimum version, up to and including the maximum version".

Typically, these ranges are used when bundle compatibility is known for a very specific set of releases.

Import-Package: com.packt.some.package;version="[1.2.3, 1.2.99]"

#### Exclusive Minimum, Exclusive Maximum:

The version is denoted with "(minimum, maximum)", which translates to "wire to package higher than the minimum version, up to but not including the maximum version."

Import-Package: com.packt.some.package;version="(1.2.3, 2.0.0)"

Note that the range [1.2.4, 2.0.0) may be more practical to use.

#### Exclusive Minimum, Inclusive Maximum:

The version is denoted with "(minimum, maximum]", which translates to "wire to package higher than the minimum version, up to and including the maximum version".

Import-Package: com.packt.some.package;version="(1.2.3, 2.0.0]"

#### Inclusive Minimum:

The version is denoted with "minimum", which translates to "wire to package with at least minimum version". This particular notation can be a source of confusion as the framework will wire the highest version available of the package requested. It's a good practice to always state a version range to avoid issues from future packages not being compatible with your bundle.

Import-Package: com.packt.some.package;version="1.2.3"

### Tip

**Apache Karaf command**

To view which bundles are providing packages to your bundle imports, issue the dev:show-tree (bundle:tree-show on Karaf 3.x) command, as follows:

karaf@root> dev:show-tree BundleID

### Export-Package

This header is used to tell the framework if any packages are being provided by this bundle to the environment. Package names should be fully qualified and include a version attribute. When the version attribute is not provided, the version is defaulted to 0.0.0.

Export-Package: com.packt.util, com.packt.osgi;version="1.0"

Now that we have a deeper understanding of bundles, let's look closer at their life cycle.

## 2 – OSGi life cycle

We have described OSGi applications as living entities; by this we mean that these applications appear to evolve as the life cycles of their constituent bundles are lived. The life cycle layer facilitates this functionality.

OSGi bundles are dynamically installed, resolved, started, updated, stopped, and uninstalled. The framework enforces the transitions between states, one cannot directly install a bundle and jump to an Active state without first passing through the resolved and starting states. The transitions between each state are illustrated in the following figure:

### Installed

Bundles came into existence in an OSGi framework in the installed state. A bundle in this state cannot be immediately started, as the preceding diagram depicts that there is no direct transition from the installed state to the starting state. An installed bundle is also not active. There are three possible transitions: the bundle may become resolved, uninstalled, or refreshed.

### Tip

**Apache Karaf command**

To install a bundle in Karaf, issue the osgi:install (bundle:install on Karaf 3.x) command, as follows:

karaf@root> osgi:install URLs

Having a bundle installed to the OSGi framework does not mean it is ready to be used; next we must resolve its dependencies.

### Resolved

Entering the resolved state requires the framework to ensure that all the dependencies of a bundle have been met. Upon having its dependencies ensured, the bundle is now a candidate to be transitioned to the starting state. A resolved bundle may be refreshed, transitioning the bundle back to the installed state. A resolved bundle may also be transitioned to the uninstalled state. A resolved bundle is not active; however, it is ready to be activated.

### Tip

**Apache Karaf command**

To resolve an installed bundle in Karaf, issue the osgi:resolve (bundle:resolve on Karaf 3.x) command, as follows:

karaf@root> osgi:resolve BundleID

### Starting

A resolved bundle may be started. The starting state is transitory; the framework is initializing the resolved bundle into a running active state. In fact, the transition from the starting to active state is implicit.

### Tip

**Apache Karaf command**

To start a resolved bundle in Karaf, issue the osgi:start (bundle:start on Karaf 3.x) command, as follows:

karaf@root> osgi:start BundleID

### Active

The bundle is fully resolved, providing and consuming services in the OSGi environment. To perform any more transitions on an active bundle, it must first be stopped.

### Updating

Bundle updates occur when the framework is instructed to re-evaluate a bundle's dependencies; this action is synonymous with refreshing a bundle. When this action occurs, all of the wiring to and from the bundle is broken, so care must be taken before refreshing to avoid starting a bundle storm (one bundle refreshing causes a domino effect of other bundles refreshing).

### Tip

**Apache Karaf command**

To update a bundle in Karaf, issue the osgi:update (bundle:update on Karaf 3.x) command, as follows:

karaf@root> osgi:update BundleID [location]

The location option allows you to update the bundle via its predefined updated location or to specify a new location to find bundle updates.

### Stopping

Stopping a bundle transitions it from the active to the resolved state. The bundle can be restarted while it remains in the resolved state.

### Tip

**Apache Karaf command**

To stop an active bundle in Karaf, issue the osgi:stop (bundle:stop on Karaf 3.x) command, as follows:

karaf@root> osgi:stop BundleID

### Uninstalled

Uninstalling a bundle transitions an installed or resolved bundle out of the OSGi environment; however, the bundle is not removed from the environment! Why is this? While the bundle is no longer available for use, references to the bundle may still exist and used for introspection.

To help leverage these states in your bundles, the OSGi specification provides a hook into your bundle state via the Activator interface.

### Tip

**Apache Karaf command**

To uninstall a bundle in Karaf, issue the osgi:uninstall (bundle:uninstall on Karaf 3.x) command, as follows:

karaf@root> osgi:uninstall BundleID

### BundleActivator

A bundle may optionally declare an Activator class implementing the org.osgi.framework.BundleActivator interface. This class must be referenced in the bundle manifest file via the BundleActivator header. Implementing the activator allows the bundle developer to specify actions to be performed upon starting or stopping a bundle. Generally, such operations include gaining access to or freeing resources, and registering and unregistering services.

The entry in manifest.mf will appear as follows:

Bundle-Activator: com.packt.osgi.starter.sample.Activator

When building with maven-bundle-plugin, the following configuration instruction is added:

<Bundle-Activator>

com.packt.osgi.starter.sample.Activator

</Bundle-Activator>

The process can be seen in the following screenshot:

## 3 – OSGi core services

In this section we'll review the core OSGi services. The published specification for Rev 4.2 of the OSGi Framework is 332 pages long. As such, we've highly condensed the material in this section. To read the entire specification please visit <http://www.osgi.org/Download/Release4V42>.

| OSGi Core Service | Purpose |
| --- | --- |
| Conditional Permission Admin | The OSGi framework specification contains both the Permission Admin and the Conditional Permission Admin that supersedes the former one.  You can also specify a Java policy file for security.  The recommended usage is that of the Conditional Permission Admin; it is newer and more powerful.  Conditional Permission Admin service extends the Permission Admin service with permissions that can apply when certain conditions are either true or false at the time the permission is checked. These conditions determine the selection of the bundles to which the permissions apply. Permissions are activated immediately after they are set. |
| Package Admin | A framework service allows bundle programmers to inspect the packages exported in the framework and eagerly update or uninstall bundles. |
| Permission Admin | The Permission Admin service enables the OSGi framework management agent to administer the permissions of a specific bundle and to provide defaults for all bundles. A bundle can have a single set of permissions that are used to verify that it is authorized to execute privileged code. You can dynamically manipulate permissions by changing policies on the fly and by adding new policies for newly installed components. Policy files are used to control what bundles can do. |
| Start Level | The core specification controls for start levels are as follows:   * The beginning start level of the OSGi framework * Can modify the active start level of the framework * Can be used to assign a specific start level to a bundle * Can set the initial start level for newly installed bundles |
| Service hooks | The OSGi service hooks are the framework primitives for service interactions; those primitives are as follows:   * Register a service (Publish) * Find services (Find) * Get a service (Bind) * Listen for service events |
| URL Handlers | The URL Handlers service extends the standard Java URL stream and content handler mechanism to work in an OSGi environment.  The way that the built-in URL protocol and content handlers are discovered is by probing packages for the appropriate classes to handle the protocol/content.  If someone tries to create a URL for the HTTP protocol, then the class to handle the protocol will be sun.net.www.protocol.http.Handler as it is registered by default from a Sun JVM. |

The preceding table provides a concise introduction to the framework services. Our experience in using OSGi environments, however, encourages us to further explore Service hooks as an area that requires more attention.

### Service Hooks

These Service Hooks are not intended for regular bundle developers, they are there to facilitate things like distributed OSGi. Service Hooks are not to be confused with the service engine publish, find, and bind methods.

A common usage scenario for a Service Hook is in an OSGi system where all communication is normally tunneled through services; this makes it a very interesting place for a handler to intercept the service communications. The hooks allow you to install handlers that can help facilitate things like proxying, security, authentication, and other functions more or less like interceptors.

This behavior will be completely transparent to the consumer of the service that will only be interacting with the OSGi service registry.

To proxy an existing service for a specific bundle, we would be required to perform the following steps:

1. Hide the existing service X.
2. Register a proxy X with the same properties of X.

Properties here are very simple; we really are only talking about the same interface and potential filters necessary. When these criteria are met, a proxy can pose as the original service and add additional work to the registration.

## 4 – OSGi Compendium Services

In attempting to keep our view of OSGi simple, we've tried to keep our review to the core OSGi specification; however, this leaves out the richness found in the OSGi compendium. All of these additional services will allow you to enhance the bundle life cycle, control, and manage various things such as dependency injection, configuration metadata, user administration, and so on. To discover these services more in detail, please visit <http://www.osgi.org/Download/Release4V42>.

| OSGi Compendium Service | Purpose |
| --- | --- |
| Application Admin | Application manager abstraction is used to manage application types. |
| Blueprint Container | This is a dependency injection framework based on the Spring DM programming model. It has been designed to handle OSGi's dynamic environment where services come and go. |
| Configuration Admin | This is used for handling bundle configuration data. It is commonly used for setting up port allocations, setting URLs, and other variables. Use this service aide in providing a dynamic execution environment. |
| Declarative Services | A component model is used to simplify making components that publish or reference OSGi services. |
| Deployment Admin | This provides standardized access to the life cycle management of resources in an OSGi environment. This service helps to maintain the overall consistency of the runtime. |
| Device Access | This service coordinates adding and removing devices, and provisioning of their drivers. This helps to facilitate a hot deploy, or a plug and play model. |
| DMT Admin | A generic Device Management Tree API is provided to manage devices by mapping the generic tree to specific device functions. |
| Event Admin | This is a high-capacity event service for inter-bundle communication utilizing a publish and subscribe model. |
| Foreign Application Access | This service provides a mechanism to allow non-OSGi Java applications to interoperate with the OSGi environment. |
| HTTP | The HTTP service provides support for registering servlets and resources. This allows users to access, retrieve information from, and control the OSGi environment. |
| Initial Provisioning | A specification that defines how a management agent becomes part of and interacts with the OSGi environment. |
| IO Connector | This is a basic communication infrastructure based on the J2ME javax.microedition.io package. |
| Metatype | This allows services to specify datatypes they can use. Key/value pairs are used to represent data attributes. |
| Monitor Admin | It defines how a bundle may publish status variables, and how administrative bundles can discover and use their values. |
| Preferences | It provides bundles as a mechanism to persist data through starting and stopping of the bundle, or of the OSGi environment. This service is not intended for large quantities of data such as documents or images, but for preferences or setting values (properties). |
| User Admin | This service manages the persistent storage of user credentials, and their attributes, providing an authentication service for end users and/or devices that need to initiate actions in the OSGi environment. |
| UPnPTM | It defines how the OSGi environment can interoperate with **Universal Plug and Play (UPnPTM)** devices and control points. |
| Wire Admin | An administrative service that provides control over the dynamic wiring of producers to their consumers in an OSGi environment. |
| XML Parser | This service covers how classes in JAXP can be used in an OSGi environment. |

While the preceding table is helpful in quickly understanding the OSGi compendium, we feel that we should look a little closer at the Blueprint Container and Configuration Admin.

### Blueprint Container

OSGi Service Platform Release 4 Version 4.2 specifications introduced the Blueprint Container specification.

This specification describes how declarative programming and dependency injection is done in an OSGi container. There are two separate implementations of the Blueprint specification, one from the Apache Foundation—Apache Aries Blueprint—as well as one from the Eclipse Foundation—Eclipse Gemini. The examples in this book and the demonstration code were tested using the Apache Aries version.

Blueprint is built around an OSGi extender pattern. Once a bundle has resolved its dependencies, it is up to the Blueprint extender to do the following:

* Parse the Blueprint XML files
* Instantiate recipes
* Wire the components together
* Manage services' registrations
* Look up service references

A Blueprint file's basic building blocks are the beans, shown as follows:

Beans can also have properties; these properties can be references or values, as shown in the following screenshot:

A Blueprint Container also provides a model for interaction with the OSGi service registry allowing you to define beans that can then be exported.

As seen from earlier examples, services are exported and found via their interface class.

Once we have a service that we want to consume, we can do so from another bundle by referencing it across the service registry with a reference tag, as follows:

These are the basic building blocks in Blueprint; the specification also provides for namespace handlers so that developers can extend the container with a specific behavior for named beans. This is used quite extensively in projects such as Apache Camel, Apache CXF, and of course Apache Aries.

Utilizing the namespace handler techniques, the container is enriched with web service wiring, context resolution, configuration admin integration, and property injection.

### Configuration Admin

One of the most powerful of all services in the OSGi environment is the Configuration Admin service. It is a "merge" between the simple paradigm of reading configuration data at startup time combined with the fact that the said data can and will change during an application's life cycle. In a fully dynamic environment, your configuration can change at any time and you're expected to react to these changes; the Configuration Admin API classes allow you to do exactly this; your bundles will be notified of new, updated, and removed configuration data.

This also allows for some very useful patterns you can build on top of org.osgi.service.cm.Managed ServiceFactories. Apache Felix provides configuration interfaces via fileinstall as well as the webconsole.

## 5 – OSGI and modular patterns

In this section we'll discuss several OSGI and modular programming patterns that we believe you should follow, which will help in producing successful projects.

### Whiteboard pattern

A whiteboard pattern is a very well documented pattern that has a detailed description on the OSGi forum, <http://www.osgi.org/wiki/uploads/Links/whiteboard.pdf>. It is somewhat similar to an extender pattern but relies on the OSGi service registry instead of raw bundles.

#### Idea

Java has, since 1.0, had an event platform. These events unfortunately could lead to fairly cumbersome development cycles with more than 130 events and adapters available already in Java 1.3. The whiteboard pattern provides events in a simple manner without forcing a listener pattern to be implemented. This is done relying on the OSGi service registry for informational messages and further processing.

#### Implementation

A whiteboard pattern in its simplest form is implemented via BundleActivator and the registration of a ServiceTracker object at<http://www.osgi.org/javadoc/r4v42/org/osgi/util/tracker/ServiceTracker.html>.

ServiceTracker gives us an implementation that correctly handles all the details of listening to ServiceEvents and getting and ungetting services. It is also a thread-safe class so it will aid bundle developers in what otherwise would be a fairly cumbersome process involving quite a bit of manual checking of services and registrations.

The whiteboard service tracker bundle subscribes to service registration information, as shown in the following screenshot:

Here we are registering ServiceTracker on a bundle context, we give it an HttpServlet interface to match against and by providing a null as the last argument we are saying that we want to be notified of every single event.

With our listener installed, we are now ready to start looking for these services being added, removed, suspended, and so on. If we say that a servlet example would be deployed in blueprint for the sake of argument, that deployment would look something like the following screenshot:

It would now be up to our ServiceTracker bundle to grab this service, and publish this in a servlet container under the path /myservlet.

As you can see, this is a very graceful, not to mention useful, "Factory" instantiation pattern that will allow for full dynamism, thanks to the nature of OSGi bundles and the service registry.

#### Common uses

Whiteboard patterns are used quite frequently in OSGi. Some examples would be generic commands, such as the original PAX web extender for servlet enhancement of an OSGi container. Refer to <http://karaf.apache.org/manual/2.2.6/users-guide/http.html>.

### Extender pattern

The OSGi extender pattern is one of the most used and implemented strategies for framework enhancement. It is a pattern that models itself towards extensive enhancement of deployments and is used to provide EE behavior in the core of many of today's most advanced Java containers. An extender pattern allows the enhancing of bundles with a custom life cycle that will react as bundles come and go. Prominent implementations of the extender pattern are Apache Aries Blueprint and Spring Dynamic Modules; they both rely on this pattern to control their application participants.

#### Idea

The basic idea around the extender pattern is for the developer of the "extender" to take advantage of the event information and life cycle inherently available to a bundle. We know when a bundle is resolved, we know when it is started, and we know when it is starting. This all leads us to using the OSGi container and runtime; we utilize all of the information we get from the environment to allow us to control and influence the life cycle of other bundles. These bundles in turn provide the extender with custom information, such as OSGi headers or XML files placed in a specific location.

#### Implementation

The OSGi life cycle allows a bundle, very much like the activator, to participate and listen to events regarding the installation, update, and removal of other bundles. This life cycle is dynamic in nature. If we were to just gather the event information, we'd do so utilizing a normal org.osgi.framework.BundleListener interface. Refer to<http://www.osgi.org/javadoc/r4v43/core/org/osgi/framework/BundleListener.html>.

The regular BundleListener is an asynchronous ordered implementation that cannot be called concurrently; it is up to the framework to call our BundleListener interface and dispatch information to us. As seen from this description, the BundleListener interface lends itself to logging and informational purposes but there is alsoorg.osgi.framework.SynchronousBundleListener. Refer to <http://www.osgi.org/javadoc/r4v43/core/org/osgi/framework/SynchronousBundleListener.html>.

This listener actually allows us to take an action on these events, as seen in JavaDocs for this class. Unlike normal BundleListener objects,SynchronousBundleListeners are synchronously called during bundle life cycle processing.

The bundle life cycle processing will not proceed until all SynchronousBundleListeners have completed. SynchronousBundleListener objects will be called prior toBundleListener objects. The following diagram shows the extender pattern:

Armed with this class, we suddenly have a fairly powerful mechanism for building completely new flows for our bundles. If we go back to the Apache Aries Blueprint mentioned earlier, in essence it is really a listener, and when our bundle goes active, it will look for files in OSGI-INF/blueprint ending in .xml.

The synchronous invocation gives us the ability to do things before the actual event has passed. So we can perform initialization, register services, evaluate files, check for resources, and manipulate bundle information so that we can fully take control of the execution environment.

Depicted on the left, the extender bundle that is controlling the flow of information is going to utilize org.osgi.framework.BundleEvent as it receives and looks for bundles containing Type A information. Once it finds a matching bundle, the extender has control of the bundle providing the event, so it can instantiate classes, set up application contexts, and register further metadata and information before handing back control of the event to the framework.

#### Common uses

The extender pattern is used to implement Apache Aries Blueprint, Spring Dynamic Modules as well as Apache Felix Declarative services. It is typically used to extend functionality into an OSGi container. A slight word of caution should be raised as it can become something of a slight anti-pattern, especially if the extenders start manipulating the normal class-loading mechanisms or startup sequencing.