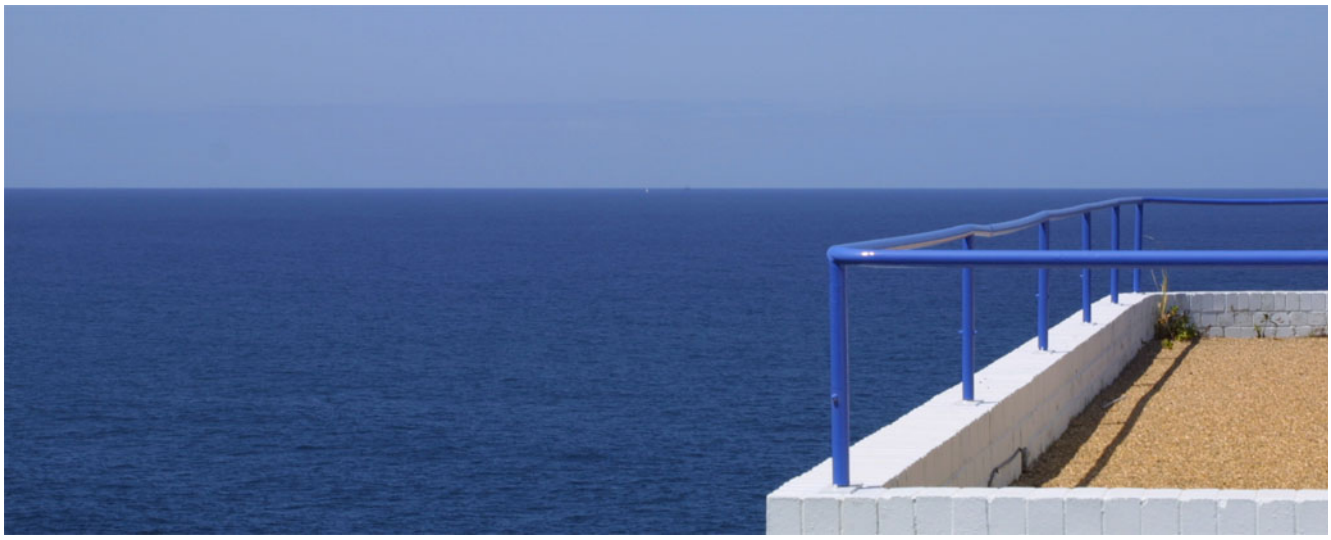

User Guide

Version:	1.0.0
Date:	7/22/11
Creator:	Thomas Diesler

Table of Contents

1. Introduction

1.1. What is OSGi

The OSGi specifications define a standardized, component-oriented, computing environment for networked services that is the foundation of an enhanced service-oriented architecture.

Developing on the OSGi platform means first creating your OSGi bundles, then deploying them in an OSGi Framework.

What does OSGi offer to Java developers?

OSGi modules provide classloader semantics to partially expose code that can then be consumed by other modules. The implementation details of a module, although scoped public by the Java programming language, remain private to the module. On top of that you can install multiple versions of the same code and resolve dependencies by version and other criteria. OSGi also offers advanced lifecycle and services layers, which are explained in more detail further down.

What kind of applications benefit from OSGi?

Any application that is designed in a modular fashion where it is necessary to start, stop, update individual modules with minimal impact on other modules. Modules can define their own transitive dependencies without the need to resolve these dependencies at the container level.

1.2. OSGi Framework Overview

The functionality of the Framework is divided in the following layers:

- Security Layer (optional)
- Module Layer
- Life Cycle Layer
- Service Layer
- Actual Services

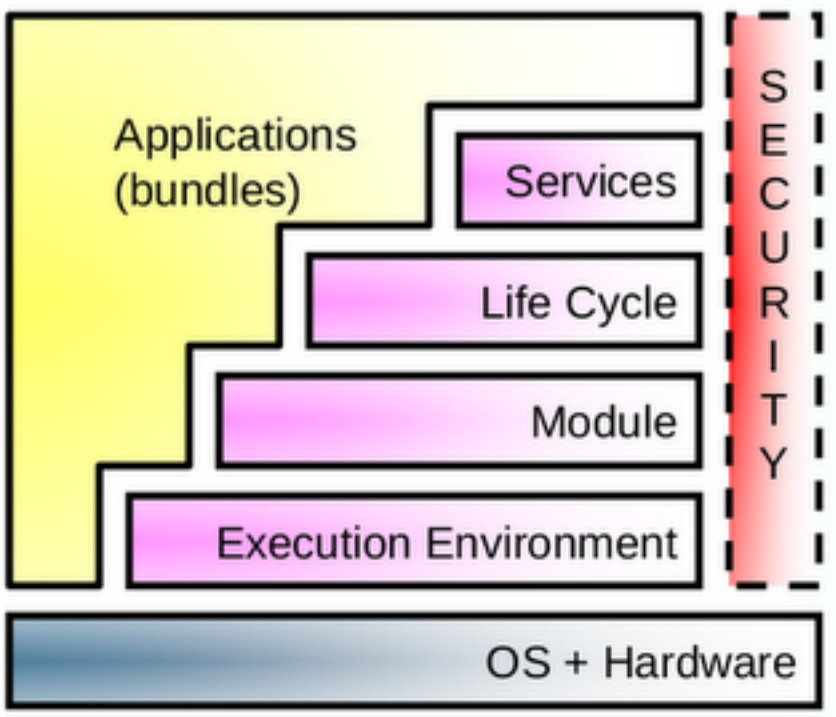

Figure 1. Source: OSGi Alliance

OSGi Security Layer

The OSGi Security Layer is an optional layer that underlies the OSGi Service Platform. The layer is based on the Java 2 security architecture. It provides the infrastructure to deploy and manage applications that must run in fine grained controlled environments.

The OSGi Service Platform can authenticate code in the following ways:

- By location
- By signer

For example, an Operator can grant the ACME company the right to use networking on their devices. The ACME company can then use networking in every bundle they digitally sign and deploy on the Operator's device. Also, a specific bundle can be granted permission to only manage the life cycle of bundles that are signed by the ACME company.

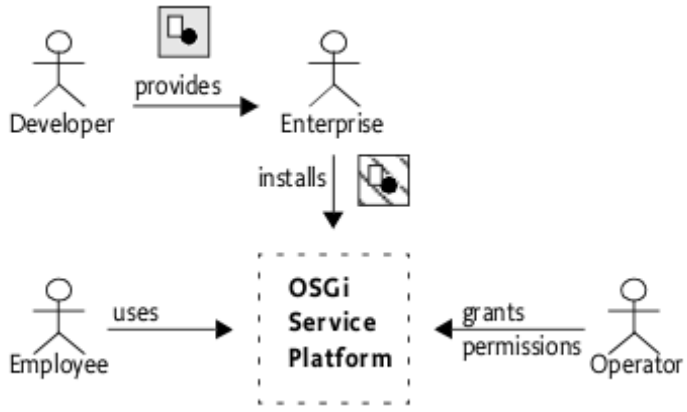

Figure 2. Source: OSGi Alliance

The current version of JBoss OSGi does not provide this optional layer. If you would like to see this implemented, let us know on the forums: http://community.jboss.org/en/jbossosgi.

OSGi Module Layer

The OSGi Module Layer provides a generic and standardized solution for Java modularization. The Framework defines a unit of modularization, called a bundle. A bundle is comprised of Java classes and other resources, which together can provide functions to end users. Bundles can share Java packages among an exporter bundle and an importer bundle in a well-defined way.

Once a Bundle is started, its functionality is provided and services are exposed to other bundles installed in the OSGi Service Platform. A bundle carries descriptive information about itself in the manifest file that is contained in its JAR file. Here are a few important Manifest Headers defined by the OSGi Framework:

- **Bundle-Activator** - class used to start, stop the bundle
- **Bundle-SymbolicName** - identifies the bundle
- **Bundle-Version** - specifies the version of the bundle
- **Export-Package** - declaration of exported packages
- **Import-Package** - declaration of imported packages

The notion of OSGi Version Range describes a range of versions using a mathematical interval notation. For example

```
Import-Package: com.acme.foo;version="[1.23, 2)", com.acme.bar;version="[4.0, 5.0)"
```

With the OSGi Class Loading Architecture many bundles can share a single virtual machine (VM). Within this VM, bundles can hide packages and classes from other bundles, as well as share packages with other bundles.

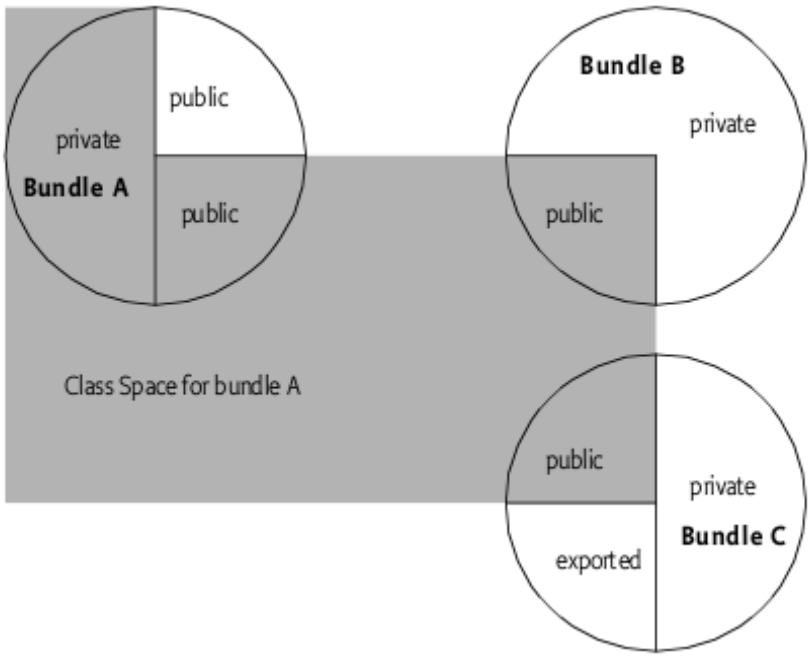

Figure 3. Source: OSGi Alliance

For example, the following import and export definition resolve correctly because the version range in the import definition matches the version in the export definition:

```
A: Import-Package: p; version="[1,2)"
```

```
    B: Export-Package: p; version=1.5.1
```

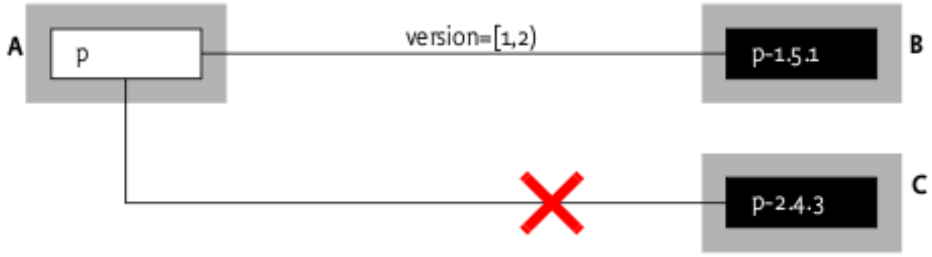

Figure 4. Source: OSGi Alliance

Apart from bundle versions, OSGi Attribute Matching is a generic mechanism to allow the importer and exporter to influence the matching process in a declarative way. For example, the following statements will match.

```
A: Import-Package: com.acme.foo;company=ACME
    B: Export-Package: com.acme.foo;company=ACME; security=false
```

An exporter can limit the visibility of the classes in a package with the include and exclude directives on the export definition.

```
Export-Package: com.acme.foo; include:="Qux*,BarImpl"; exclude:=QuxImpl
```

OSGi Life Cycle Layer

The Life Cycle Layer provides an API to control the security and life cycle operations of bundles.

A bundle can be in one of the following states:

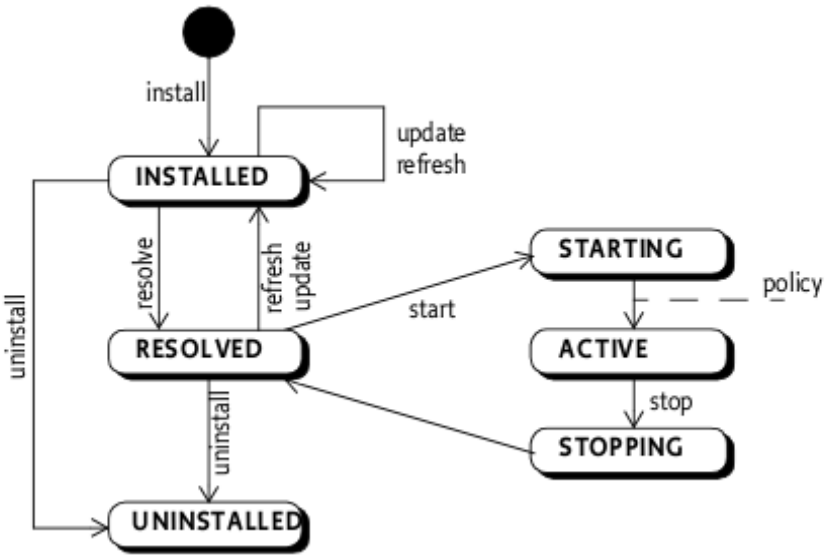

Figure 5. Source: OSGi Alliance

A bundle is activated by calling its Bundle Activator object, if one exists. The BundleActivator interface defines methods that the Framework invokes when it starts and stops the bundle.

A Bundle Context object represents the execution context of a single bundle within the OSGi Service Platform, and acts as a proxy to the underlying Framework. A *Bundle Context* object is created by the Framework when a bundle is started. The bundle can use this private BundleContext object for the following purposes:

- Installing new bundles into the OSGi environment

- Interrogating other bundles installed in the OSGi environment
- Obtaining a persistent storage area
- Retrieving service objects of registered services
- Registering services in the Framework service
- Subscribing or unsubscribing to Framework events

OSGi Service Layer

The OSGi Service Layer defines a dynamic collaborative model that is highly integrated with the Life Cycle Layer. The service model is a publish, find and bind model. A service is a normal Java object that is registered under one or more Java interfaces with the service registry. OSGi services are dynamic, they can come and go at any time. OSGi service consumers, when written correctly, can deal with this dynamicity. This means that OSGi services provide the capability to create a highly adaptive application which, when written using services, can even be updated at runtime without taking the service consumers down.

The OSGi Declarative Services[9] and OSGi Blueprint [9] specifications significantly simplify the use of OSGi Services which means that a consumer gets the benefits of a dynamic services model for very little effort.

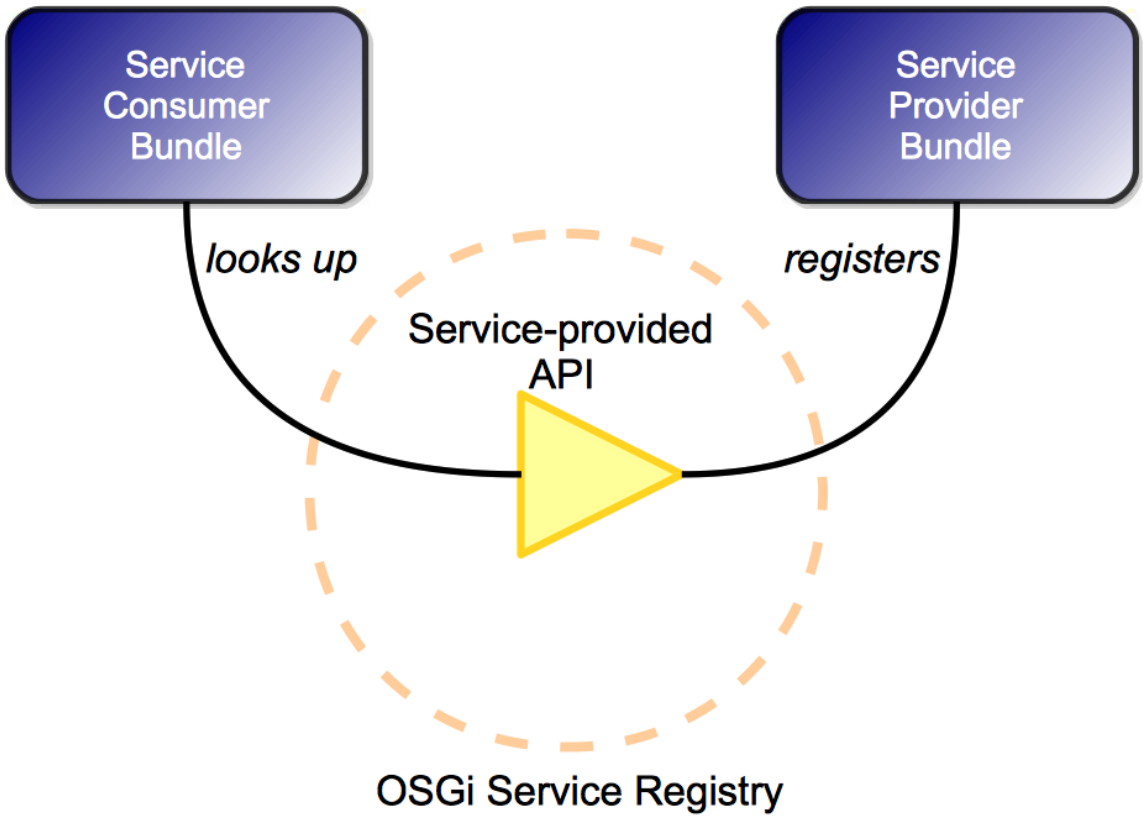

Figure 6. OSGi Services

1.3. OSGi Service Compendium

The OSGi Service Compendium is described in the OSGi Compendium and Enterprise specifications. It specifies a number of services that may be available in an OSGi runtime environment. Although the OSGi Core Framework specification is useful in itself already, it only defines the OSGi core infrastructure. The services defined in the compendium specification define the scope and functionality of some common services that bundle developers might want to use. Here is a quick summary of the popular ones:

Log Service

Chapter 101 in the Compendium and Enterprise specifications.

The Log Service provides a general purpose message logger for the OSGi Service Platform. It consists of two services, one for logging information and another for retrieving current or previously recorded log information.

The JBoss OSGi Framework provides an implementation of the Log Service which channels logging information through to the currently configured system logger.

Http Service

Chapter 102 in the Compendium and Enterprise specifications.

The Http Service supports a standard mechanism for registering servlets and resources from inside an OSGi Framework. This can be used to develop communication and user interface solutions for standard technologies such as HTTP, HTML, XML, etc.

Configuration Admin Service

Chapter 104 in the Compendium and Enterprise specifications.

The Configuration Admin service allows an operator to set the configuration information of deployed bundles.

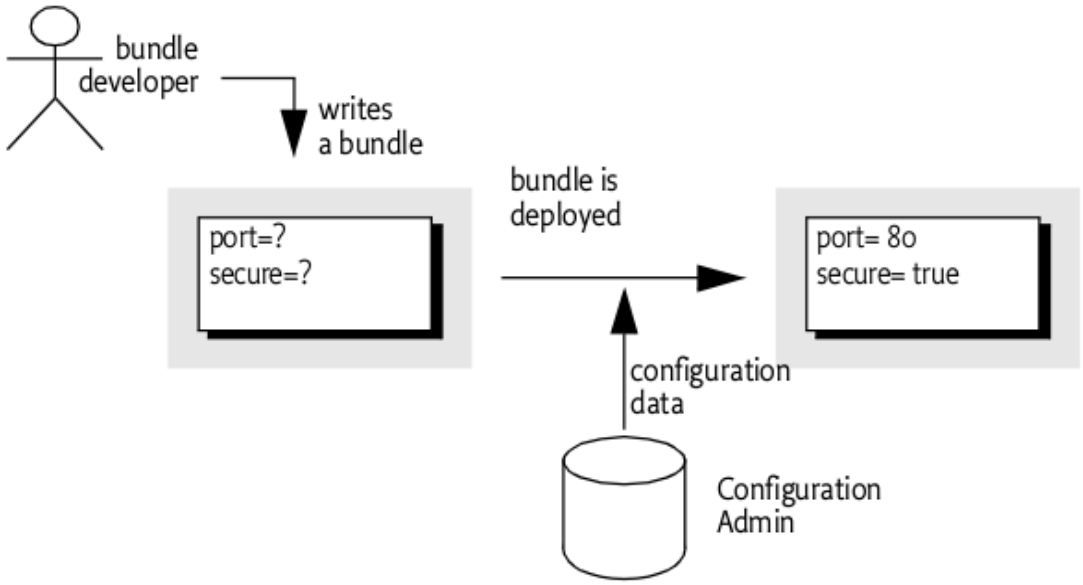

Figure 7. Source: OSGi Alliance

The JBoss OSGi Framework provides an implementation of the Configuration Admin Service which obtains its configuration information from the JBoss Application Server configuration data, for instance the `standalone.xml` file.

Metatype Service

Chapter 105 in the Compendium and Enterprise specifications.

The Metatype Service specification defines interfaces that allow bundle developers to describe attribute types in a computer readable form using so-called metadata. This service is mostly used to define the attributes and datatypes used by Configuration Admin Service information.

User Admin Service

Chapter 107 in the Compendium and Enterprise specifications.

Bundles can use the User Admin Service to authenticate an initiator and represent this authentication as an Authorization object. Bundles that execute actions on behalf of this user can use the Authorization object to verify if that user is authorized.

Declarative Services Specification

Chapter 112 in the Compendium and Enterprise specifications.

The Declarative Services (DS) specification describes a component model to be used with OSGi services. It enables the creation and consumption of OSGi services without directly using any OSGi APIs. Service consumers are informed of their services through injection. The handling of the OSGi service dynamics is done by DS. See also the Blueprint Specification [9].

Event Admin Service

Chapter 113 in the Compendium and Enterprise specifications.

The Event Admin Service provides an asynchronous inter-bundle communication mechanism. It is based on a event publish and subscribe model, popular in many message based systems.

Blueprint Specification

Chapter 121 in the Enterprise specification.

The OSGi Blueprint Specification describes a component framework which simplifies working with OSGi services significantly. To a certain extent, Blueprint and DS [9] have goals in common, but the realization is different. One of the main differences between Blueprint and DS is in the way service-consumer components react to a change in the availability of required services. In the case of DS the service-consumer will disappear when its required dependencies disappear, while in Blueprint the component stays around and waits for a replacement service to appear. Each model has its uses and it can be safely said that both Blueprint as well as DS each have their supporters. The Blueprint specification was heavily influenced by the Spring framework.

Remote Services Specifications

Chapters 13 and 122 in the Enterprise specification.

OSGi Remote Services add distributed computing to the OSGi service programming model. Where in an ordinary OSGi Framework services are strictly local to the Java VM, with Remote Services the services can be remote. Services are registered and looked up just like local OSGi services, the Remote Services specifications define standard service properties to indicate that a service is suitable for remoting and to find out whether a service reference is a local one or a remote one.

JTA Specification

Chapter 123 in the Enterprise specification.

The OSGi-JTA specification describes how JTA can be used from an OSGi environment. It includes standard JTA-related services that can be obtained from the OSGi registry if an OSGi application needs to make use of JTA.

JMX Specification

Chapter 124 in the Enterprise specification.

The OSGi-JMX specification defines a number of MBeans that provide management and control over the OSGi Framework.

JDBC Specification

Chapter 125 in the Enterprise specification.

The OSGi-JDBC specification makes using JDBC drivers from within OSGi easy. Rather than loading a database driver by class-name (the traditional approach, which causes issues with modularity in general and often requires external access to internal implementation classes), this specification registers the available

JDBC drivers under a standard interface in the Service Registry from where they can be obtained by other Bundles without the need to expose internal implementation packages of the drivers.

JNDI Specification

Chapter 126 in the Enterprise specification.

The OSGi-JNDI specification provides access to JNDI through the OSGi Service Registry. Additionally, it provides access to the OSGi Service Registry through JNDI. The special `osgi:` namespace can be used to look up OSGi services via JNDI.

JPA Specification

Chapter 127 in the Enterprise specification.

The OSGi-JPA specification describes how JPA works from within an OSGi framework.

Web Applications Specification

Chapter 128 in the Enterprise specification.

The Web Applications specification describes Web Application Bundles. A WAB is a `.WAR` file which is effectively turned into a bundle. The specification describes how Servlets can interact with the OSGi Service Registry and also how to find all the available Web Applications in an OSGi Framework.

Additionally, the Web Applications spec defines a mechanism to automatically turn an ordinary `.WAR` file into a Web Application Bundle.

Service Tracker Specification

Chapter 701 in the Compendium and Enterprise specifications.

The Service Tracker specification defines a utility class, ServiceTracker. The ServiceTracker API makes tracking the registration, modification, and unregistration of services much easier.

1.4. Where to find OSGi Service implementations

Some, but not all the above specifications are provided with the JBoss OSGi Framework out of the box, however there are many other projects that implement these specifications. If an implementation is written for a compliant OSGi 4.2 framework it should be possible to use it in the JBoss OSGi Framework, as long as this implementation does not depend on any proprietary extensions of existing OSGi Frameworks.

A list of all the available specification implementations can be found on Wikipedia here: http://en.wikipedia.org/wiki/OSGi_Specification_Implementations.

Images courtesy of the OSGi Alliance.

2. Getting Started

This chapter takes you through the first steps of getting JBoss OSGi and provides the initial pointers to get up and running.

The JBoss OSGi Framework can be obtained in the following ways:

- As part of the JBoss Application Server 7 (recommended)
- As a Standalone Distribution

2.1. Download as part of AS7

To get started with OSGi in JBoss AS7, see the AS7 getting started guide: Getting Started Developing Applications Guide.

The remainder of this document focuses on getting started with the Standalone Distribution.

2.2. Download the Standalone Distribution

JBoss OSGi is distributed as an IzPack installer archive. The installer is available from the JBoss OSGi download area.

2.3. Running the Installer

To run the installer execute the following command:

```
java -jar jboss-osgi-installer-1.0.0.Beta10.jar
```

The installer first shows a welcome screen

Figure 8.

Then you select the installation path for the JBoss OSGi distribution. This is the directory where you find the binary build artifacts, the java sources, documentation and the JBoss OSGi Runtime.

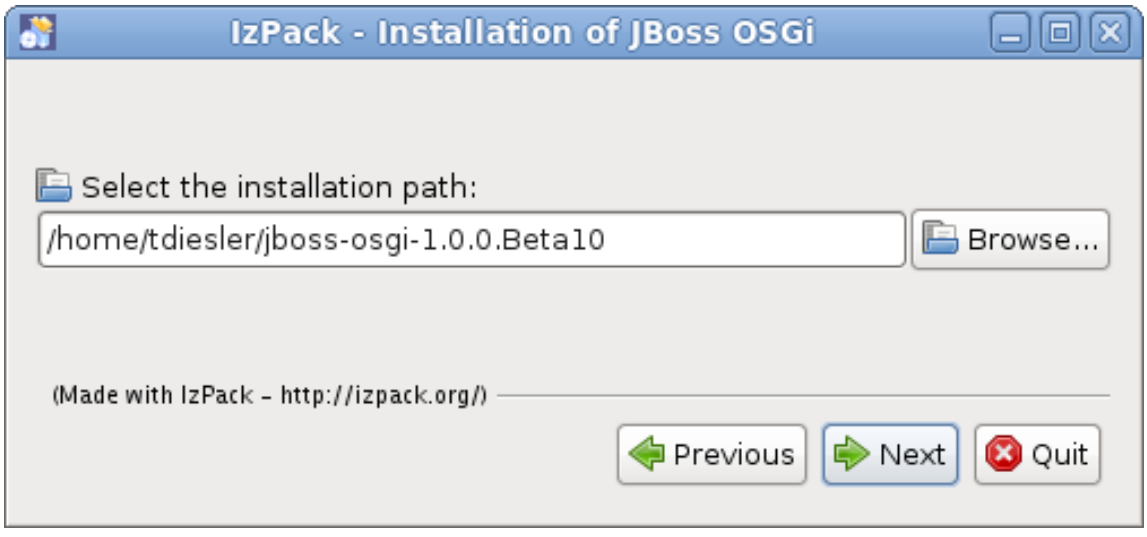

Figure 9.

The installer contains multiple installation packs. Greyed packs are required, others are optional and can be deselected.

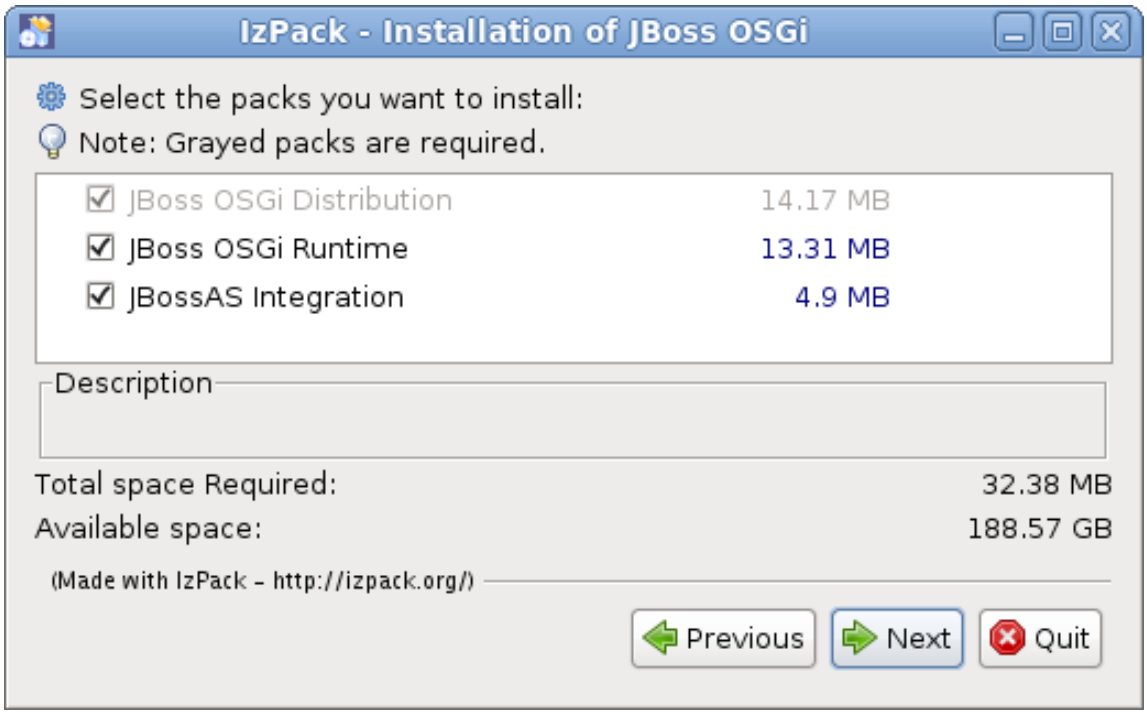

Figure 10.

- JBoss OSGi Distribution - Documentation, Binary Artifacts and Sources
- JBoss OSGi Runtime - Standalone JBoss OSGi Runtime

You can then verify the selected installation options and proceed with the actual installation.

Figure 11.

The installer reports its installation progress and finally displays a confirmation screen. You can now optionally generate an "automatic installation script" that you can use when you want to repeat what you have just done without user interaction.

Figure 12.

2.4. Starting the Runtime

If you selected JBoss OSGi Runtime during installation you should see a *runtime* folder, which contains the JBoss OSGi Runtime distribution. The JBoss OSGi Runtime is an OSGi container onto which services and applications can be deployed.

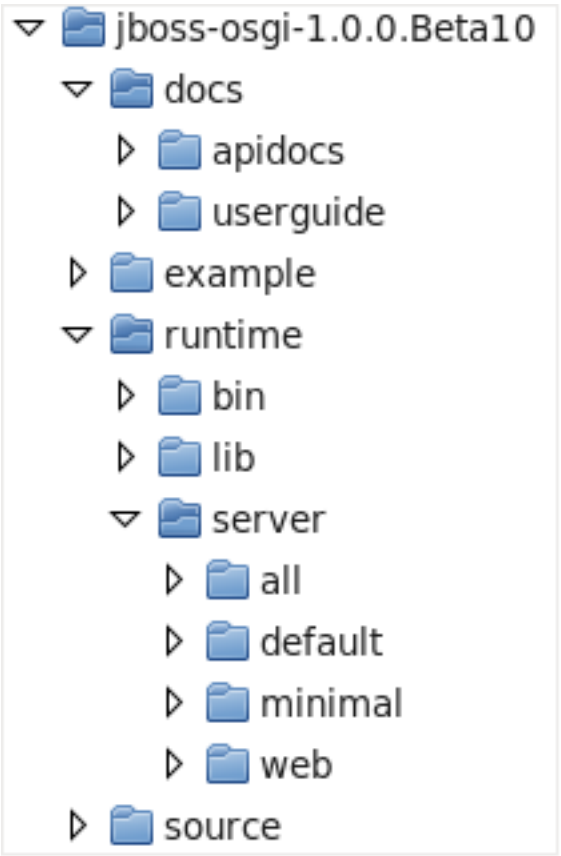

Figure 13.

You can start the Runtime by running **bin/run.sh**. The supported command line options are:

- -c (--server-name) - The runtime profile to start. The default is the 'default' profile.
- -b (--bind-address) - The network address various services can bind to. The default is 'localhost'

```
$ bin/run.sh
================================================================================

  JBossOSGi Bootstrap Environment

  OSGI_HOME: /home/tdiesler/jboss-osgi-1.0.0.Beta10/runtime

  JAVA: /usr/java/jdk1.6/bin/java

  JAVA_OPTS: ...

================================================================================

12:08:37,172 INFO  JBossOSGi Framework Core - 1.0.0.Alpha23
12:08:37,722 INFO  Starting bundles for start level: 1
12:08:37,860 INFO  Bundle started: org.apache.felix.log:1.0.0
12:08:37,904 INFO  Bundle started: jboss-osgi-common:1.0.6
12:08:37,923 INFO  Bundle started: jbosgi-hotdeploy:1.0.9
12:08:37,925 INFO  JBossOSGi Runtime booted in 0.674sec
12:08:38,343 INFO  Bundle started: jboss-osgi-jmx:1.0.9
12:08:38,411 INFO  Bundle started: org.apache.felix.configadmin:1.2.8
12:08:38,567 INFO  Bundle started: org.apache.aries.util:0.1.0.incubating
12:08:38,659 INFO  Bundle started: org.apache.felix.eventadmin:1.2.6
12:08:38,661 INFO  Bundle started: jboss-osgi-common-core:2.2.17.SP1
...
12:08:38,696 INFO  JBossOSGi Runtime started in 0.741000sec
```

2.5. Provided Examples

JBoss OSGi comes with a number of examples that you can build and deploy. Each example deployment is verified by an accompanying test case

- **blueprint** - Basic Blueprint Container examples
- **event** - EventAdmin examples
- **http** - HttpService examples
- **interceptor** - Examples that intercept and process bundle metadata
- **jmx** - Standard and extended JMX examples
- **jndi** - Bind objects to the Naming Service
- **jta** - Transaction examples
- **simple** - Simple OSGi examples (start here)
- **webapp** - WebApplication (WAR) examples
- **xml parser** - SAX/DOM parser examples

For more information on these examples, see the Section 8, “Provided Examples” section.

2.6. Bundle Deployment

Bundle deployment works, as you would probably expect, by dropping your OSGi Bundle into the JBoss OSGi Runtime **deploy** folder.

```
$ cp .../test-libs/example/example-http.jar .../runtime/server/web/deploy
...
13:59:38,284 INFO  [BundleRealDeployer] Installed: example-http [9]
13:59:38,289 INFO  [example-http] BundleEvent INSTALLED
13:59:38,297 INFO  [example-http] BundleEvent RESOLVED
13:59:38,304 INFO  [example-http] ServiceEvent REGISTERED
```

```
13:59:38,306 INFO  [BundleStartStopDeployer] Started: example-http [9]
13:59:38,306 INFO  [example-http] BundleEvent STARTED
```

2.7. Managing installed Bundles

JBoss OSGi comes with a simple Web Console, which is currently based on the Apache Felix Web Console project. The JBoss OSGi Web Console is included in the runtime profiles 'web' or 'all'. After startup you can point your browser to http://localhost:8090/jboss-osgi.

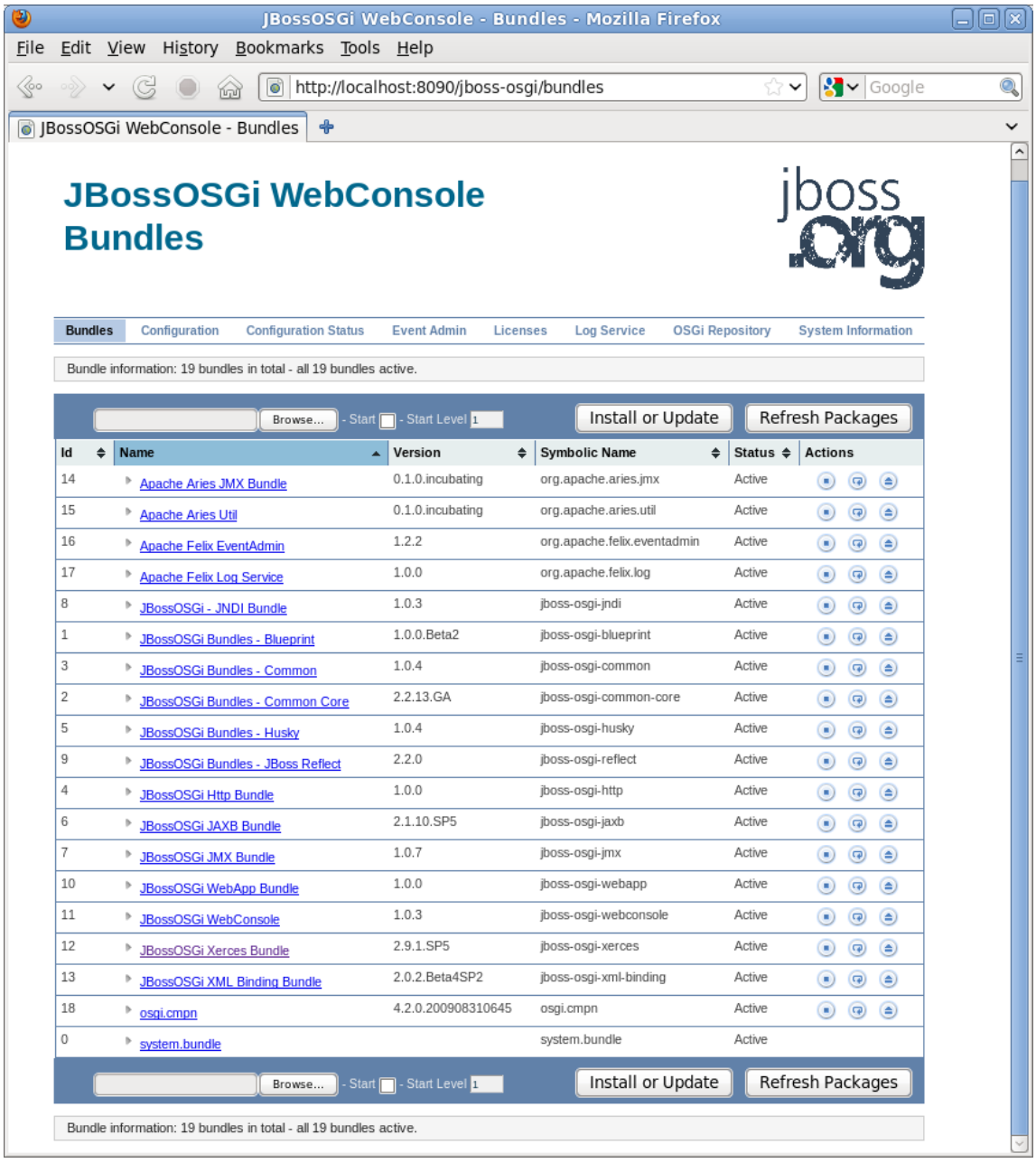

Figure 14.

The Web Console can also be used to install, start, stop and uninstall bundles.

2.8. Hudson QA Environment

Setup the Hudson QA Environment

The JBoss OSGi Hudson QA Environment is an integral part of the JBoss OSGi code base. It is designed for simplicity because we believe that comprehensive QA will only get done if it is dead simple to do so.

Consequently, you only have to execute two simple ant targets to setup the QA environment that was used to QA the JBoss OSGi release that you currently work with.

If in future we should discover a problem with a previous JBoss OSGi release, it will be possible to provide a patch and verify that change using the original QA environment for that release.

With every release we test the matrix of supported target containers and frameworks

Set Hudson Properties

You need to set a few properties

```
$ cd build/hudson
$ cp ant.properties.example ant.properties
$ vi ant.properties

# Tomcat settings
tomcat.base=/usr/share/tomcat6
tomcat.conf=/etc/tomcat6/tomcat6.conf

# SCM settings
# ------------
scm.git.url.jbosgi=git://github.com/jbosgi/jbosgi.git
scm.http.url.jbosgi=http://github.com/jbosgi/jbosgi

# JDK settings
# ------------
java.home.jdk16=/usr/java/jdk1.6.0_21

# Maven settings
# --------------
maven.name=apache-maven-3.0
maven.path=/usr/java/apache-maven

# The JBoss settings
# ------------------
jboss.server.instance=default
jboss.bind.address=127.0.0.1

# Hudson Default settings
# -----------------------
hudson.home=/usr/share/tomcat6/workspace/hudson-home
hudson.version=1.382
github.plugin.version=0.2
git.plugin.version=1.1
```

Run Hudson Setup

```
$ ant hudson-setup
Buildfile: build.xml

init-hudson:
     [echo]
     [echo] hudson.root = /home/hudson/workspace/hudson/jboss-osgi
     [echo] hudson.home = /home/hudson/workspace/hudson/jboss-osgi/hudson-home
     [echo]

...

hudson-setup:
     [copy] Copying 2 files to /home/hudson/workspace/jboss-osgi/hudson-home
     [copy] Copying 13 files to /home/hudson/workspace/jboss-osgi/hudson-home/jobs
     [echo]
     [echo] ************************************
     [echo] * Hudson setup successfully
     [echo] * sudo service tomcat6 restart
     [echo] ************************************
```

3. Standalone Runtime

3.1. Overview

The JBoss OSGi Runtime is an OSGi container onto which components, services and applications can be deployed.

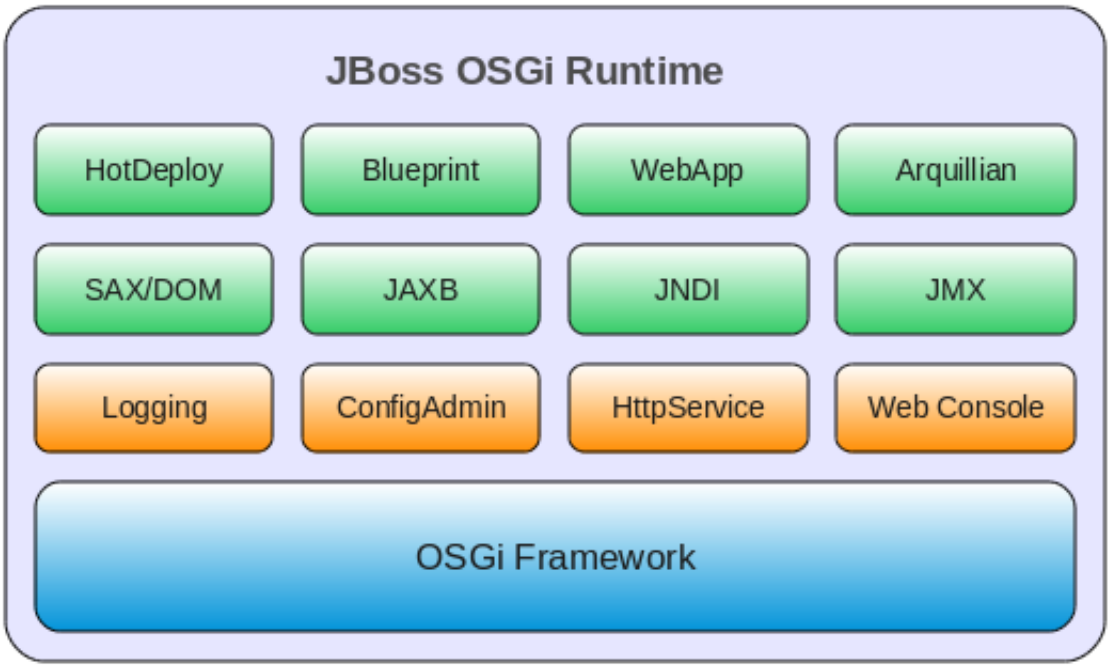

Figure 15.

Preconfigured profiles, contain OSGi bundles that logically work together. A profile can be bootstrapped either as a standalone server or embedded in some other environment. With a startup time of less than 600ms, the runtime can be easily be bootstrapped from within plain JUnit4 test cases.

The JBoss OSGi Runtime has an integration layer for the underlying OSGi framework.

Through local and remote management capabilities the JBoss OSGi Runtime can be provisioned with new or updated bundles. Similar to JBossAS it supports hot-deployment by dropping bundles into the 'deploy' folder. Management of the runtime is provided through a Web Console

JBoss OSGi comes with an implementation of Blueprint Service, which standardizes a POJO programming model without much "pollution" of OSGi specific API.

Great care has been taken about testability of deployed components and services. The Arquillian Test Framework allows you to write plain JUnit tests that do not need to extend any specific test base class. Access to the Runtime has been abstracted sufficiently that you can run the same test case against an embedded (bootstrapped from within the test case) as well as a remote instance of the Runtime. You can run your OSGi tests from Maven, Ant, Eclipse or any other test runner that supports JUnit4.

3.2. Features

The current JBoss OSGi Runtime feature set includes

- **Embedded and Standalone usage** - The runtime can be bootstrapped as standalone container with a start-up time of less than 2 sec in its default configuration or embedded in some other container environment.

- **Various Runtime Profiles** - It comes with the preconfigured profiles 'Minimal', 'Default', 'Web', 'All'. Setting up a new profile is a mater of creating a new directory and putting some bundles in it.
- **Hot Deployement** - Similar to JBossAS there is a deployment scanner that scans the 'deploy' folder for new or removed bundles.
- **Local and Remote JMX Support** - There is local as well as remote JSR160 support for JMX.
- **JNDI Support** - Components can access the JNDI InitialContext as a service from the registry.
- **JTA Support** - Components can interact with the JTA TransactionManager and UserTransaction service.
- **SAX/DOM Parser Support** - The Runtime comes with an implementation of an XMLParserActivator which provides access to a SAXParserFactory and DocumentBuilderFactory.
- **HttpService and WebApp Support** - HttpService and WebApp support is provided by Pax Web.
- **ConfigAdmin Support** - ConfigAdmin support is provided by the Apache Felix Configuration Admin Service.
- **EventAdmin Support** - EventAdmin support is provided by the Apache Felix Event Admin Service.
- **Provisioning** - Bundle provisioning can be done through the JMX based Runtime Managment Interface.
- **Logging System** - The logging bridge writes OSGi LogEntries to the configured logging framework (e.g. Log4J).
- **Blueprint Container Support** - The Blueprint Container service allows bundles to contain standard blueprint descriptors, which can be used for component wiring and injection of blueprint components. The idea is to use a plain POJO programming model and let Blueprint do the wiring for you. There should be no need for OSGi API to "pollute" your application logic.

3.3. Runtime Profiles

A runtime profile is a collection bundles that logically work together. The OSGi runtime configuration contains the list of bundles that are installed/started automatically. You can start create you own profile by setting up a new directory with your specific set of bundles.

A runtime profile can be started using the **-c comand line option**.

```
$ bin/run.sh -c minimal
================================================================================

  JBossOSGi Bootstrap Environment

  OSGI_HOME: /home/tdiesler/jboss-osgi-1.0.0.Beta10/runtime

  JAVA: /usr/java/jdk1.6/bin/java

  JAVA_OPTS: -Dprogram.name=run.sh ...

================================================================================

14:14:38,355 INFO  JBossOSGi Framework Core - 1.0.0.Alpha8
14:14:38,422 INFO  Bundle STARTED: system.bundle:0.0.0
14:14:38,766 INFO  Bundle STARTED: org.apache.felix.log:1.0.0
14:14:38,816 INFO  Bundle STARTED: jboss-osgi-common:1.0.6
14:14:38,867 INFO  Bundle STARTED: jbosgi-hotdeploy:1.0.8
14:14:38,870 INFO  JBossOSGi Runtime booted in 0.514sec
```

Minimal Profile

The 'minimal' profile provides logging and hot-deployment.

The following bundles are installed:

- **jboss-osgi-common.jar** - JBoss OSGi common services
- **jbosgi-hotdeploy.jar** - JBoss OSGi hot deployment service
- **org.apache.felix.log.jar** - Apache LogService
- **org.osgi.compendium.jar** - OSGi compendium API

Default Profile

The 'default' profile extends the 'minimal' profile by JNDI and JMX

These additional bundles are installed:

- **jboss-osgi-common-core.jar** - JBoss Common Core functionality
- **jboss-osgi-jmx.jar** - JBoss OSGi JMX service
- **org.apache.aries.jmx.jar** - Apache Aries JMX services
- **org.apache.felix.configadmin.jar** - Apache Config Admin service
- **org.apache.felix.eventadmin.jar** - Apache Event Admin service

Web Profile

The 'web' profile extends the 'default' profile by HttpService and ConfigAdmin

These additional bundles are installed:

- **jboss-osgi-http.jar** - JBoss OSGi HttpService
- **jbosgi-webapp.jar** - JBoss OSGi WebApp Support
- **jbosgi-webconsole.jar** - JBoss OSGi Web Console

All Profile

The 'all' profile extends the 'web' profile by SAX/DOM, JTA, Blueprint

These additional bundles are installed:

- **arquillian-osgi-bundle.jar** - Arquillian test support
- **jboss-osgi-xerces.jar** - Apache Xerces support
- **jbosgi-blueprint.jar** - Blueprint Container support

4. Application Server Integration

4.1. Overview

The JBoss OSGi framework is fully integrated into the JBoss Application Server 7. OSGi bundles can be deployed like any other deployment that is supported by AS. Hot deployment is supported by dropping an OSGi bundle into the 'deployments' folder. JMX and an OSGi management console is also supported.

OSGi components can interact with non OSGi services that are natively provided by AS. This includes, but is not limited to, the Transaction Service and Naming Service (JNDI).

Standard OSGi Config Admin functionality is supported and integrated with the general AS management layer.

By default the OSGi subsystem is activated on-demand. Only when there is an OSGi bundle deployment the subsystem activates and the respective OSGi services become available.

4.2. Configuration

The OSGi subsystem is configured like any other subsystem in the standalone/domain XML descriptor. The configuration options are:

- **Subsystem Activation** - By default the OSGi subsystem is activated on-demand. The activation attribute can be set to 'eager' to initialize the subsystem on server startup.
- **Framework Properties** - OSGi supports the notion of framework properties. Property values are of type string. A typical configuration includes a set of packages that are provided by the server directly. Please refer to the OSGi core specification for a list of standard OSGi properties.
- **Module Dependencies** - The Framework can export packages from server system modules. The property 'org.jboss.osgi.system.modules' contains a list of module identifiers that are setup as dependencies of the OSGi Framework.
- **Preinstalled Bundles** - OSGi bundles can be installed from their respective module identifier as part of OSGi subsystem startup.
- **Config Admin properties** - Supported are multiple configurations identified by persistent id (PID). Each configuration may contain multiple configuration key/value pairs.Below is a sample configuration for the OSGi subsystem

```
<subsystem xmlns="urn:jboss:domain:osgi:1.0" activation="lazy">
            <configuration
 pid="org.apache.felix.webconsole.internal.servlet.OsgiManager">
                <property name="manager.root">jboss-osgi</property>
            </configuration>
            <properties>
                <property name="org.jboss.osgi.system.modules">
                org.apache.log4j,
                javax.inject.api,
                </property>
                <property name="org.osgi.framework.system.packages.extra">
                org.apache.log4j;version=1.2,
                javax.inject
                </property>
            </properties>
            <modules>
                <module identifier="org.osgi.compendium"/>
                <module identifier="org.apache.aries.util"/>
                <module identifier="org.apache.aries.jmx" start="true"/>
                <module identifier="org.jboss.osgi.jmx" start="true"/>
            </modules>
        </subsystem>
```

For more details on the Application Service integration configuration see AS7 Subsystem Configuration documentation.

4.3. Features

The current JBoss OSGi feature set in AS includes

- **Hot Deployment** - Similar to JBossAS there is a deployment scanner that scans the `deployments` folder for new or removed bundles.
- **Local and Remote JMX Support** - There is local as well as remote JSR160 support for JMX. The OSGi-JMX MBeans are provided through the Apache Aries JMX implementation.

- **JNDI Support** - Components can access the JNDI InitialContext as a service from the registry.
- **JTA Support** - Components can interact with the JTA TransactionManager and UserTransaction service.
- **SAX/DOM Parser Support** - The Runtime comes with an implementation of an XMLParserActivator which provides access to a SAXParserFactory and DocumentBuilderFactory.
- **HttpService and WebApp Support** - HttpService and WebApp support is provided by Pax Web.
- **ConfigAdmin Support** - ConfigAdmin support is provided by the Apache Felix Configuration Admin Service.
- **EventAdmin Support** - EventAdmin support is provided by the Apache Felix Event Admin Service.
- **Logging System** - The logging bridge writes OSGi Log Service LogEntries to the server log file.
- **Blueprint Container Support** - The OSGi Blueprint Container allows bundles to contain standard blueprint descriptors, which can be used to create or consume OSGi services. Blueprint components consume OSGi services via injection. The idea is to use a plain POJO programming model and let Blueprint do the wiring for you. There should be no need for OSGi API to "pollute" your application logic.

5. Developer Documentation

5.1. Service Provider Interface

The JBoss OSGi Service Provider Interface (SPI) is the integration point for:

- Supported OSGi Frameworks
- Supported Target Containers
- Administration, Provisioning and Management
- Various Provided Services

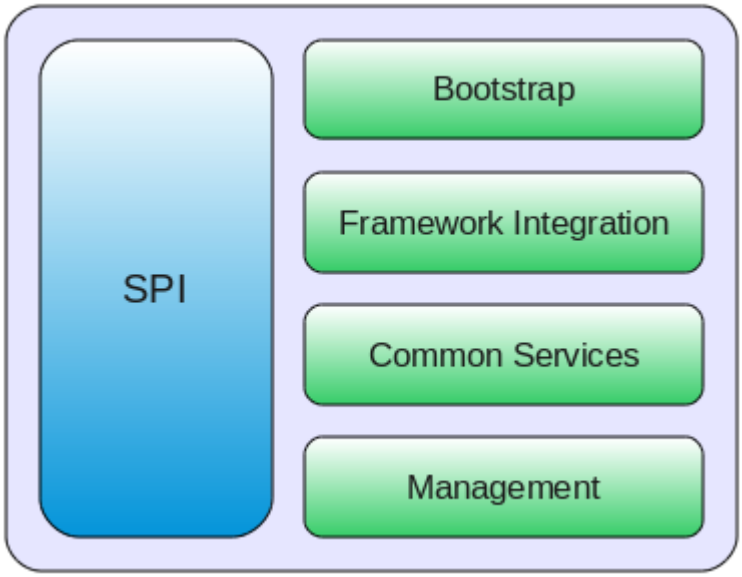

Figure 16.

The latest version of the JBoss OSGi API.

- org.jboss.osgi.spi - Common classes and interfaces.
- org.jboss.osgi.spi.capability - Capabilities that can be installed in the OSGi framework.
- org.jboss.osgi.spi.framework - Framework integration and bootstrap.
- org.jboss.osgi.spi.util - A collection of SPI provided utilities.
- org.jboss.osgi.testing - Test support classes and interfaces.

5.2. Bootstrapping JBoss OSGi

OSGiBootstrap provides an OSGiFramework through a OSGiBootstrapProvider.

A OSGiBootstrapProvider is discovered in two stages

1. Read the bootstrap provider class name from a system property
2. Read the bootstrap provider class name from a resource file

In both cases the key is the fully qualified name of the `org.jboss.osgi.spi.framework.OSGiBootstrapProvider` interface.

The following code shows how to get the default `OSGiFramework` from the `OSGiBootstrap-Provider`.

```
OSGiBootstrapProvider bootProvider = OSGiBootstrap.getBootstrapProvider();
    OSGiFramework framework = bootProvider.getFramework();
    Bundle bundle = framework.getSystemBundle();
```

The `OSGiBootstrapProvider` can also be configured explicitly. The `OSGiFramework` is a named object from the configuration.

```
OSGiBootstrapProvider bootProvider = OSGiBootstrap.getBootstrapProvider();
    bootProvider.configure(configURL);

    OSGiFramework framework = bootProvider.getFramework();
    Bundle bundle = framework.getSystemBundle();
```

The JBoss OSGi SPI comes with a default bootstrap provider:

- PropertiesBootstrapProvider

OSGiBootstrapProvider implementations that read their configurtation from some other source are possible, but currently not part of the JBoss OSGi SPI.

5.3. Management View

JBoss OSGi provides standard org.osgi.jmx management. Additional to that we provide an MBeanServer service and a few other extensions through the org.jboss.osgi.jmx API

Accessing the Management Objects

If you work with the JBoss OSGi runtime abstraction you get access to these managed objects through OSGiRuntime.

5.4. Writing Test Cases

Note: see also the Section 6, “Arquillian Test Framework” section for documentation on writing Arquillian-based test cases.

JBoss OSGi comes with JUnit test support as part of the SPI provided org.jboss.osgi.testing package. There are two distinct test scenarios that we support:

- Embedded OSGi Framework
- Remote OSGi Framework

A test case that takes advantage of the OSGi runtime abstraction transparently handles the various remote scenarios.

5.4.1. Simple Framework Test Case

The most basic form of OSGi testing can be done with an OSGiFrameworkTest. This would boostrap the framework in the @BeforeClass scope and make the framework instance available through getFramework(). Due to classloading restrictions, you cannot however not share non-primitive types between the test and the framework.

```
public class SimpleFrameworkTestCase extends OSGiFrameworkTest
{
   @Test
   public void testSimpleBundle() throws Exception
   {
      // Get the bundle location
      URL url = getTestArchiveURL("example-simple.jar");

      // Install the Bundle
      BundleContext sysContext = getFramework().getBundleContext();
      Bundle bundle = sysContext.installBundle(url.toExternalForm());
      assertBundleState(Bundle.INSTALLED, bundle.getState());

      // Start the bundle
      bundle.start();
      assertBundleState(Bundle.ACTIVE, bundle.getState());

      // Stop the bundle
      bundle.stop();
      assertBundleState(Bundle.RESOLVED, bundle.getState());

      // Uninstall the bundle
      bundle.uninstall();
      assertBundleState(Bundle.UNINSTALLED, bundle.getState());
   }
}
```

These tests always work with an embedded OSGi framework. You can however use the -Dframework property to run the test against a different framework implemenation (i.e. Apache Felix).

5.4.2. Lifecycle Interceptors

A common pattern in OSGi is that a bundle contains some piece of meta data that gets processed by some other infrastructure bundle that is installed in the OSGi Framework. In such cases the well known Extender Pattern is often being used. JBoss OSGi offeres a differnet approach to address this problem which is covered by the Extender Pattern vs. Lifecycle Interceptor post in the JBoss OSGi Diary.

Extending an OSGi Bundle

1. Extender registers itself as BundleListener
2. Bundle gets installed/started# Framework fires a BundleEvent
3. Extender picks up the BundleEvent (e.g. STARTING)
4. Extender reads metadata from the Bundle and does its work

There is no extender specific API. It is a pattern rather than a piece of functionality provided by the Framework. Typical examples of extenders are the Blueprint or Web Application Extender.

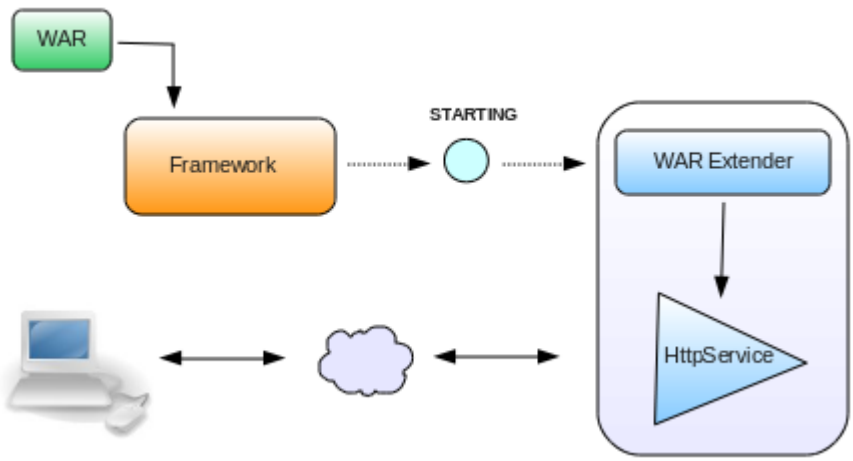

Figure 17.

Client code that installs, starts and uses the registered endpoint could look like this.

```
// Install and start the Web Application bundle
Bundle bundle = context.installBundle("mywebapp.war");
bundle.start();

// Access the Web Application
String response = getHttpResponse("http://localhost:8090/mywebapp/foo");
assertEquals("ok", response);
```

This seemingly trivial code snippet has a number of issues that are probably worth looking into in more detail

- The WAR might have missing or invalid web metadata (i.e. an invalid WEB-INF/web.xml descriptor)
- The WAR Extender might not be present in the system
- There might be multiple WAR Extenders present in the system
- Code assumes that the endpoint is available on return of bundle.start()

Most Blueprint or WebApp bundles are not useful if their Blueprint/Web metadata is not processed. Even if they are processed but in the "wrong" order a user might see unexpected results (i.e. the webapp processes the first request before the underlying Blueprint app is wired together).

As a consequence the extender pattern is useful in some cases but not all. It is mainly useful if a bundle can optionally be extended in the true sense of the word.

Intercepting the Bundle Lifecycle

If the use case requires the notion of "interceptor" the extender pattern is less useful. The use case might be such that you would want to intercept the bundle lifecycle at various phases to do mandatory metadata processing.

An interceptor could be used for annotation processing, byte code weaving, and other non-optional/optional metadata processing steps. Typically interceptors have a relative order, can communicate with each other, veto progress, etc.

Lets look at how multiple interceptors can be used to create Web metadata and publish endpoints on the HttpService based on that metadata.

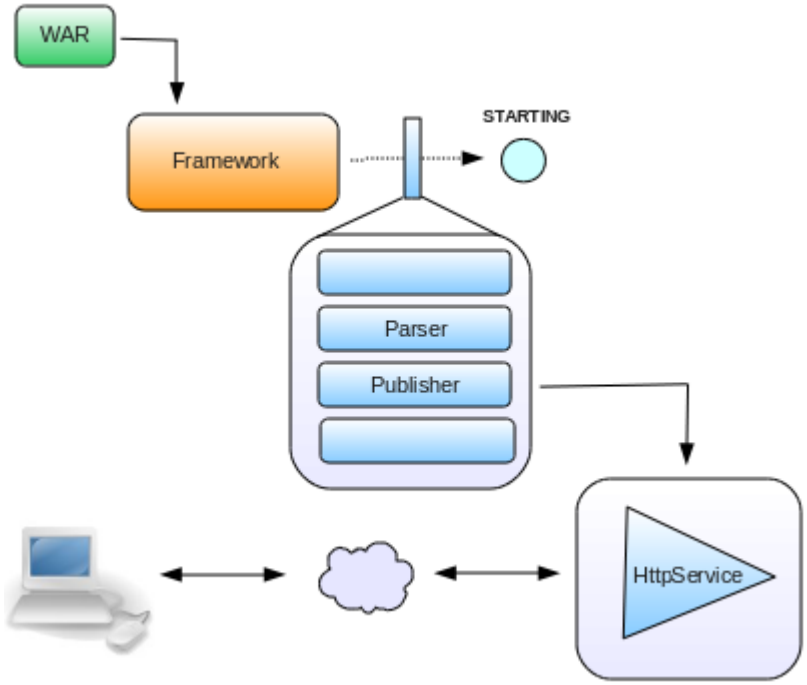

Figure 18.

Here is how it works

1. The Wep Application processor registers two LifecycleInterceptors with the LifecycleInterceptorService
2. The Parser interceptor declares no required input and WebApp metadata as produced output
3. The Publisher interceptor declares WebApp metadata as required input
4. The LifecycleInterceptorService reorders all registered interceptors according to their input/output requirements and relative order
5. The WAR Bundle gets installed and started
6. The Framework calls the LifecycleInterceptorService prior to the actual state change
7. The LifecycleInterceptorService calls each interceptor in the chain
8. The Parser interceptor processes WEB-INF/web.xml in the invoke(int state, InvocationContext context) method and attaches WebApp metadata to the InvocationContext
9. The Publisher interceptor is only called when the InvocationContext has WebApp metadata attached. If so, it publishes the endpoint from the WebApp metadata
10. If no interceptor throws an Exception the Framework changes the Bundle state and fires the BundleEvent.

Client code is identical to above.

```
// Install and start the Web Application bundle
Bundle bundle = context.installBundle("mywebapp.war");
bundle.start();

// Access the Web Application
String response = getHttpResponse("http://localhost:8090/mywebapp/foo");
assertEquals("ok", response);
```

The behaviour of that code however, is not only different but also provides a more natural user experience.

- Bundle.start() fails if WEB-INF/web.xml is invalid
- An interceptor could fail if web.xml is not present
- The Publisher interceptor could fail if the HttpService is not present
- Multiple Parser interceptors would work mutually exclusiv on the presents of attached WebApp metadata
- The endpoint is guaranteed to be available when Bundle.start() returns

The general idea is that each interceptor takes care of a particular aspect of processing during state changes. In the example above WebApp metadata might get provided by an interceptor that scans annotations or by another one that generates the metadata in memory. The Publisher interceptor would not know nor care who attached the WebApp metadata object, its task is to consume the WebApp metadata and publish endpoints from it.

For details on howto provide and register liefecycle interceptors have a look at the Lifecycle Interceptor Example.

6. Arquillian Test Framework

6.1. Overview

Arquillian is a Test Framework that allows you to run plain JUnit4 test cases from within an OSGi Framework. That the test is actually executed in the the OSGi Framework is transparent to your test case. There is no requirement to extend a specific base class. Your OSGi tests execute along side with all your other (non OSGi specific) test cases in Maven, Ant, or Eclipse.

Some time ago I was looking for ways to test bundles that are deployed to a remote instance of the JBoss OSGi Runtime. I wanted the solution to also work with an OSGi Framework that is bootstrapped from within a JUnit test case.

The basic problem is of course that you cannot access the artefacts that you deploy in a bundle directly from your test case, because they are loaded from different classloaders.

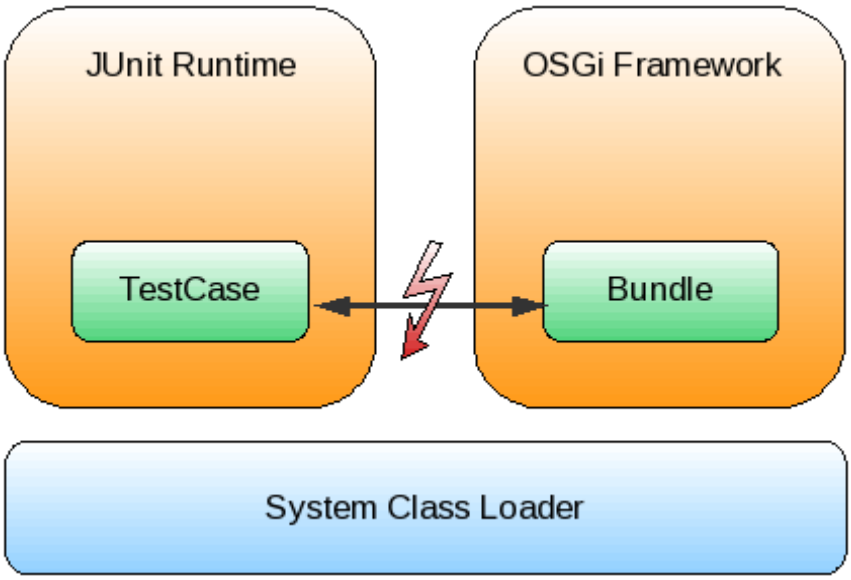

Figure 19.

For this release, we extended the Arquillian Test Framework to provide support for these requirements.

- Test cases SHOULD be plain JUnit4 POJOs
- There SHOULD be no requirement to extend a specific test base class
- There MUST be no requirement on a specific test runner (i.e. MUST run with Maven)
- There SHOULD be a minimum test framework leakage into the test case
- The test framework MUST support embedded and remote OSGi runtimes with no change required to the test
- The same test case MUST be executable from outside as well as from within the OSGi Framework
- There SHOULD be a pluggable communication layer from the test runner to the OSGi Framework
- The test framework MUST NOT depend on OSGi Framework specific features
- There MUST be no automated creation of test bundles required by the test framework

6.2. Configuration

In the target OSGi Framework, you need to have the **arquillian-osgi-bundle.jar** up and running. For remote testing you also need **jboss-osgi-jmx.jar** because Arquillian uses the a standard JSR-160 to communicate between the test client and the remote OSGi Framework.

See jboss-osgi-jmx on how the JMX protocol can be configured.

6.3. Writing Arquillian Tests

In an Arquillian test you

- need to use the **@RunWith(Arquillian.class)** test runner
- may have a **@Deployment** method that generates the test bundle
- may have **@Inject BundleContext** to get the system BundleContext injected
- may have **@Inject Bundle** to get the test bundle injected

```
@RunWith(Arquillian.class)
public class SimpleArquillianTestCase
{
   @Inject
   public Bundle bundle;

   @Deployment
   public static JavaArchive createdeployment()
   {
      final JavaArchive archive = ShrinkWrap.create(JavaArchive.class, "example-
arquillian");
      archive.addClasses(SimpleActivator.class, SimpleService.class);
      archive.setManifest(new Asset()
      {
         public InputStream openStream()
         {
            OSGiManifestBuilder builder = OSGiManifestBuilder.newInstance();
            builder.addBundleSymbolicName(archive.getName());
            builder.addBundleManifestVersion(2);
            builder.addBundleActivator(SimpleActivator.class.getName());
            return builder.openStream();
         }
      });
```

```
      return archive;
   }

   @Test
   public void testBundleInjection() throws Exception
   {
      // Assert that the bundle is injected
      assertNotNull("Bundle injected", bundle);

      // Assert that the bundle is in state RESOLVED
      // Note when the test bundle contains the test case it
      // must be resolved already when this test method is called
      assertEquals("Bundle RESOLVED", Bundle.RESOLVED, bundle.getState());

      // Start the bundle
      bundle.start();
      assertEquals("Bundle ACTIVE", Bundle.ACTIVE, bundle.getState());

      // Get the service reference
      BundleContext context = bundle.getBundleContext();
      ServiceReference sref = context.getServiceReference(SimpleService.class.getName());
      assertNotNull("ServiceReference not null", sref);

      // Get the service for the reference
      SimpleService service = (SimpleService)context.getService(sref);
      assertNotNull("Service not null", service);

      // Invoke the service
      int sum = service.sum(1, 2, 3);
      assertEquals(6, sum);

      // Stop the bundle
      bundle.stop();
      assertEquals("Bundle RESOLVED", Bundle.RESOLVED, bundle.getState());
   }
}
```

7. Provided Bundles and Services

7.1. Blueprint Container Service

The JBoss OSGi **jbosgi-blueprint.jar** bundle provides together with **org.apache.aries.blueprint.jar** access to the Blueprint extender service.

The Blueprint Container service allows bundles to contain standard blueprint descriptors, which can be used for component wiring and injection of blueprint components. The idea is to use a plain POJO programming model and let Blueprint do the wiring for you. There should be no need for OSGi API to "pollute" your application logic.

The Blueprint API is divided into the **Blueprint Container** and **Blueprint Reflection** packages.

- org.osgi.service.blueprint.container
- org.osgi.service.blueprint.reflect

7.2. HttpService

The **pax-web-jetty-bundle.jar** bundle from the OPS4J Pax Web project provides access to the HttpService.

An example of how a bundle uses the HttpService to register servlet and resources is given in HttpService Example.

The HttpService is configured with these properties.

Key	Value	Description
org.osgi.service.http.port	8090	The property that sets the port the HttpService binds to

The service is registered with the Framework under the name

- org.osgi.service.http.HttpService

7.3. JMX Service

The JBoss OSGi **jboss-osgi-jmx.jar** bundle activator discovers and registers the MBeanServer with the framework. By default, it also sets up a remote connector at:

```
service:jmx:rmi://localhost:1198/jndi/rmi://localhost:1090/osgi-jmx-connector
```

The JMX Service is configured with these properties.

Key	Value	Description
org.jboss.osgi.jmx.host	localhost	The property that sets the host that the JMXConnector binds to
org.jboss.osgi.jmx.rmi.port	1198	The property that sets the port that the JMXConnector binds to
org.jboss.osgi.jmx.rmi.registry.port	1090	The property that sets the port that the RMI Registry binds to

Here is the complete list of services that this bundle provides

- javax.management.MBeanServer
- org.jboss.osgi.jmx.FrameworkMBeanExt
- org.jboss.osgi.jmx.BundleStateMBeanExt
- org.jboss.osgi.jmx.PackageStateMBeanExt
- org.jboss.osgi.jmx.ServiceStateMBeanExt

7.4. XML Parser Services

The JBoss OSGi **jboss-osgi-apache-xerces.jar** bundle provides services for DOM and SAX parsing.

The services are registered with the Framework under the name

- javax.xml.parsers.SAXParserFactory
- javax.xml.parsers.DocumentBuilderFactory

Please see XMLParserActivator for details.

8. Provided Examples

Note: these examples are provided as part of the standalone distribution of JBoss OSGi and are not included in the JBossAS distribution.

8.1. Build and Run the Examples

JBoss OSGi comes with a number of examples that demonstrate supported functionality and show best practices. All examples are part of the binary distribution and tightly integrated in our Maven Build Process and Hudson QA Environment.

The examples can be either run against an embedded OSGi framework or against the remote OSGi Runtime. Here is how you build and run the against the embedded framework.

```
[tdiesler@tddell example]$ mvn test

-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running org.jboss.test.osgi.example.webapp.WebAppInterceptorTestCase
Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 14.417 sec
...

Tests run: 25, Failures: 0, Errors: 0, Skipped: 0

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1 minute 31 seconds
[INFO] Finished at: Tue Dec 08 11:15:08 CET 2009
[INFO] Final Memory: 35M/139M
[INFO] ------------------------------------------------------------------------
```

To run the examples against a remote OSGi Runtime, you need to provide the target container that the runtime should connect to. This can be done with the **target.container** system property.

```
mvn -Dtarget.container=runtime test
```

8.2. Event Admin Example

The **example-event.jar** bundle uses the EventAdmin service to send/receive events.

```
public void testEventHandler() throws Exception
{
  TestEventHandler eventHandler = new TestEventHandler();

  // Register the EventHandler
  Dictionary param = new Hashtable();
  param.put(EventConstants.EVENT_TOPIC, new String[Introduction] { TOPIC });
  context.registerService(EventHandler.class.getName(), eventHandler, param);

  // Send event through the the EventAdmin
  ServiceReference sref = context.getServiceReference(EventAdmin.class.getName());
  EventAdmin eventAdmin = (EventAdmin)context.getService(sref);
  eventAdmin.sendEvent(new Event(TOPIC, null));

  // Verify received event
  assertEquals("Event received", 1, eventHandler.received.size());
  assertEquals(TOPIC, eventHandler.received.get(0).getTopic());
}
```

8.3. Blueprint Container

The **example-blueprint.jar** bundle contains a number of components that are wired together and registered as OSGi service through the Blueprint Container Service.

The example uses this simple blueprint descriptor

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0" ...>

  <bean id="beanA" class="org.jboss.test.osgi.example.blueprint.bundle.BeanA">
    <property name="mbeanServer" ref="mbeanService"/>
  </bean>

  <service id="serviceA" ref="beanA"
 interface="org.jboss.test.osgi.example.blueprint.bundle.ServiceA">
  </service>

  <service id="serviceB"
 interface="org.jboss.test.osgi.example.blueprint.bundle.ServiceB">
    <bean class="org.jboss.test.osgi.example.blueprint.bundle.BeanB">
       <property name="beanA" ref="beanA"/>
    </bean>
  </service>

  <reference id="mbeanService" interface="javax.management.MBeanServer"/>

</blueprint>
```

The Blueprint Container registers two services **ServiceA** and **ServiceB**. ServiceA is backed up by **BeanA**, ServiceB is backed up by the anonymous **BeanB**. BeanA is injected into BeanB and the **MBeanServer** gets injected into BeanA. Both beans are plain POJOs. There is **no BundleActivator** neccessary to register the services.

The example test verifies the correct wiring like this

```
@Test
public void testServiceA() throws Exception
{
  ServiceReference sref = context.getServiceReference(ServiceA.class.getName());
  assertNotNull("ServiceA not null", sref);

  ServiceA service = (ServiceA)context.getService(sref);
  MBeanServer mbeanServer = service.getMbeanServer();
  assertNotNull("MBeanServer not null", mbeanServer);
}
```

```
@Test
public void testServiceB() throws Exception
{
  ServiceReference sref = context.getServiceReference(ServiceB.class.getName());
  assertNotNull("ServiceB not null", sref);

  ServiceB service = (ServiceB)context.getService(sref);
  BeanA beanA = service.getBeanA();
  assertNotNull("BeanA not null", beanA);
}
```

8.4. HttpService

The **example-http.jar** bundle contains a Service that registeres a servlet and a resource with the HttpService.

```
ServiceTracker tracker = new ServiceTracker(context, HttpService.class.getName(), null);
tracker.open();

HttpService httpService = (HttpService)tracker.getService();
if (httpService == null)
   throw new IllegalStateException("HttpService not registered");

Properties initParams = new Properties();
initParams.setProperty("initProp", "SomeValue");
httpService.registerServlet("/servlet", new EndpointServlet(context), initParams, null);
httpService.registerResources("/file", "/res", null);
```

The test then verifies that the registered servlet context and the registered resource can be accessed.

8.5. JMX Service

8.5.1. MBeanServer Service

The **example-jmx.jar** bundle tracks the MBeanServer service and registers a pojo with JMX. It then verifies the JMX access.

```
public class FooServiceActivator implements BundleActivator
{
   public void start(BundleContext context)
   {
      ServiceTracker tracker = new ServiceTracker(context, MBeanServer.class.getName(),
 null)
      {
         public Object addingService(ServiceReference reference)
         {
            MBeanServer mbeanServer = (MBeanServer)super.addingService(reference);
            registerMBean(mbeanServer);
            return mbeanServer;
         }

         @Override
         public void removedService(ServiceReference reference, Object service)
         {
            unregisterMBean((MBeanServer)service);
            super.removedService(reference, service);
         }
      };
      tracker.open();
   }

   public void stop(BundleContext context)
   {
      ServiceReference sref = context.getServiceReference(MBeanServer.class.getName());
      if (sref != null)
      {
         MBeanServer mbeanServer = (MBeanServer)context.getService(sref);
         unregisterMBean(mbeanServer);
      }
   }
   ...
}
```

```
public void testMBeanAccess() throws Exception
{
  FooMBean foo = (FooMBean)MBeanProxy.get(FooMBean.class, MBEAN_NAME,
 runtime.getMBeanServer());
  assertEquals("hello", foo.echo("hello"));
}
```

Please note that access to the MBeanServer from the test case is part of the OSGiRuntime abstraction.

8.5.2. Bundle State control via BundleStateMBean

The **BundleStateTestCase** uses JMX to control the bundle state through the BundleStateMBean. The **testBundleStateMBean** lists the available bundles over JMX.

```
public void testBundleStateMBean() throws Exception
{
  BundleStateMBean bundleState = getRuntime().getBundleStateMBean();
  assertNotNull("BundleStateMBean not null", bundleState);

  TabularData bundleData = bundleState.listBundles();
  assertNotNull("TabularData not null", bundleData);
```

```
  assertFalse("TabularData not empty", bundleData.isEmpty());
}
```

The **testUpdateBundle** updates a bundle in the framework over JMX. It starts by installing a bundle via the FrameworkMBean. This bundle exports the package **org.jboss.test.osgi.example.jmx.bundle.update1**

```
public void testUpdateBundle() throws Exception
{
  FrameworkMBean fw = getRuntime().getFrameworkMBean();
  BundleStateMBean bs = getRuntime().getBundleStateMBean();

  // Install and start a bundle via JMX that exports a package
  URL bundleURL = getTestArchiveURL("example-jmx-update1.jar");
  long bundleId = fw.installBundle(bundleURL.toString());
  fw.startBundle(bundleId);

  // Obtain the exported packages through JMX
  assertEquals("[org.jboss.test.osgi.example.jmx.bundle.update1;0.0.0]",
        Arrays.toString(bs.getExportedPackages(bundleId)));
```

Subsequently the bundle is updated through **FrameworkMBean.updateBundledFromURL()** with a revision that exports the package the package **org.jboss.test.osgi.example.jmx.bundle.update2**

```
URL updatedURL = getTestArchiveURL("example-jmx-update2.jar");
  fw.updateBundleFromURL(bundleId, updatedURL.toString());
```

8.5.3. Start Level control via FrameworkMBean

The **StartLevelTestCase** uses JMX to control bundle and framework start levels. The beginning of the **testStartLevelMBean** uses JMX to set the initial bundle start level to 2 and then installs a new bundle in the framework through JMX.

Once installed, it finds the bundle ID of the newly installed bundle through JMX, by obtaining the **TabularData** from **listBundles()**. The test then tries to start the bundle, but this doesn't actually start the bundle yet as the framework start level is still at 1. Increasing the start level of the bundle to 5 should keep the bundle in the **INSTALLED** state. Finally the framework start level is increased to 10 which will bring the bundle in the **ACTIVE** state.

```
public void testStartLevelMBean() throws Exception
{
  FrameworkMBean fw = runtime.getFrameworkMBean();
  fw.setInitialBundleStartLevel(2);

  Assert.assertEquals(1, fw.getFrameworkStartLevel());
  OSGiBundle bundle = runtime.installBundle("any_bundle.jar");

  BundleStateMBean bs = runtime.getBundleStateMBean();
  TabularData td = bs.listBundles();

  long bundleId = -1;
  for (CompositeData row : (Collection<CompositeData>)td.values())
  {
     if (bundle.getSymbolicName().equals(row.get("SymbolicName")))
     {
        bundleId = Long.parseLong(row.get("Identifier").toString());
        break;
     }
  }
  assertTrue("Could not find test bundle through JMX", bundleId != -1);

  fw.startBundle(bundleId);

  assertEquals(2, bs.getStartLevel(bundleId));
  fw.setBundleStartLevel(bundleId, 5);
  assertEquals(5, bs.getStartLevel(bundleId));
  waitForBundleState("INSTALLED", bs, bundleId);
```

```
  fw.setFrameworkStartLevel(10);
  waitForBundleState("ACTIVE", bs, bundleId);
  ...
```

8.6. JNDI Service

The **example-jndi.jar** bundle gets the InitialContext service and registers a string with JNDI. It then verifies the JNDI access.

```
ServiceReference sref = context.getServiceReference(InitialContext.class.getName());
if (sref == null)
   throw new IllegalStateException("Cannot access the InitialContext");

InitialContext iniContext = (InitialContext)context.getService(sref);
iniCtx.createSubcontext("test").bind("Foo", new String("Bar"));
```

```
public void testJNDIAccess() throws Exception
{
  InitialContext iniCtx = runtime.getInitialContext();
  String lookup = (String)iniCtx.lookup("test/Foo");
  assertEquals("JNDI bound String expected", "Bar", lookup);

  // Uninstall should unbind the object
  bundle.uninstall();

  try
  {
     iniCtx.lookup("test/Foo");
     fail("NameNotFoundException expected");
  }
  catch (NameNotFoundException ex)
  {
     // expected
  }
}
```

Please note that access to the InitialContext from the test case is part of the OSGiRuntime abstraction.

8.7. JTA Service

The **example-jta.jar** bundle gets the javax.transaction.UserTransaction service and registers a transactional user object (i.e. one that implements Synchronization) with the javax.transaction.TransactionManager service. It then verifies that modifications on the user object are transactional.

```
Transactional txObj = new Transactional();

ServiceReference userTxRef =
 context.getServiceReference(UserTransaction.class.getName());
assertNotNull("UserTransaction service not null", userTxRef);

UserTransaction userTx = (UserTransaction)context.getService(userTxRef);
assertNotNull("UserTransaction not null", userTx);

userTx.begin();
try
{
   ServiceReference tmRef =
 context.getServiceReference(TransactionManager.class.getName());
   assertNotNull("TransactionManager service not null", tmRef);

   TransactionManager tm = (TransactionManager)context.getService(tmRef);
   assertNotNull("TransactionManager not null", tm);
```

```
   Transaction tx = tm.getTransaction();
   assertNotNull("Transaction not null", tx);

   tx.registerSynchronization(txObj);

   txObj.setMessage("Donate $1.000.000");
   assertNull("Uncommited message null", txObj.getMessage());

   userTx.commit();
}
catch (Exception e)
{
   userTx.setRollbackOnly();
}

assertEquals("Donate $1.000.000", txObj.getMessage());
```

```
class Transactional implements Synchronization
{
  public void afterCompletion(int status)
  {
     if (status == Status.STATUS_COMMITTED)
        message = volatileMessage;
  }

  ...
}
```

8.8. Lifecycle Interceptor

The interceptor example deployes a bundle that contains some metadata and an interceptor bundle that processes the metadata and registeres an http endpoint from it. The idea is that the bundle does not process its own metadata. Instead this work is delegated to some specialized metadata processor (i.e. the interceptor).

Each interceptor is itself registered as a service. This is the well known Whiteboard Pattern.

```
public class InterceptorActivator implements BundleActivator
{
   public void start(BundleContext context)
   {
      LifecycleInterceptor publisher = new PublisherInterceptor();
      LifecycleInterceptor parser = new ParserInterceptor();

      // Add the interceptors, the order of which is handles by the service
      context.registerService(LifecycleInterceptor.class.getName(), publisher, null);
      context.registerService(LifecycleInterceptor.class.getName(), parser, null);
   }
}
```

```
public class ParserInterceptor extends AbstractLifecycleInterceptor
{
   ParserInterceptor()
   {
      // Add the provided output
      addOutput(HttpMetadata.class);
   }

   public void invoke(int state, InvocationContext context)
   {
      // Do nothing if the metadata is already available
      HttpMetadata metadata = context.getAttachment(HttpMetadata.class);
      if (metadata != null)
         return;

      // Parse and create metadta on STARTING
      if (state == Bundle.STARTING)
      {
          VirtualFile root = context.getRoot();
```

```
            VirtualFile propsFile = root.getChild("/http-metadata.properties");
            if (propsFile != null)
            {
               log.info("Create and attach HttpMetadata");
               metadata = createHttpMetadata(propsFile);
               context.addAttachment(HttpMetadata.class, metadata);
            }
         }
      }
      ...
}
```

```
public class PublisherInterceptor extends AbstractLifecycleInterceptor
{
   PublisherInterceptor()
   {
      // Add the required input
      addInput(HttpMetadata.class);
   }

   public void invoke(int state, InvocationContext context)
   {
      // HttpMetadata is guaratied to be available because we registered
      // this type as required input
      HttpMetadata metadata = context.getAttachment(HttpMetadata.class);

      // Register HttpMetadata on STARTING
      if (state == Bundle.STARTING)
      {
         String servletName = metadata.getServletName();

         // Load the endpoint servlet from the bundle
         Bundle bundle = context.getBundle();
         Class servletClass = bundle.loadClass(servletName);
         HttpServlet servlet = (HttpServlet)servletClass.newInstance();

         // Register the servlet with the HttpService
         HttpService httpService = getHttpService(context, true);
         httpService.registerServlet("/servlet", servlet, null, null);
      }

      // Unregister the endpoint on STOPPING
      else if (state == Bundle.STOPPING)
      {
         log.info("Unpublish HttpMetadata: " + metadata);
         HttpService httpService = getHttpService(context, false);
         if (httpService != null)
            httpService.unregister("/servlet");
      }
   }
}
```

8.9. Web Application

The **example-webapp.war** archive is an OSGi Bundle and a Web Application Archive (WAR) at the same time. Similar to HTTP Service Example it registers a servlet and resources with the WebApp container. This is done through a standard web.xml descriptor.

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee" ... version="2.5">

  <display-name>WebApp Sample</display-name>

  <servlet>
    <servlet-name>servlet</servlet-name>
    <servlet-class>org.jboss.test.osgi.example.webapp.bundle.EndpointServlet</servlet-
class>
    <init-param>
      <param-name>initProp</param-name>
```

```
      <param-value>SomeValue</param-value>
    </init-param>
  </servlet>

  <servlet-mapping>
    <servlet-name>servlet</servlet-name>
    <url-pattern>/servlet</url-pattern>
  </servlet-mapping>

</web-app>
```

The associated OSGi manifest looks like this.

```
Manifest-Version: 1.0
Bundle-Name: example-webapp
Bundle-ManifestVersion: 2
Bundle-SymbolicName: example-webapp
Bundle-ClassPath: .,WEB-INF/classes
Import-Package:
 org.osgi.service.http,org.ops4j.pax.web.service,javax.servlet,javax.servlet.http
```

The test verifies that we can access the servlet and some resources.

```
@Test
public void testResourceAccess() throws Exception
{
  assertEquals("Hello from Resource", getHttpResponse("/message.txt"));
}

@Test
public void testServletAccess() throws Exception
{
  assertEquals("Hello from Servlet", getHttpResponse("/servlet?test=plain"));
}
```

8.10. XML Parser Service

The **example-xml-parser.jar** bundle gets a DocumentBuilderFactory/SAXParserFactory respectivly and unmarshalls an XML document using that parser.

```
ServiceReference sref =
 context.getServiceReference(DocumentBuilderFactory.class.getName());
if (sref == null)
   throw new IllegalStateException("DocumentBuilderFactory not available");

DocumentBuilderFactory factory = (DocumentBuilderFactory)context.getService(sref);
factory.setValidating(false);

DocumentBuilder domBuilder = factory.newDocumentBuilder();
URL resURL = context.getBundle().getResource("example-xml-parser.xml");
Document dom = domBuilder.parse(resURL.openStream());
assertNotNull("Document not null", dom);
```

```
ServiceReference sref = context.getServiceReference(SAXParserFactory.class.getName());
if (sref == null)
   throw new IllegalStateException("SAXParserFactory not available");

SAXParserFactory factory = (SAXParserFactory)context.getService(sref);
factory.setValidating(false);

SAXParser saxParser = factory.newSAXParser();
URL resURL = context.getBundle().getResource("example-xml-parser.xml");

SAXHandler saxHandler = new SAXHandler();
saxParser.parse(resURL.openStream(), saxHandler);
assertEquals("content", saxHandler.getContent());
```

9. References

9.1. Resources

- OSGi 4.2 Specifications
- OSGi 4.2 JavaDoc
- JBoss OSGi Project Page
- JBoss OSGi Wiki
- JBoss OSGi Diary
- Issue Tracking
- Source Repository
- User Forum
- Design Forum

9.2. Authors

- `<Thomas Diesler>`
- `<David Bosschaert>`

10. Getting Support

We offer free support through the JBoss OSGi User Forum.

Please note, that posts to this forum will be dealt with at the community's leisure.
If your business is such that you need to rely on qualified answers within a known time frame, this forum might not be your preferred support channel.

For professional support please go to JBoss Support Services.