An introduction to creating OSGi applications in Rational Application Developer Version 8.0

September 2010
Overview

The OSGi applications framework provides a programming model for developing, assembling, and deploying modular applications that use Java™ EE and OSGi technologies. OSGi application development tools provide a way to build enterprise applications that benefit from the modularity, dynamism, versioning, and third-party library integration provided by the OSGi applications framework.

Using Rational Application Developer, you can develop the following Java EE applications as OSGi applications:

- Web applications
- Java Persistence API (JPA) applications
- JavaServer Faces applications
- XML applications

OSGi is a module system that is compatible with Java-based systems and implements a dynamic component model. Enterprise systems can use OSGi to improve the maintainability of runtime infrastructures. Applications, in the form of bundles, can be remotely installed, started, stopped, updated, and uninstalled without requiring a reboot.

**OSGi features**

OSGi tools include the following major features:

**Container for OSGi Blueprint components**
The OSGi application framework includes the Apache Software Foundation's Aries open implementation of the OSGi Version 4.2 Blueprint component model that defines a standard dependency injection mechanism for Java™ components. The implementation is derived from the Spring Framework and extended for OSGi to declaratively register component interfaces as services in the OSGi service registry.

**Model for assembling bundles**
The OSGi tools include a model for assembling an application into a deployable unit. The unit can consist of multiple bundles and includes the metadata that describes the version and external location of the constituent bundles of the application.

**Runtime components**
The OSGi tools support the development of OSGi applications that run in an OSGi framework, exploiting enterprise Java technologies common in web
applications and integration scenarios including web application bundles, remote services integration, and JPA.

Extensions
The OSGi tools include extensions that go beyond the OSGi Enterprise Expert Group specifications to provide a more complete integration of OSGi modularity with Java enterprise technologies. In particular it delivers support that includes but is not restricted to the following features:

- Isolated enterprise applications composed of multiple, versioned bundles with dynamic lifecycle.
- Declarative transactions and security for Blueprint components.
- Container-managed JPA for Blueprint components.
- Message-driven Blueprint components.
- Configuration of resource references in module Blueprint Services.
- Annotation-based Blueprint configuration.
- Federation of lookup mechanisms between local JNDI and the OSGi service registry.
- Fully declarative application metadata to enable reflection of an SCA component type definition.

Benefits of OSGi
OSGi modularity provides standard mechanisms to address the issues faced by Java EE applications. The OSGi framework provides the following benefits:

- Applications are portable, easier to re-engineer, and adaptable to changing requirements.
- The framework provides the declarative assembly and simplified unit test of the Spring Framework, but in a standardized form that is provided as part of the application server runtime rather than being a third-party library deployed as part of the application.
- The framework integrates with the Java EE programming model, giving you the option of deploying a web application as a set of versioned OSGi bundles with dynamic lifecycle.
- It supports administration of application bundle dependencies and versions, simplifying and standardizing third-party library integration.
- The framework provides isolation for enterprise applications that are composed of multiple, versioned bundles with dynamic life cycles.
- It has a built-in bundle repository that can host common and versioned bundles shared between multiple applications, so that each application does not deploy its own copy of each common library.
• OSGi applications can access external bundle repositories.
• The framework reinforces service-oriented design at the module level.
• OSGi applications can be composed of coarser-grained SCA assemblies

The OSGi specification

OSGi specifications are defined and maintained by the OSGi Alliance, an open standards organization. The specification outlines open standards for the management of voice, data, and multimedia wireless and wired networks. The OSGi Service Platform Specification defines an open common architecture for service delivery and management using bundles.

The OSGi Applications framework provides a programming model for developing, assembling, and deploying, as bundles, modular applications that use both Java™ EE and OSGi technologies.

OSGi Service Platform Specifications Version 4.2 bring the benefits of OSGi to the Java EE application developer. The OSGi Version 4.2 standard defines the Blueprint component model. This model defines how you can exploit OSGi modularity in your applications, in particular to help with third-party library integration and versioning. The OSGi Applications framework in WebSphere® Application Server includes the following major features:

• It implements a set of application-centric enterprise OSGi technologies that are used by Java application components.
• It advocates and extends the use of the Blueprint component model.
• It builds on the Apache Software Foundation Aries project, which provides an open implementation of the Blueprint container.

Enterprise OSGi

OSGi for Java enterprise applications is the focus of Version 4.2 of the OSGi specification.

Version 4.2 of the OSGi specification includes the definition of the Blueprint component model, a standardized version of the Spring Framework assembly model. The Blueprint component model describes how components can be wired together within a bundle and how configurations and dependencies are injected by a Blueprint component container in the runtime environment.

Components and the references they consume are declared in an XML module
Blueprint file that is a standardization of the Spring application context. The file is extended for the OSGi environment so that components can be automatically published as services for the service registry, and references can be automatically resolved as services discovered from the service registry.

The Blueprint component model provides the simplicity of the Spring Framework, including its ability to form a unit test that is separated from the server environment. Blueprint standardizes the configuration metadata, and therefore brings governance to the specification of the component model.

**OSGi architecture**

The core part of the OSGi Service Platform defines a secure and managed Java-based service platform that supports the deployment of extensible and downloadable applications known as bundles. The specification defines a security model, an application life cycle management model, a service registry, an Execution environment, and Modules.

OSGi defines a dynamic module system for Java™. The core OSGi Service Platform has a layered architecture, and is designed to run on a variety of standard Java profiles. OSGi introduces the notion of a bundle as a modular unit, and the platform architecture is based upon bundles as the unit of deployment. The OSGi architecture has the following layers:

- Execution environment layer
- Module layer
- Life cycle layer
- Service registry layer

**Execution environment layer**

The execution environment layer specifies the Java environment (for example, Java EE or Java SE) under which a bundle will run. For OSGi applications running in WebSphere® Application Server, you do not need to specify the execution environment.

Module layer

The module layer is where the OSGi Framework processes the modular aspects of a bundle. The metadata that enables the OSGi Framework to do this is set in a bundle manifest file.
One of the key advantages of OSGi is its class loader model, which uses the metadata in the manifest file. OSGi does not have a global class path. When bundles are installed into the OSGi Framework, the metadata is processed by the module layer and the declared external dependencies are reconciled against the versioned exports declared by other installed modules. The OSGi Framework determines the dependencies based on the manifest, and calculates the independent required class path for each bundle. This approach resolves the shortcomings of plain Java class loading by ensuring that the following requirements are met:

- Only packages explicitly exported by a particular bundle, through the metadata, are visible to other bundles for import.
- Each package can be resolved to specific versions.
- Multiple versions of a package can be available concurrently to different clients.

**Life cycle layer**

The bundle life cycle management layer in OSGi eliminates the problem with Java class loading and the "class not found" exception at run time, in which dependant classes cannot be loaded because they cannot be found. When an installed bundle is deployed into the framework, the framework first resolves all of its declared dependences. If there are unresolved dependences, the framework reports these and does not start the bundle.

In the bundle life cycle:

- Bundles are dynamic, and can be started and stopped independent of the rest of the framework.
- Each bundle can provide a bundle activator that the framework calls on start and stop events. This is declared in the bundle manifest.

Applications typically do not need to provide a bundle activator. However, if initialization is required when the bundle starts or stops, a bundle activator can be created.

**Service registry layer**

The service registry layer in OSGi intrinsically supports service orientated architecture (SOA). Bundles publish services to the service registry, and other bundles can discover these services from the service registry.
These services are the primary means of collaboration between bundles. An OSGi service is a Plain Old Java Object (POJO), published to the service registry under one or more Java interface names, with optional metadata stored as custom properties (name/value pairs). A discovering bundle looks up a service in the service registry by an interface name, and then can potentially filter the services based on the custom properties.

Services are fully dynamic and typically have the same lifecycle as the bundle that provides them.

**OSGi bundles**

An OSGi bundle is a Java™ archive file that contains Java code, resources, and a manifest that describes the bundle and its dependencies. The bundle is the unit of deployment for an application.

**OSGi bundle manifest file**

An OSGi bundle JAR file contains a JAR manifest file. This file contains metadata that enables the OSGi Framework to process the modular aspects of the bundle.

The following code is an example of the contents of a bundle manifest file, META-INF/MANIFEST.MF:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: MyService bundle
Bundle-SymbolicName: com.sample.myservice
Bundle-Version: 1.0.0
Bundle-Activator: com.sample.myservice.Activator
Import-Package: org.apache.commons.logging;version="1.0.4"
Export-Package: com.sample.myservice.api;version="1.0.0"
```

**OSGi Composite bundles**

Composite bundles group bundles into aggregates to ensure consistent behavior. A composite bundle contains bundles or references to bundles outside
of the workspace or target platform. A composite bundle ensures consistent behavior from a set of shared bundles at a specific version.

A composite bundle archive (CBA) groups shared bundles together into aggregates. A CBA can contain OSGi bundles or reference bundles that are hosted in the internal bundle repository. Create a CBA when you want to ensure consistent behavior across a set of shared bundles. You can use the CBA to wire that set of bundles, at a specific version, to an application.

A CBA is an archive file with a .cba file extension. It contains a composite manifest META-INF/COMPOSITEBUNDLE.MF which defines the CBA and optionally some OSGi bundles with which to seed the repository. The bundles that a CBA contains or references are defined with exact versions, in contrast to an EBA, where bundles are defined with version ranges.

A composite bundle is installed in the internal bundle repository of the runtime. If the CBA directly contains OSGi bundles, these bundles are installed into the repository as though they were individually uploaded. The CBA is also added to the bundle repository. If the CBA references OSGi bundles, these bundles must be present in the internal bundle repository.

After a CBA is installed in the internal bundle repository, its bundles are available to all applications that want to use the bundles when the application is resolved. If a required package or service is available at the same version from both a bundle and a CBA, the provisioning process selects the package or service from the CBA.

A CBA has the following differences from an enterprise bundle archive (EBA) file:

- A CBA has a composite manifest, which is a modularity statement that asserts that bundles can be deployed, not that they will resolve. An EBA file has an application manifest, which is a provisioning statement.
- A CBA can import or export packages, but an EBA file cannot.
- An EBA file does not need to fully define its content. When the application is deployed, a dependency analysis is performed and additional bundles can be provisioned.
- An EBA file can define that bundles in its content are shared.
- Bundles in a CBA are visible in the repository, but bundles in an EBA file are private to the application.
Tools for OSGi application development

The Rational Application Development OSGi tools help you to create OSGi applications.

The development environment contains a selection of views, wizards, and editors that are customized to be the most useful for an OSGi developer.

The editors range from standard source editors with content assist features to more full featured editors, such as the following:

- Bundle manifest editor helps you to describe your bundle and bundle dependencies. For more information, refer to OSGi bundle manifest file.
- OSGi composite bundle manifest editor helps you describe your composite bundle and bundle dependencies.
- Blueprint XML editor helps you to define and describe the various components of an application.
- OSGi application manifest editor helps you define and describe the metadata that enables the framework to process the modular aspects of your bundles.
- OSGi Bundle Explorer visualizes your bundles and the dependencies between them.

For more information

To learn more about Web Services and Rational Application Developer, see these resources:

- Rational Application Developer Web site:
- Rational Application Developer Information Center:
- Rational Application Developer wiki:
  https://www.ibm.com/developerworks/wikis/display/rad/Home