JBossOSGi - User Guide

Version: 1.0.0.Beta8-SNAPSHOT

1. Introduction 1
1.1. What is OSGi 1
1.2. OSGi Framework Overview 2
1.3. OSGi Service Compendium 6
2. Getting Started 11
2.1. Download the Distribution 11
2.2. Running the Installer 11
2.3. Starting the Runtime 15
2.4. Provided Examples 16
2.5. Bundle Deployment 17
2.6. Managing installed Bundles 17
2.7. Hudson QA Environment 18
3. JBoss OSGi Runtime 23
3.1. Overview 23
3.2. Features 24
3.3. Runtime Profiles 25
4. Framework Integration 29
4.1. JBossMC Framework Integration 29
4.2. Apache Felix Integration 29
4.3. Equinox Integration 31
5. Developer Documentation 33
5.1. Service Provider Interface 33
5.2. Management View 35
5.3. Writing Test Cases 36
5.3.1. Simple Test Case 36
5.3.2. Simple Husky Test Case 38
5.4. Lifecycle Interceptors 39
6. Husky Test Framework 43
6.1. Overview 43
6.2. Architecture 44
6.3. Configuration 45
6.4. Writing Husky Tests 46
7. Provided Bundles and Services 49
7.1. Blueprint Container Service 49
7.2. HttpService 49
7.3. JAXB Service 49
7.4. JMX Service 50
7.5. JNDI Service 50
7.6. JTA Service 51
7.7. Microcontainer Service 51
7.8. ServiceLoader Interceptor 52
7.9. WebApp Extender 52
7.10. XML Parser Services 53
7.11. XML Binding Services 53

8. Provided Examples 55
8.1. Build and Run the Examples 55
8.2. Event Admin Example 56
8.3. Blueprint Container 57
8.4. HttpService 58
8.5. JAXB Service 59
8.6. JMX Service 59
8.7. JNDI Service 61
8.8. JTA Service 62
8.9. Lifecycle Interceptor 63
8.10. Microcontainer Service 66
8.11. Web Application 66
8.12. ServiceLoader Example 67
8.13. XML Parser Service 68
8.14. XML Unmarshaller Service 68
9. References 71
10. Getting Support 73

Introduction

1.1. What is OSGi

The *Open Services Gateway Initiative (OSGi)* [http://www2.osgi.org/Release4/HomePage], specifications define a standardized, component-oriented, computing environment for networked services that is the foundation of an enhanced service-oriented architecture.

The OSGi specification defines two things:

- A set of services that an OSGi container must implement
- A contract between the container and your application

Developing on the OSGi platform means first building your application using OSGi APIs, then deploying it in an OSGi container.

The *JBoss OSGi Project* [http://www.jboss.org/community/docs/DOC-13273] project has two distinct goals

1. Provide an integration platform for 3rd party OSGi Frameworks
2. Provide an OSGi compliant framework implementation based on the *JBoss Microcontainer* [http://www.jboss.org/jbossmc]

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 security and lifecycle, which I'll explain 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. The OSGi platform builds an exellent foundation for the next generation JBoss ESB for example.

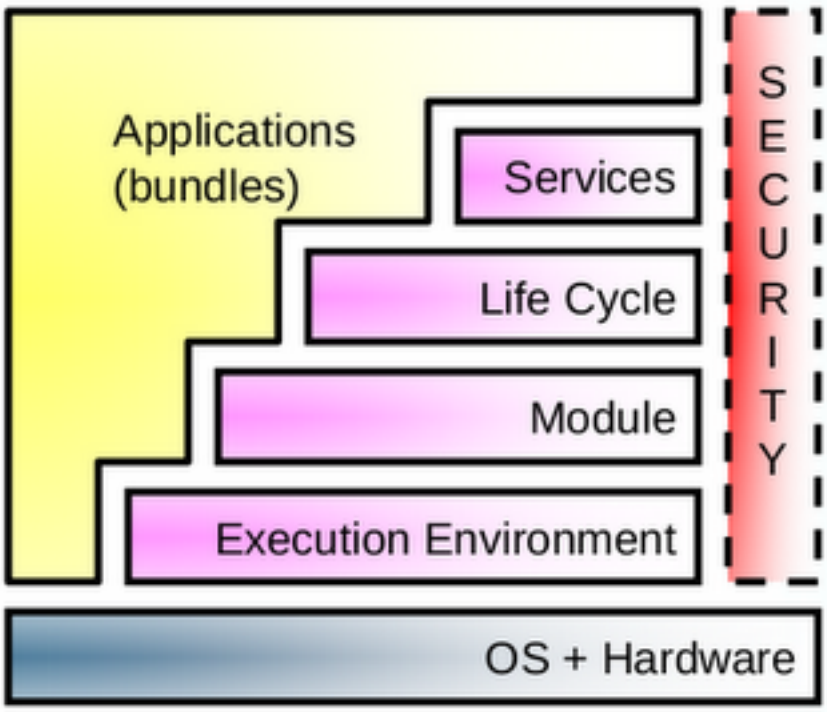
Should Java EE developers adopt the OSGi programming model?

Probably not. The OSGi runtime may be used internally by Java EE container providers to achieve the desired isolation and configuration flexibility that the container wishes to provide. At the application programming level, the Java EE model will continue to exist in its own right, whereas the OSGi model may provide the more suitable runtime environment for applications that require the modular isolation, security and lifecycle management that OSGi offers.

1.2. OSGi Framework Overview

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

- Security Layer
- Module Layer
- Life Cycle Layer
- Service Layer
- Actual Services



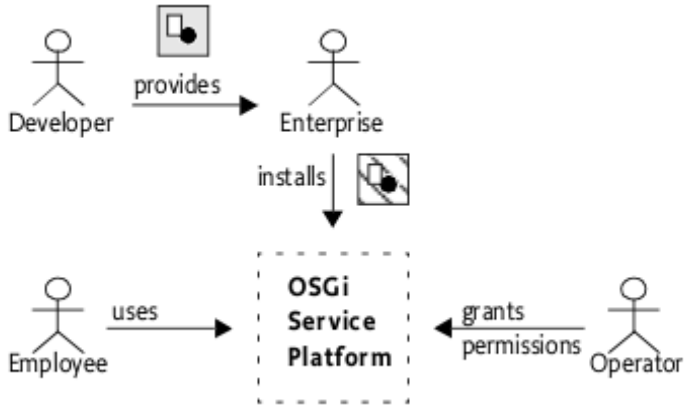
OSGi Security Layer

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.



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.

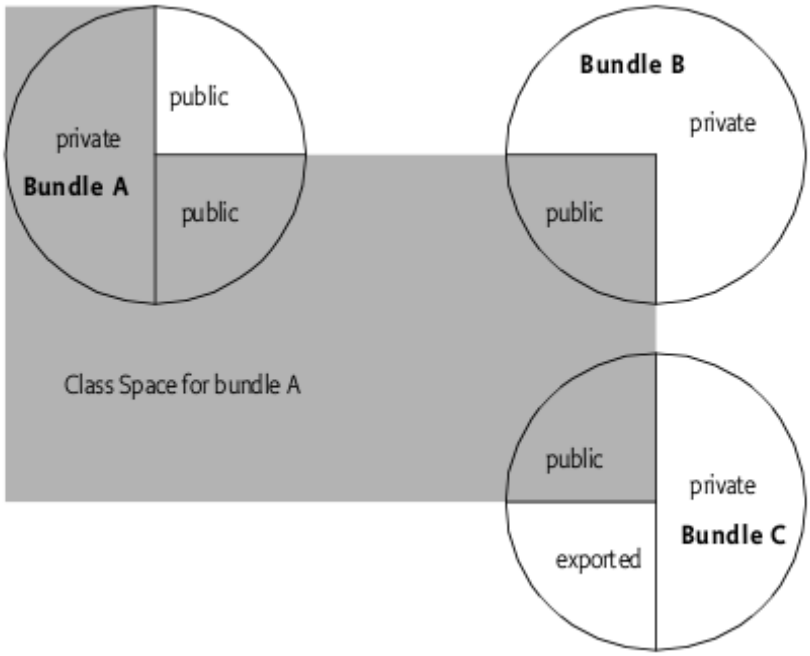
Once a **Bundle** is started, its functionality is provided and services are exposed to other bundles installed in the OSGi Service Platform. A bundle can carry descriptive information about itself in the manifest file that is contained in its JAR file. Here are a few important **Manifest Headers** defined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

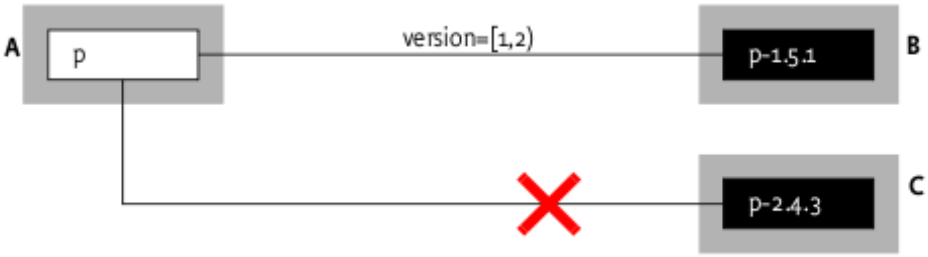
```
Import-Package: com.acme.foo;version="[1.23, 2)", com.acme.bar;version="[4.0, 5.0)"
```

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.



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
```



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
```

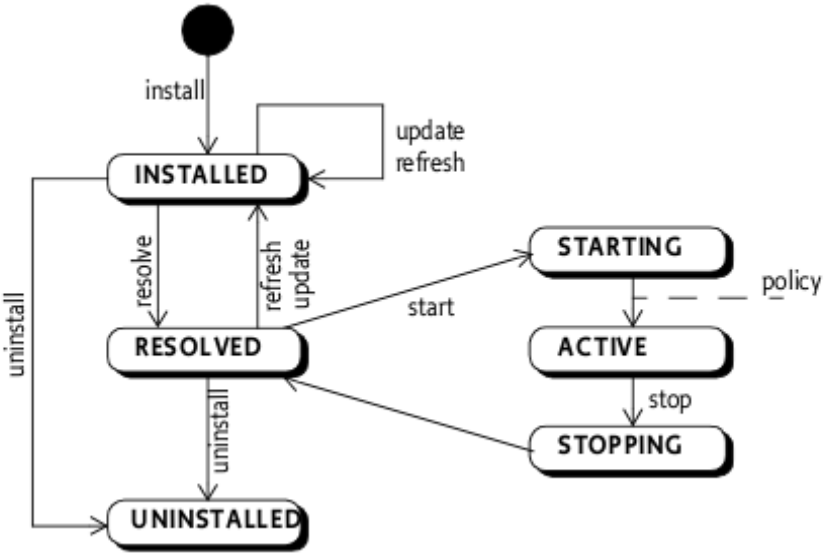
An exporter can limit the visibility of the classes in a package with the include and exclude directives on the export definition.

```
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:



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.

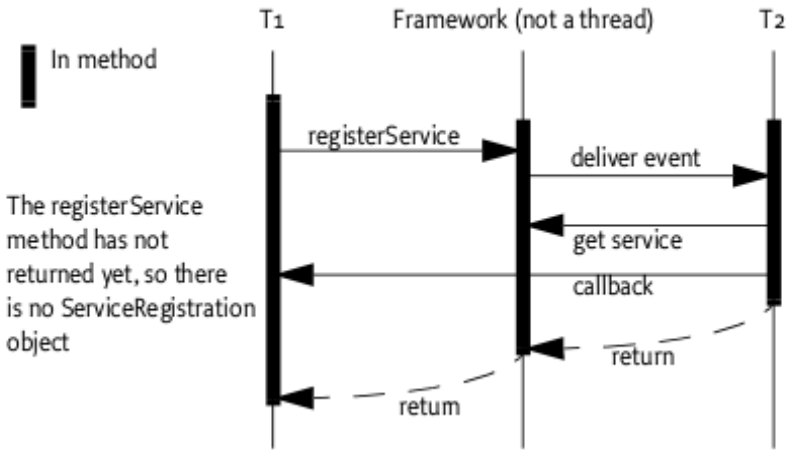
A Bundle Context object represents the execution context of a single bundle within the OSGi Service Platform, and acts as a proxy to the underlying Framework. A **Bundle Context** object is created by the Framework when a bundle is started. The bundle can use this private BundleContext object for the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



1.3. OSGi Service Compendium

The OSGi Service Compendium specifies a number of services that may be available in an OSGi runtime environment. Although the OSGi 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:

Log Service

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.

Http Service

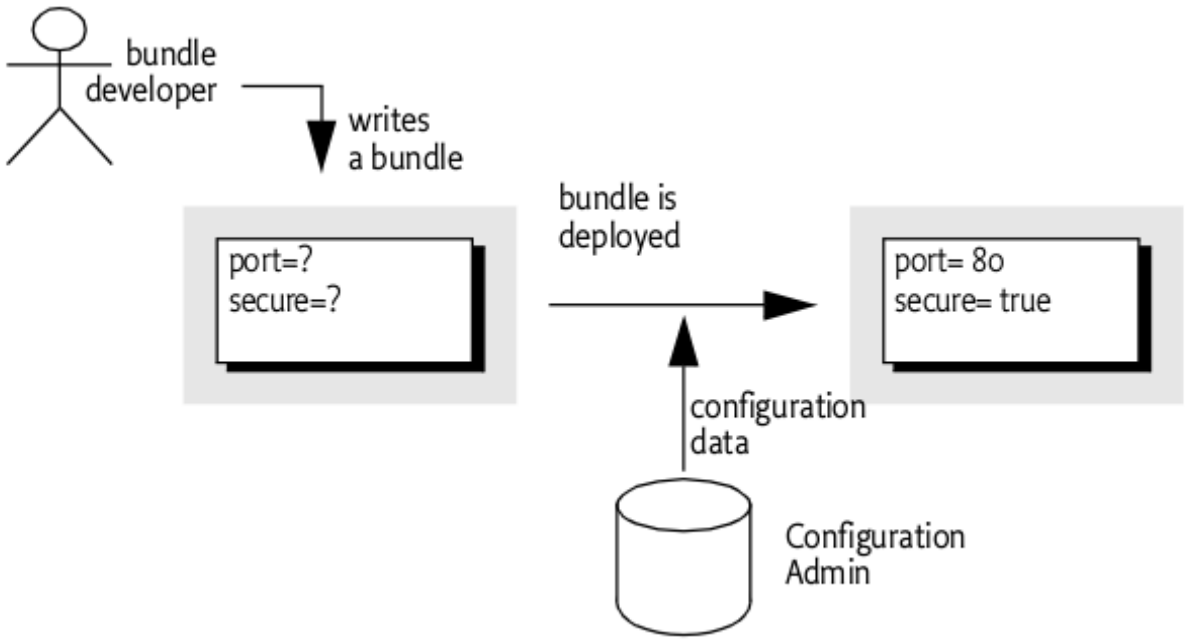
The Http Service supports two standard techniques for registering servlets and resources to develop communication and user interface solutions for standard technologies such as HTTP, HTML, XML, etc.

Device Access Specification

The Device Access specification supports the coordination of automatic detection and attachment of existing devices on an OSGi Service Platform, facilitates hot-plugging and -unplugging of new devices, and downloads and installs device drivers on demand.

Configuration Admin Service

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



Metatype Service

The Metatype Service specification defines interfaces that allow bundle developers to describe attribute types in a computer readable form using so-called metadata.

Preferences Service

The Preferences Service allows storage of data that is specific to a particular user.

User Admin Service

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.

Wire Admin Service

The Wire Admin Service is an administrative service that is used to control a wiring topology in the OSGi Service Platform. It is intended to be used by user interfaces or management programs that control the wiring of services in an OSGi Service Platform.

IO Connector Service

The IO Connector Service specification adopts the Java 2 Micro Edition (J2ME) javax.microedition.io packages as a basic communications infrastructure.

UPnP Device Service

The UPnP Device Service specifies how OSGi bundles can be developed that interoperate with UPnP (Universal Plug and Play) devices and UPnP control points.

Declarative Services Specification

The Declarative Services specification addresses some of the complications that arise when the OSGi service model is used for larger systems and wider deployments, such as: Startup Time, Memory Footprint, Complexity. The service component model uses a declarative model for publishing, finding and binding to OSGi services.

Event Admin Service

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

Deployment Admin Service

The Deployment Admin Service specification, standardizes the access to some of the responsibilities of the management agent: that is, the lifecycle management of interlinked resources on an OSGi Service Platform.

Auto Configuration Specification

The Auto Configuration Specification is to allow the configuration of bundles. These bundles can be embedded in Deployment Packages or bundles that are already present on the OSGi Service Platform.

Application Admin Service

The Application Admin Service is intended to simplify the management of an environment with many different types of applications that are simultaneously available.

DMT Admin Service

The DMT Admin Service specification defines an API for managing a device using concepts from the OMA DM specifications.

Monitor Admin Service

The Monitor Admin Service specification outlines how a bundle can publish Status Variables and how administrative bundles can discover Status Variables as well as read and reset their values.

Foreign Application Access Specification

The Foreign Application Access specification is to enable foreign application models like MIDP, Xlets, Applets, other Java application models to participate in the OSGi service oriented architecture.

Service Tracker Specification

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

XML Parser Service Specification

The XML Parser Service specification addresses how the classes defined in JAXP can be used in an OSGi Service Platform.

Position Specification

The Position Specification provides bundle developers with a consistent way of handling geographic positions in OSGi applications.

Measurement and State Specification

The Measurement and State Specification provides a consistent way of handling a diverse range of measurements for bundle developers.

Execution Environment Specification

This Execution Environment Specification defines different execution environments for OSGi Server Platform Servers.

Getting Started

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

2.1. Download the Distribution

JBoss OSGi is distributed as an *IzPack* [http://izpack.org] installer archive. The installer is available from the JBoss OSGi *download area* [http://sourceforge.net/projects/jboss/files/JBossOSGi].

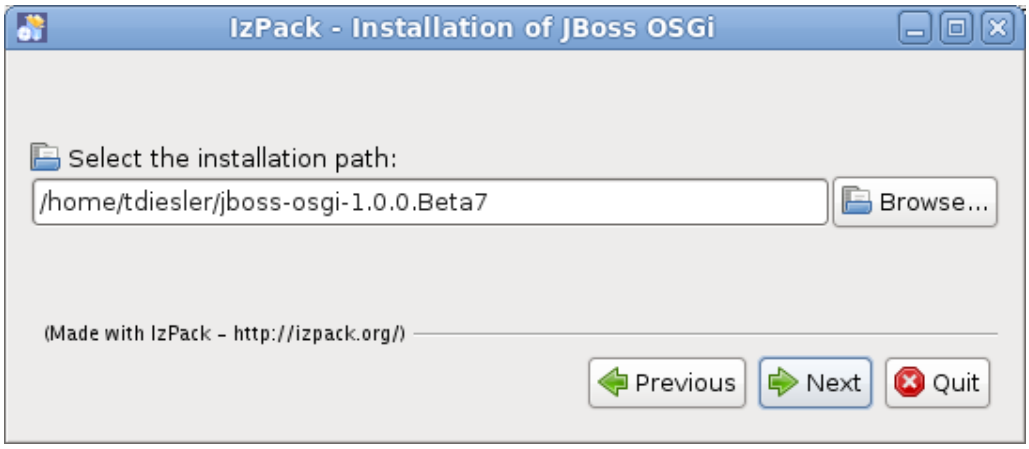
2.2. Running the Installer

To run the installer execute the following command:

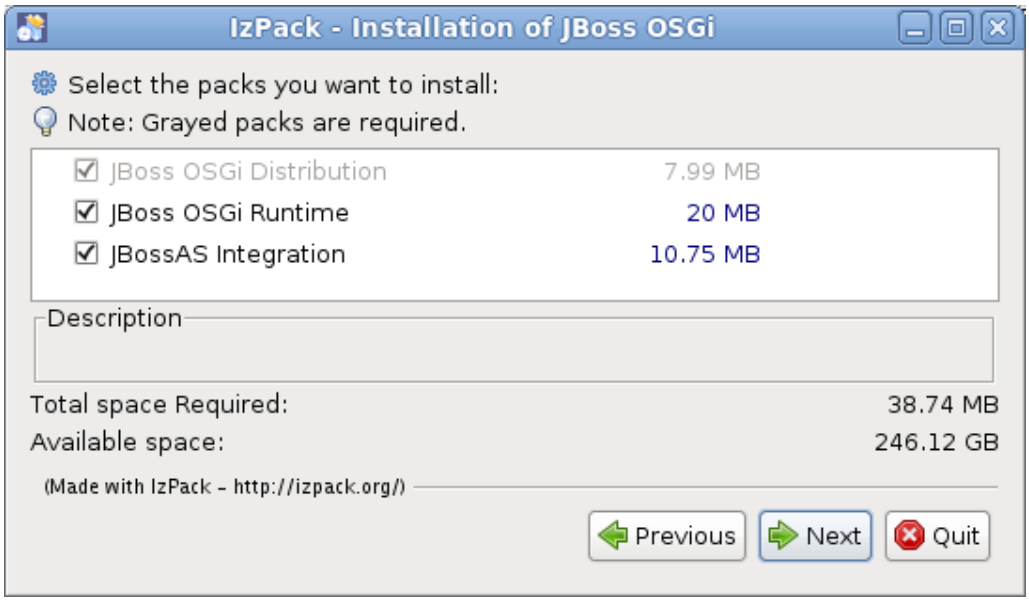
```
java -jar jboss-osgi-installer-1.0.0.Beta7.jar
```

The installer first shows a welcome screen

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.



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



- **JBoss OSGi Distribution** - Documentation, Binary Artifacts and Sources
- **JBoss OSGi Runtime** - Standalone JBoss OSGi Runtime
- **JBossAS Integration** - Integration with an existing JBossAS instance

Next, you will be presented with the choice of supported OSGi Frameworks.

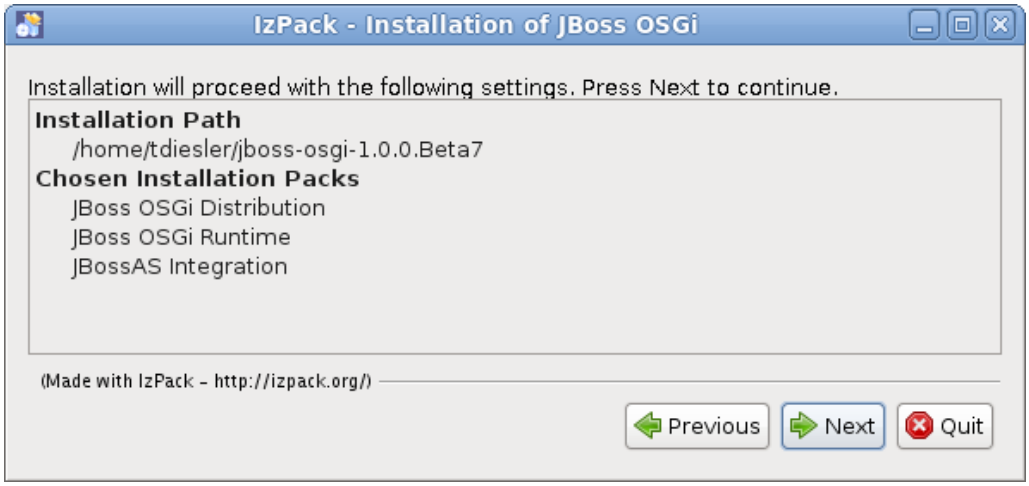
In case you have selected 'JBossAS Integration', you will be presented with the choice of supported target containers.

You will then have to point the installer to your existing *JBossAS* [http://http://jboss.org/jbossas] installation.

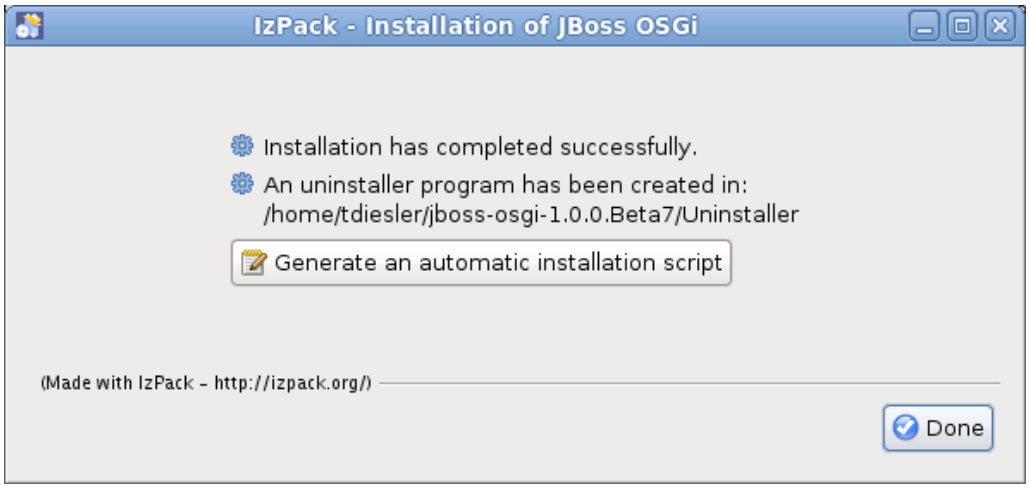
Supported JBossAS versions

The Microcontainer based OSGi Framework is only supported on JBoss-6.0.0 and above.

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



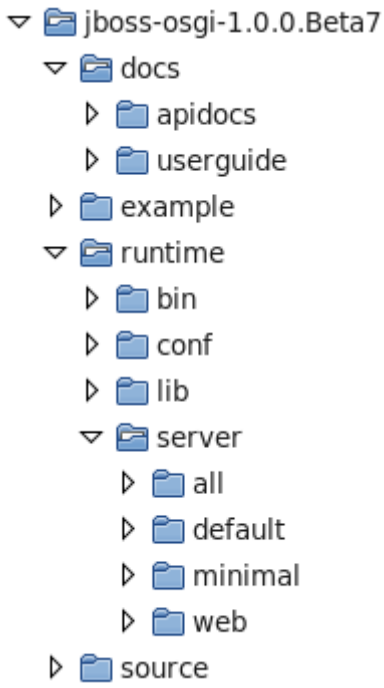
The installer reports its installation progress and finally displays a confirmation screen. You can now optionally generate an "automatic installation script" that you can use when you want to repeat what you have just done without user interaction.



2.3. 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.

The layout of the JBoss OSGi Runtime after installation is similar to what you know from *JBossAS* [http://http://jboss.org/jbossas].



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.Beta7/runtime

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

```
  JAVA_OPTS: ...

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

13:10:35,143 INFO  [OSGiBundleManager] JBossOSGi Core Framework - 1.0.0.Alpha4
...
13:10:36,405 INFO  Start DeploymentScanner: [scandir=.../server/default/
deploy,interval=2000ms]
13:10:36,416 INFO  Bundle STARTED: Bundle{system.bundle-0.0.0}
13:10:36,442 INFO  JBossOSGi Runtime booted in 1.297sec
13:10:36,742 INFO  Bundle INSTALLED: Bundle{jboss-osgi-jndi-1.0.2}
13:10:36,967 INFO  Bundle INSTALLED: Bundle{jboss-osgi-common-core-2.2.13.GA}
13:10:36,994 INFO  Bundle INSTALLED: Bundle{jboss-osgi-jmx-1.0.3}
13:10:37,018 INFO  Bundle INSTALLED: Bundle{org.apache.felix.eventadmin-1.0.0}
13:10:37,167 INFO  Bundle STARTED: Bundle{jboss-osgi-common-core-2.2.13.GA}
13:10:37,297 INFO  Bundle STARTED: Bundle{jboss-osgi-jmx-1.0.3}
13:10:37,829 INFO  Bundle STARTED: Bundle{jboss-osgi-jndi-1.0.2}
13:10:38,070 INFO  Bundle STARTED: Bundle{org.apache.felix.eventadmin-1.0.0}
...
13:10:38,334 INFO  JBossOSGi Runtime started in 1.939sec
```

2.4. Provided Examples

JBoss OSGi comes with a number of examples that you can build and deploy. Each example deployment is verified by an accompaning 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
- **microcontainer** - JBossMC/OSGi integration examples
- **serviceloader** - Autoregister META-INF/services
- **simple** - Simple OSGi examples (start here)

- **webapp** - WebApplication (WAR) examples
- **xml binding** - JBoss XML Binding examples
- **xml jaxb** - JAXB examples
- **xml parser** - SAX/DOM parser examples

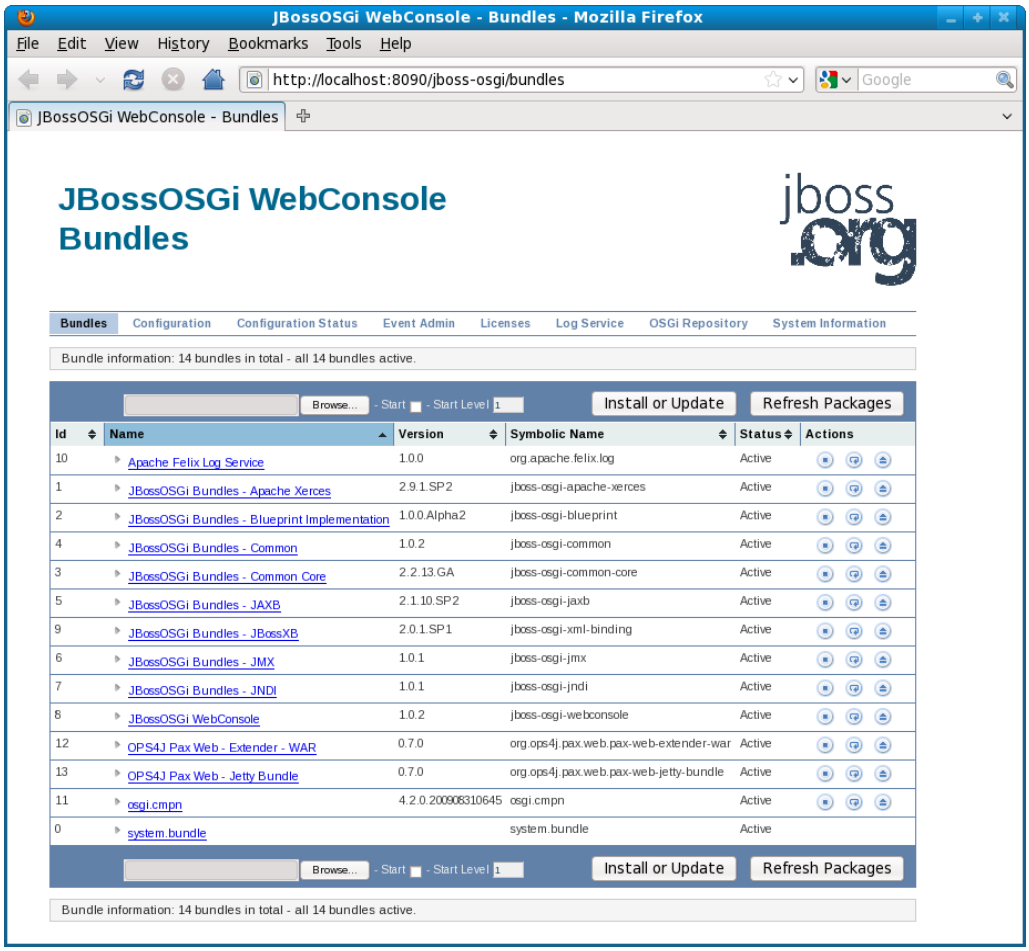
2.5. Bundle Deployment

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
```

2.6. Managing installed Bundles

JBoss OSGi comes with a simple Web Console, which is currently based on the *Apache Felix Web Console* [http://felix.apache.org/site/apache-felix-web-console.html] 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*.



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

2.7. Hudson QA Environment

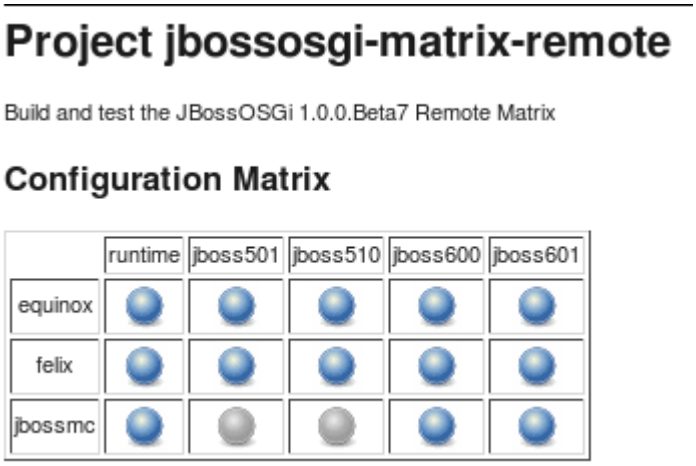
Setup the Hudson QA Environment

The JBoss OSGi *Hudson QA Environment* [http://jbmuc.dyndns.org:8280/hudson] 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, especially these

- hudson.maven.path
- hudson.username
- hudson.password
- hudson.root

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

# Hudson Workspace Root
# hudson.root=/home/username/workspace/hudson/jboss-osgi

# Hudson QA Environment
# hudson.username=username

#hudson.jboss501.zip=file:///home/username/Download/java/jboss/jboss-5.0.1.GA.zip
#hudson.jboss510.zip=file:///home/username/Download/java/jboss/jboss-5.1.0.GA.zip
#hudson.jboss600.zip=file:///home/username/Download/java/jboss/jboss-6.0.0.M2.zip

# JDK settings
java.home.jdk15=/usr/java/jdk1.5.0_22
java.home.jdk16=/usr/java/jdk1.6.0_17

# Maven settings
hudson.maven.name=apache-maven-2.2.1
```

```
hudson.maven.path=/usr/java/apache-maven-2.2.1
hudson.maven.profile=$HUDSONDIR/profiles.xml.local.qa

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

hudson.host=localhost
hudson.admin.port=8250
hudson.http.port=8280

hudson.mail.recipients=
hudson.mail.admin=yourname@yourdomain.com
hudson.smtp.host=localhost

apache-tomcat=5.5.27
sun-hudson=1.336
```

Run Hudson Setup

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

init:
    [echo] V1.0.0.Beta7

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/jboss-osgi/apache-tomcat
    ...
    [echo]
    [echo] ************************************
    [echo] * Hudson setup successfully         *
    [echo] * ant hudson-start                  *
```

```
    [echo] ************************************
    [echo]
```

Run Hudson Start

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

init:
    [echo] V1.0.0.Beta7

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-start:
    [echo]
    [echo] ************************************
    [echo] * Hudson started successfully       *
    [echo] * http://localhost:8280/hudson      *
    [echo] ************************************
    [echo]

BUILD SUCCESSFUL
```

Run Hudson Stop

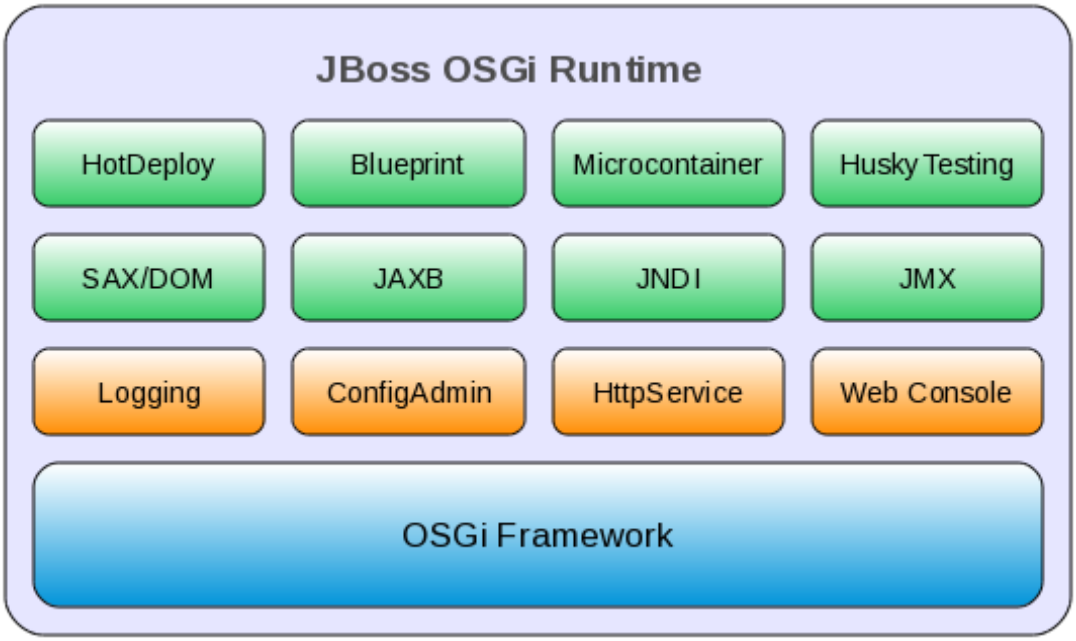
```
$ ant hudson-stop
Buildfile: build.xml

init:
    [echo] V1.0.0.Beta7

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-stop:
    [echo]
    [echo] ************************************
    [echo] * Hudson stopped successfully       *
    [echo] * ant hudson-start                   *
    [echo] ************************************
    [echo]

BUILD SUCCESSFUL
```

JBoss OSGi Runtime

3.1. Overview

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



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 frameworks. It comes with a choice of *Apache Felix*, *Eclipse Equinox* or *JBoss Microcontainer*.

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

Integration of the *JBoss Microcontainer* [http://www.jboss.org/jbossmc] as an OSGi service allows you to write your applications in a POJO programming model without much "pollution" of OSGi specific API - the MC will do the wiring for you. JBoss OSGi also comes with an implementation of *Blueprint Service (RFC-124)* [http://jbossosgi.blogspot.com/2009/04/osgi-blueprint-service-rfc-124.html], which standardizes this idea and takes it further.

Great care has been taken about testability of deployed components and services. The *Husky Test Framework* allows you to write plain JUnit tests that do not have a requirement on a specific test

runner nor 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* [http://maven.apache.org], *Ant* [http://ant.apache.org], *Eclipse* [http://www.eclipse.org] or any other test runner that supports JUnit4.

JBoss OSGi Runtime can be installed as a *JBossAS* [http://http://jboss.org/jbossas] service with abstractions of the available OSGi services. The JBoss OSGi testsuite in fact runs the same set of tests against the embedded, standalone and AS integrated instance of the Runtime

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 startup 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* [http://http://jboss.org/jbossas] there is a deployment scanner that scans the 'deploy' folder for new or removed bundles.
- **Multiple OSGi Frameworks** - The Installer can setup the JBoss OSGi Runtime using *Felix*, *Equinox* or *JBossMC*.
- **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* [http://www.osgi.org/javadoc/r4v41/org/osgi/util/xml/XMLParserActivator.html] which provides access to a SAXParserFactory and DocumentBuilderFactory.
- **JAXB Support** - There is a bundle that provides JAXB support.
- **HttpService and WebApp Support** - HttpService and WebApp support is provided by *Pax Web* [http://wiki.ops4j.org/display/paxweb/Pax+Web].
- **ConfigAdmin Support** - ConfigAdmin support is provided by the *Apache Felix Configuration Admin Service* [http://felix.apache.org/site/apache-felix-configuration-admin-service.html].
- **EventAdmin Support** - EventAdmin support is provided by the *Apache Felix Event Admin Service* [http://felix.apache.org/site/apache-felix-event-admin.html].

- **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).
- **Microcontainer Support** - The *Microcontainer* [http://www.jboss.org/jbossmc] service allows bundles to contain a *-beans.xml descriptor, which can be used for component wiring and injection of base services. It also comes with a set of *deployers* - so instead of simply installing a bundle to the underlying OSGi framework it passes to the chain of deployers which each deal with a specific aspect of bundle deployment.
- **Blueprint Container Support** - The *Blueprint Container* [http://jbossosgi.blogspot.com/2009/04/osgi-blueprint-service-rfc-124.html] 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.Beta7/runtime

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

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

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

12:10:48,713 INFO  JBossOSGi Core Framework - 1.0.0.Alpha4
12:10:49,089 INFO  Bundle INSTALLED: Bundle{osgi.cmpn:4.2.0.200908310645}
12:10:49,188 INFO  Bundle INSTALLED: Bundle{org.apache.felix.log:1.0.0}
12:10:49,282 INFO  Bundle INSTALLED: Bundle{jboss-osgi-common:1.0.2}
12:10:49,313 INFO  Bundle INSTALLED: Bundle{jboss-osgi-hotdeploy:1.0.2}
12:10:50,047 INFO  Bundle STARTED: Bundle{jboss-osgi-hotdeploy:1.0.2}
```

```
12:10:50,050 INFO  Bundle STARTED: Bundle{system.bundle:0.0.0}
12:10:50,076 INFO  JBossOSGi Runtime booted in 1.357sec
```

Minimal Profile

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

The following bundles are installed:

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

Default Profile

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

These additional bundles are installed:

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

Web Profile

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

These additional bundles are installed:

- **org.apache.felix.configadmin.jar** - Apache Config Admin service
- **pax-web-jetty-bundle.jar** - Pax Web HttpService
- **pax-web-extender-war.jar** - Pax Web WebApp Extender
- **jboss-osgi-webconsole.jar** - JBoss OSGi Web Console

All Profile

The 'all' profile extends the 'web' profile by SAX/DOM, JAXB, JBossXB and Microcontainer

These additional bundles are installed:

- **jboss-osgi-apache-xerces.jar** - Apache Xerces support
- **jboss-osgi-jaxb.jar** - JAXB support
- **jboss-osgi-jta.jar** - JTA support
- **jboss-osgi-xml-binding.jar** - XML Binding (JBossXB) support
- **jboss-osgi-microcontainer.jar** - Microcontainer support
- **jboss-osgi-blueprint.jar** - Blueprint Container support

Framework Integration

4.1. JBossMC Framework Integration

Starting from 1.0.0.Beta4 JBoss OSGi provides integration for our native *Microcontainer based* [http://www.jboss.org/jbossmc/] OSGi Framework. When deployed in JBossAS this Framework will eventually allow us to integrate with components from other programming models. (i.e. OSGi services can access MC beans, EJB3 can access OSGi services and vice versa)

JBossMC integration can be configured through an XML beans configuration in the *JBoss OSGi Runtime*.

```
cat server/default/conf/jboss-osgi-bootstrap.xml

<deployment xmlns="urn:jboss:bean-deployer:2.0">

 <!-- The OSGiFramework -->
                                        <bean                    name="OSGiBundleManager"
class="org.jboss.osgi.framework.bundle.OSGiBundleManager">
 <property name="properties">
   ...
 </property>
 </bean>
...
</deployment>
```

In the *JBossAS* [http://www.jboss.org/jbossas] integration we also use *JBoss Microcontainer* [http://www.jboss.org/jbossmc] beans configuration.

4.2. Apache Felix Integration

JBoss OSGi provides integration for the *Apache Felix* [http://felix.apache.org] OSGi Framework and some of its core services

- *Log Service* [http://felix.apache.org/site/apache-felix-log-service.html] - General purpose message logger
- *Config Admin Service* [http://felix.apache.org/site/apache-felix-configuration-admin-service.html] - Management of configuration data for configurable components

The Apache Felix integration can be configured through properties in the *JBoss OSGi Runtime*.

```
cat conf/jboss-osgi-framework.properties

# Properties to configure the Framework
org.osgi.framework.storage=${osgi.server.home}/data/osgi-store
org.osgi.framework.storage.clean=onFirstInit

# Hot Deployement
org.jboss.osgi.hotdeploy.scandir=${osgi.server.home}/bundles

...

# Bundles that need to be installed with the Framework automatically
org.jboss.osgi.spi.framework.autoInstall=\
  file://${osgi.home}/server/minimal/bundles/org.osgi.compendium.jar

# Bundles that need to be started automatically
org.jboss.osgi.spi.framework.autoStart=\
   file://${osgi.home}/server/minimal/bundles/org.apache.felix.log.jar \
   file://${osgi.home}/server/minimal/bundles/jboss-osgi-common.jar \
   file://${osgi.home}/server/minimal/bundles/jboss-osgi-hotdeploy.jar
```

In the *JBossAS* [http://www.jboss.org/jbossas] integration we use *JBoss Microcontainer* [http://www.jboss.org/jbossmc] beans configuration.

```
cat server/default/deployers/osgi.deployer/META-INF/osgi-deployers-jboss-beans.xml

<deployment xmlns="urn:jboss:bean-deployer:2.0">

  <!-- The OSGiFramework -->
  <bean name="jboss.osgi:service=Framework" class="org.jboss.osgi.felix.FelixIntegration">
    <property name="properties">
       ...
    </property>
    <property name="autoStart">
     <list elementClass="java.net.URL">
      ...
     </list>
    </property>
  </bean>
```

```
  ...
</deployment>
```

The following is a description of the configuration properties for the Apache Felix integration.

Table 4.1.

Key	Value	Description
org.osgi.framework.storage	.../osgi-store	OSGi Framework storage area
org.osgi.framework.storage.clean	onFirstInit	Clean the storage area on first init
org.osgi.service.http.port	8090	The default Http Service port
felix.cm.dir	.../osgi-configadmin	Config Admin Service storage area
org.osgi.framework.system.packages.extra	javax.management, javax.xml...	Packages provided by the OSGi System ClassLoader
org.jboss.osgi.deferred.start	true	Bundles can be deployed in any order

4.3. Equinox Integration

JBoss OSGi also provides basic integration for the *Eclipse Equinox* [http://www.eclipse.org/equinox] OSGi Framework.

Equinox integration can be configured through properties in the *JBoss OSGi Runtime*.

```
cat conf/jboss-osgi-framework.properties

# Properties to configure the Framework
org.osgi.framework.storage=${osgi.server.home}/data/osgi-store
org.osgi.framework.storage.clean=onFirstInit

# Hot Deployement
org.jboss.osgi.hotdeploy.scandir=${osgi.server.home}/bundles

...

# Bundles that need to be installed with the Framework automatically
org.jboss.osgi.spi.framework.autoInstall=\
 file://${osgi.home}/server/minimal/deploy/org.eclipse.osgi.services.jar \
 file://${osgi.home}/server/minimal/deploy/org.eclipse.osgi.util.jar
```

```
# Bundles that need to be started automatically
org.jboss.osgi.spi.framework.autoStart=\
  file://${osgi.home}/server/minimal/bundles/org.apache.felix.log.jar \
  file://${osgi.home}/server/minimal/bundles/jboss-osgi-common.jar \
  file://${osgi.home}/server/minimal/bundles/jboss-osgi-hotdeploy.jar
```

In the *JBossAS* [http://www.jboss.org/jbossas] integration we use *JBoss Microcontainer* [http://www.jboss.org/jbossmc] beans configuration.

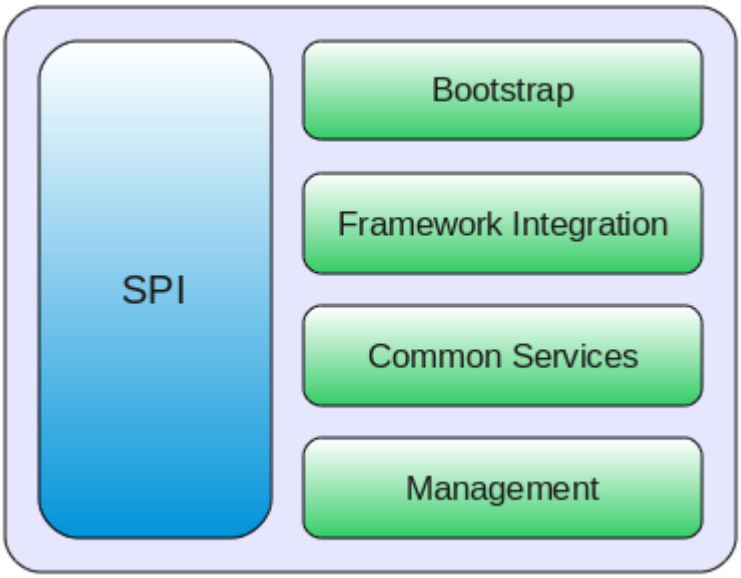
```
<deployment xmlns="urn:jboss:bean-deployer:2.0">

 <!-- The OSGiFramework -->
                                    <bean                    name="jboss.osgi:service=Framework"
class="org.jboss.osgi.equinox.EquinoxIntegration">
   <property name="properties">
     ...
   </property>
   <property name="autoStart">
    <list elementClass="java.net.URL">
     ...
    </list>
   </property>
 </bean>
 ...
</deployment>
```

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



The latest version of the *JBoss OSGi SPI* [../../apidocs].

- *org.jboss.osgi.spi* [../../apidocs/org/jboss/osgi/spi/package-summary.html] - Common classes and interfaces.
- *org.jboss.osgi.spi.capability* [../../apidocs/org/jboss/osgi/spi/capability/package-summary.html] - Capabilities that can be installed in the OSGi framework.
- *org.jboss.osgi.spi.framework* [../../apidocs/org/jboss/osgi/spi/framework/package-summary.html] - Framework integration and bootstrap.
- *org.jboss.osgi.spi.service* [../../apidocs/org/jboss/osgi/spi/service/package-summary.html] - A collection of SPI provided services.
- *org.jboss.osgi.spi.util* [../../apidocs/org/jboss/osgi/spi/util/package-summary.html] - A collection of SPI provided utilities.
- *org.jboss.osgi.testing* [../../apidocs/org/jboss/osgi/testing/package-summary.html] - Test support classes and interfaces.

Bootstrapping JBoss OSGi

The 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 qalified name of the `org.jboss.osgi.spi.framework.OSGiBootstrapProvider` interface.

The following code shows how to get the default `OSGiFramework` from the `OSGiBootstrapProvider`.

```
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* [../../apidocs/org/jboss/osgi/spi/framework/PropertiesBootstrapProvider.html]

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

Using the SPI from within JBossAS deployed components

If you need access to the OSGi Framework from within an JBossAS deployed component (e.g. servlet, ejb, mbean) you **would not** bootstrap JBoss OSGi

through the SPI. Instead, you would inject the already bootstrapped OSGi Framework instance into your component.

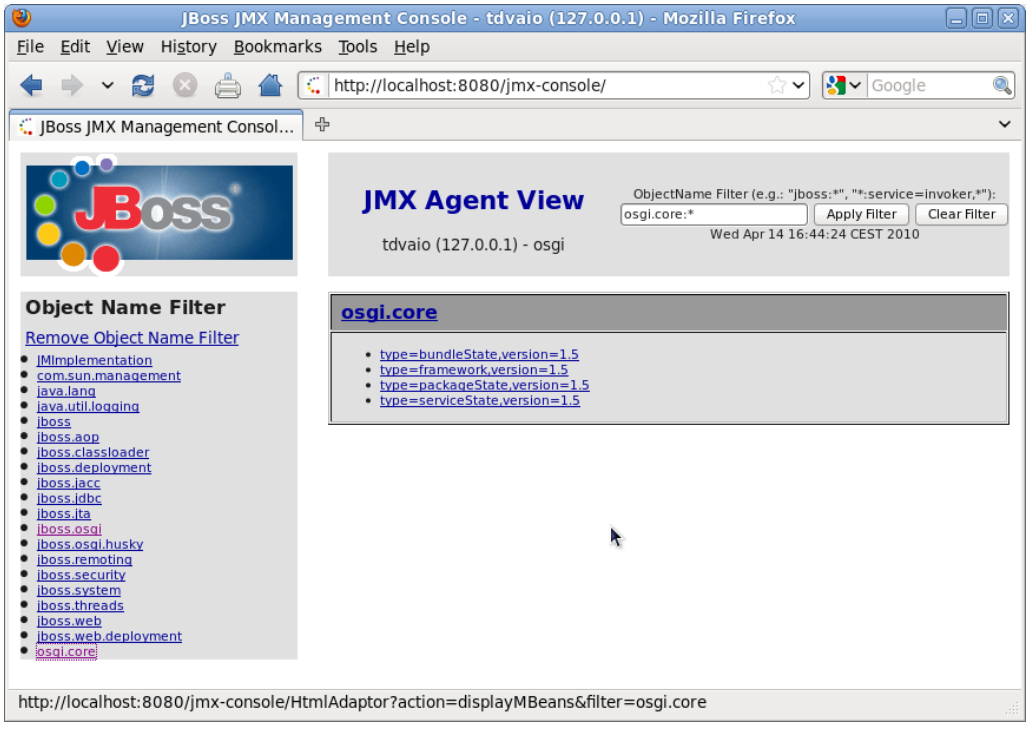
5.2. Management View

JBoss OSGi provides standard *org.osgi.jmx* [http://www.osgi.org/javadoc/r4v42/org/osgi/jmx/package-frame.html] management. Additional to that we provide an *MBeanServer* [http://java.sun.com/j2se/1.5.0/docs/api/javax/management/MBeanServer.html] service and a few other extensions through the *org.jboss.osgi.jmx* [../../apidocs/org/jboss/osgi/jmx/package-summary.html] API

Accessing the Management Objects

If you work with the JBoss OSGi runtime abstraction you get access to these managed objects through *OSGiRuntime* [../../apidocs/org/jboss/osgi/testing/OSGiRuntime.html].

If you install JBoss OSGi in an already existing JBossAS instance you also get access to the Managed Objects through the JBoss provided JMX Console (*http://localhost:8080/jmx-console*).



Note

The JMX Console is **not part** of the *JBoss OSGi Runtime*.

5.3. Writing Test Cases

JBoss OSGi comes with *JUnit* [http://www.junit.org] test support as part of the SPI provided *org.jboss.osgi.testing* [../../apidocs/org/jboss/osgi/testing/package-summary.html] package. There are two distinct test scenarios that we support:

- Embedded OSGi Framework
- Remote OSGi Framework

The remote scenario can actually be separated again in:

- Standalone JBoss OSGi Runtime
- JBoss OSGi Runtime running in JBossAS

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

5.3.1. Simple Test Case

The test case bootstraps the OSGi Runtime, installes/starts the bundle, asserts the bundle state and finally shuts down the runtime again. Please note, this is a plain JUnit4 test case that transparently handles embedded/remote nature of the runtime.

```
public class SimpleTestCase extends OSGiTest
{
  @Test
  public void testSimpleBundle() throws Exception
  {
    // Get the default runtime
    OSGiRuntime runtime = getDefaultRuntime();

    try
    {
      // Install the bundle
      OSGiBundle bundle = runtime.installBundle("example-simple.jar");

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

      // Uninstall the bundle
      bundle.uninstall();
    }
    finally
```

```
    {
      // Shutdown the runtime
      runtime.shutdown();
    }
  }
}
```

To run the test in embedded mode (which is the default) you would execute your test runner like this

```
[tdiesler@tddell example]$ mvn -Dtest=SimpleTestCase test
...
Running org.jboss.test.osgi.example.simple.SimpleTestCase
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.361 sec
```

To run the test against the remote JBoss OSGi Runtime you would execute your test runner like this

```
[tdiesler@tddell example]$ mvn -Dtarget.container=runtime -Dtest=SimpleTestCase test
...
Running org.jboss.test.osgi.example.simple.SimpleTestCase
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.303 sec
```

In the runtime console you should see

```
12:44:30,960 INFO  [jboss-osgi-common] Installed: example-simple [8]
12:44:31,081 INFO  [example-simple] Start: example-simple [8]
12:44:31,089 INFO  [example-simple] Stop: example-simple [8]
12:44:31,095 INFO  [jboss-osgi-common] Uninstalled: example-simple [8]
```

Due to classloading restrictions it is not possible to interact with the services that get registered in the OSGi Framework directly. Instead, there must be some means for the bundle under test to communicate with the test case that lives outside the Framework. The approach of OSGi

testing based on remote log messages is covered in *Non intrusive OSGi Bundle Testing* [http://jbossosgi.blogspot.com/2009/04/non-intrusive-osgi-bundle-testing.html].

The next section explains how to write a plain JUnit test that is then executed within the OSGi Runtime.

5.3.2. Simple Husky Test Case

The test case does everthing identical to *SimpleTestCase*, but only executes the code in the test method when Husky injected the BundleContext.

```
public class SimpleHuskyTestCase
{
  @ProvideContext
  public BundleContext context;
  ...
  @Test
  public void testSimpleBundle() throws Exception
  {
    // Tell Husky to run this test method within the OSGi Runtime
    if (context == null)
      BridgeFactory.getBridge().run();

    // Stop here if the context is not injected
    assumeNotNull(context);

    // Get the SimpleService reference
    ServiceReference sref = context.getServiceReference(SimpleService.class.getName());
    assertNotNull("SimpleService Not Null", sref);

    // Access the SimpleService
    SimpleService service = (SimpleService)context.getService(sref);
    assertEquals("hello", service.echo("hello"));
  }
}
```

Running this test is also no different from *SimpleTestCase*.

In the runtime console you should see

```
    13:29:15,924 INFO  [jboss-osgi-common] Installed: example-simple-husky [16]
```

```
    13:29:15,972 INFO  [example-simple-husky] Start: example-simple-husky [16]
    13:29:15,981 INFO  [jboss-osgi-husky] Test-Package [org.jboss.test.osgi.example.simple] in
bundle: example-simple-husky [16]
    13:29:16,160 INFO  [example-simple-husky] echo: hello
    13:29:16,191 INFO  [example-simple-husky] Stop: example-simple-husky [16]
    13:29:16,196 INFO  [jboss-osgi-common] Uninstalled: example-simple-husky [16]
```

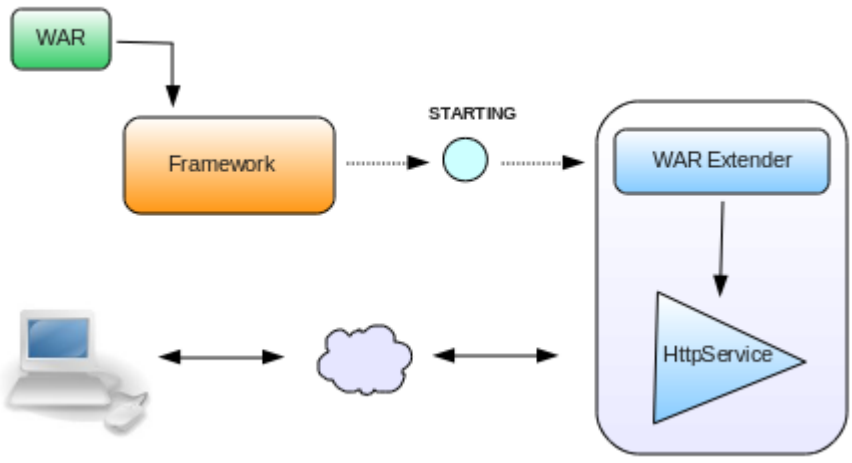
To learn more about the magic of the BridgeFactory have a look at *Husky Test Framework* which comes next.

5.4. 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* [http://www.osgi.org/blog/2007/02/osgi-extender-model.html] is often being used. JBoss OSGi offeres a differnet approach to address this problem which is covered by the *Extender Pattern vs. Lifecycle Interceptor* [http://jbossosgi.blogspot.com/2009/10/extender-pattern-vs-lifecycle.html] post in the *JBoss OSGi Diary* [http://jbossosgi.blogspot.com/].
Extending an OSGi Bundle

1. Extender registers itself as BundleListener
2. Bundle gets installed/started
3. Framework fires a BundleEvent
4. Extender picks up the BundleEvent (e.g. STARTING)
5. 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.



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).

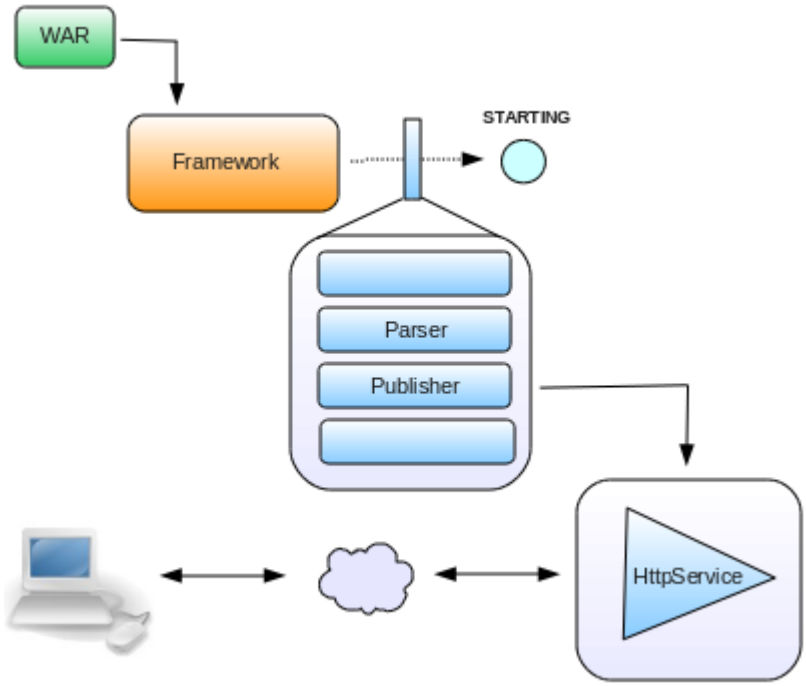
As a consequence the extender pattern is useful in some cases but not all. It is mainly useful if a bundle can optionally be extended in the true sense of the word.

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.



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*.

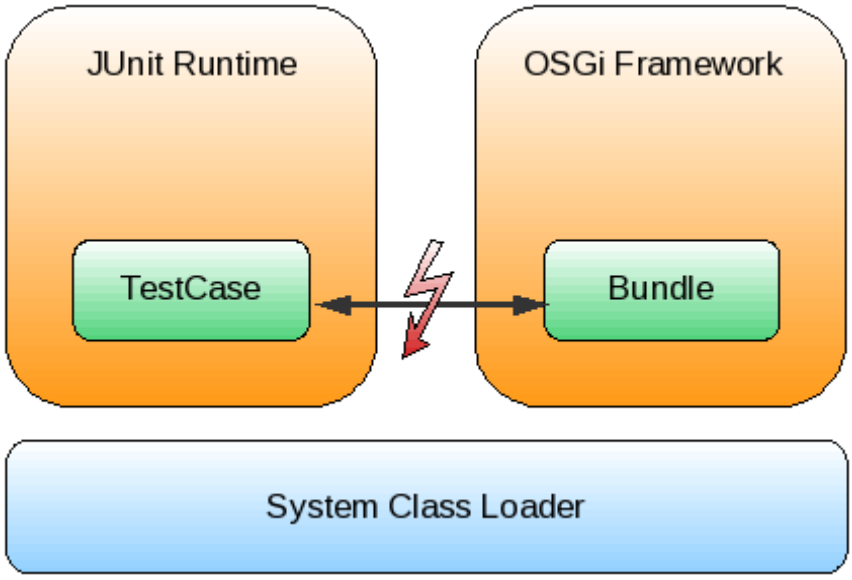
Husky Test Framework

6.1. Overview

JBoss OSGi Husky is a OSGi 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 nor do you need a special test runner. 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.



Former releases of JBoss OSGi used an approach which is documented in *Non intrusive OSGi Bundle Testing* [http://jbossosgi.blogspot.com/2009/04/non-intrusive-osgi-bundle-testing.html] and is still available. Although the remote logging approach worked for simple scenarios, it does not allow for fine grained interaction with the OSGi Framework (i.e. access to the registry). An additional problem was the asynchronous nature of LogEntry delivery.

For this release however, I revisited the problem and added a few more requirements.

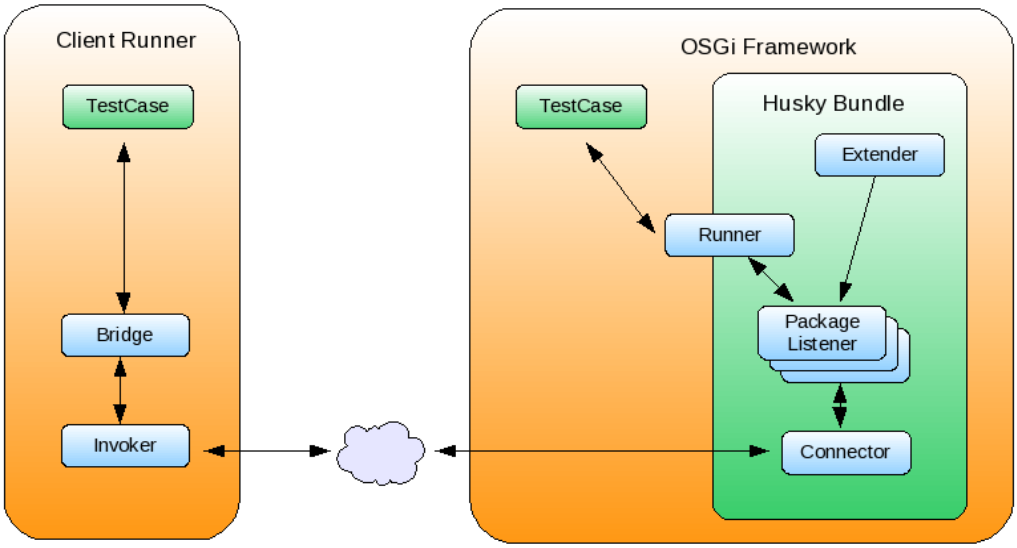
- Test cases SHOULD be plain JUnit4 POJOs
- There SHOULD be no requirement to extend a specific test base class
- There MUST be no requirement on a specific test runner (i.e. MUST run with Maven)

- 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

The next section explains the solution that now comes as JBoss OSGi Husky

6.2. Architecture

JBoss OSGi Husky has client side interceptor that fields the test request to an embedded/remote OSGi Framework where the test case is then actually executed.



Here is how it works

1. A *Bridge* [../../apidocs/org/jboss/osgi/husky/Bridge.html] intercepts a test and determines the FQN of the test case and the test method from the call stack. It then delegates the execution to the same (or another) test in and isolated test environment. An isolated test environment is one that does not have the same class loading space as the test itself.
2. A Bridge is associated with an *Invoker* [../../apidocs/org/jboss/osgi/husky/Invoker.html]. Invokers may be arbitarily complex. Local 'in proccess' invokers are possible just as well as remote invokers.

3. The Invoker sends the *Request* [../../apidocs/org/jboss/osgi/husky/Request.html] to a *Connector* [../../apidocs/org/jboss/osgi/husky/runtime/Connector.html] in the isolated test environment.
4. A Connector has associated *PackageListeners* [../../apidocs/org/jboss/osgi/husky/runtime/PackageListener.html] that are responsible for processing test cases for their respective test packages.
5. A PackageListeners delegates the Request to a test *Runner* [../../apidocs/org/jboss/osgi/husky/runtime/Runner.html], typicaly this would be a *JUnitRunner* [../../apidocs/org/jboss/osgi/husky/runtime/junit/JUnitRunner.html].
6. The Runner injects the *Context* [../../apidocs/org/jboss/osgi/husky/Context.html] into the test case and returns a *Response* [../../apidocs/org/jboss/osgi/husky/Response.html], which the Connector returns to the Invoker.
7. The Bridge finally translates potential *Failures* [../../apidocs/org/jboss/osgi/husky/Failure.html] that may be contained in the Result, to test failures on the client side.

The JBoss OSGi **jboss-osgi-husky.jar** bundle registers the Connectors. The *JMXConnector* [../../apidocs/org/jboss/osgi/husky/runtime/osgi/JMXConnector.html] is always registered. The *SocketConnector* [../../apidocs/org/jboss/osgi/husky/runtime/osgi/SocketConnector.html] is registered when the appropriate configuration options are set. It then registers the *HuskyExtender* [../../apidocs/org/jboss/osgi/husky/runtime/osgi/HuskyExtender.html], which is a *BundleListener* [http://www.osgi.org/javadoc/r4v41/org/osgi/framework/BundleListener.html] that inspects every incomming bundle for the **Test-Package** manifest header. The Extender creates a PackageListener for every package in the 'Test-Package' manifest header and registers them with the available Connectors.

6.3. Configuration

In the target OSGi Framework, which is the one that has the **jboss-osgi-husky.jar** bundle installed, you set these properties

Table 6.1.

Key	Value	Description
org.jboss.osgi.husky.runtime.connector.host	localhost	The Husky socket connector host poperty
org.jboss.osgi.husky.runtime.connector.port	5401	The Husky socket connector port poperty

Both properties must be set for the SocketConnector to be available.

On the client side, you must configure the Invoker you want to use.

Table 6.2.

Key	Value	Description

org.jboss.osgi.husky.Invoker	org.jboss.osgi.husky.internal.OSGiInvoker	The client side Husky Invoker

This particular invoker will also look for the 'org.jboss.osgi.husky.runtime.connector.host' and 'org.jboss.osgi.husky.runtime.connector.port' properties and if available will use a socket invocation.

6.4. Writing Husky Tests

In a typical Husky test you have

- A **descriminator** to distinguish between client and 'in container' execution
- An **interceptor** that delegates the call to an Invoker (i.e. Bridge.run())

For OSGi, the descriminator would be the *BundleContext* [http://www.osgi.org/javadoc/r4v41/org/osgi/framework/BundleContext.html] that gets injected by the 'in container' test Runner

The interceptor would be a call to one of the Bridge.run() variants.

```
public class SimpleHuskyTestCase
{
  @ProvideContext
  public BundleContext context;
  ...
  @Test
  public void testSimpleBundle() throws Exception
  {
    // Tell Husky to run this test method within the OSGi Runtime
    if (context == null)
      BridgeFactory.getBridge().run();

    // Stop here if the context is not injected
    assumeNotNull(context);

    // Get the SimpleService reference
    ServiceReference sref = context.getServiceReference(SimpleService.class.getName());
    assertNotNull("SimpleService Not Null", sref);

    // Access the SimpleService
    SimpleService service = (SimpleService)context.getService(sref);
    assertEquals("hello", service.echo("hello"));
  }
}
```

The bundle that contains the test case must have the **Test-Package** manifest header configured. Here is the *aQute Bnd Tool* [http://www.aqute.biz/Code/Bnd] configuration for doing so.

```
Bundle-SymbolicName: example-simple-husky

Bundle-Activator: org.jboss.test.osgi.example.simple.bundle.SimpleActivator

Private-Package: org.jboss.test.osgi.example.simple.bundle

# Export the package that contains tthe test case
Export-Package: org.jboss.test.osgi.example.simple

# Tell Husky that there are test cases in this package
Test-Package: org.jboss.test.osgi.example.simple
```

Provided Bundles and Services

7.1. Blueprint Container Service

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

The *Blueprint Container* [http://jbossosgi.blogspot.com/2009/04/osgi-blueprint-service-rfc-124.html] 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*** **[http://www.osgi.org/javadoc/r4v42/org/osgi/service/blueprint/container/package-summary.html]**
- ***org.osgi.service.blueprint.reflect*** **[http://www.osgi.org/javadoc/r4v42/org/osgi/service/blueprint/reflect/package-summary.html]**

7.2. HttpService

The **pax-web-jetty-bundle.jar** bundle from the OPS4J *Pax Web* [http://wiki.ops4j.org/display/paxweb/Pax+Web] project provides access to the *HttpService* [http://www.osgi.org/javadoc/r4v42/org/osgi/service/http/package-frame.html].

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.

Table 7.1.

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*** **[http://www.osgi.org/javadoc/r4v42/org/osgi/service/http/HttpService.html]**

7.3. JAXB Service

The JBoss OSGi **jboss-osgi-jaxb.jar** bundle provides a service to create *JAXBContext* [http://java.sun.com/javase/6/docs/api/javax/xml/bind/JAXBContext.html] instances.

The service is registered with the Framework under the name

- ***org.jboss.osgi.jaxb.JAXBService*** **[../../apidocs/org/jboss/osgi/jaxb/JAXBService.html]**

7.4. JMX Service

The JBoss OSGi **jboss-osgi-jmx.jar** bundle activator discovers and registers the *MBeanServer* [http://java.sun.com/javase/6/docs/api/javax/management/MBeanServer.html] 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.

Table 7.2.

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*** **[http://java.sun.com/javase/6/docs/api/javax/management/MBeanServer.html]**
- ***org.jboss.osgi.jmx.FrameworkMBeanExt*** **[../../apidocs/org/jboss/osgi/jmx/FrameworkMBeanExt.html]**
- ***org.jboss.osgi.jmx.BundleStateMBeanExt*** **[../../apidocs/org/jboss/osgi/jmx/BundleStateMBeanExt.html]**
- ***org.jboss.osgi.jmx.PackageStateMBeanExt*** **[../../apidocs/org/jboss/osgi/jmx/PackageStateMBeanExt.html]**
- ***org.jboss.osgi.jmx.ServiceStateMBeanExt*** **[../../apidocs/org/jboss/osgi/jmx/ServiceStateMBeanExt.html]**

7.5. JNDI Service

The JBoss OSGi **jboss-osgi-jndi.jar** bundle activator creates and registers the *InitialContext* [http://java.sun.com/javase/6/docs/api/javax/naming/InitialContext.html] with the framework.

The JNDI Service is configured with these properties.

Table 7.3.

Key	Value	Description
org.jboss.osgi.jndi.host	localhost	The property that sets the naming server host
org.jboss.osgi.jndi.rmi.port	1098	The property that sets the naming server RMI port
org.jboss.osgi.jndi.port	1099	The property that sets the naming server port

Here is the complete list of services that this bundle provides

- ***javax.naming.InitialContext*** **[http://java.sun.com/javase/6/docs/api/javax/naming/InitialContext.html]**

7.6. JTA Service

The JBoss OSGi **jboss-osgi-jta.jar** bundle registers two services with framework.

- ***javax.transaction.TransactionManager*** **[http://java.sun.com/javaee/5/docs/api/javax/transaction/TransactionManager.html]**
- ***javax.transaction.UserTransaction*** **[http://java.sun.com/javaee/5/docs/api/javax/transaction/UserTransaction.html]**

Among others the JTA Service can be configured with these properties.

Table 7.4.

Key	Value	Description
com.arjuna.ats.arjuna.objectstore.objectStoreDir	${jboss.server.data.dir}/tx-object-store	The property that sets the transaction object store directory

For details please refer to the *JBossTM documentation* [http://www.jboss.org/jbosstm/docs/index.html].

7.7. Microcontainer Service

The JBoss OSGi Microcontainer Service gives access to the *JBoss Microcontainer* [http://www.jboss.org/jbossmc] Kernel. The service is registered with the Framework under the name.

- ***org.jboss.osgi.spi.service.MicrocontainerService*** **[../../apidocs/org/jboss/osgi/spi/service/MicrocontainerService.html]**

Here is an example of how an OSGi component can access an arbitrary MC bean.

```
public class SomeService
{
  public String callSomeBean(String msg)
  {
                                                    ServiceReference sref =
 context.getServiceReference(MicrocontainerService.class.getName());
    MicrocontainerService mcService = (MicrocontainerService)context.getService(sref);
    SomeBean bean = (SomeBean)mcService.getRegisteredBean("SomeBean");
    return bean.echo(msg);
  }
}
```

7.8. ServiceLoader Interceptor

The ServiceLoader, deployed as **jboss-osgi-serviceloader.jar** bundle, is a *Lifecycle Interceptor* that automatically registers services declared in META-INF/services.

For more information, please hava a look at *ServiceLoader and how it relates to OSGi* [http://jbossosgi.blogspot.com/2010/01/suns-serviceloader-and-how-it-relates.html].

7.9. WebApp Extender

The **pax-web-extender-war.jar** bundle from the OPS4J *Pax Web* [http://wiki.ops4j.org/display/paxweb/WAR+Extender] project provides WAR processing functionality.
Deploying a WAR onto JBoss OSGi

You should have a war file compliant with Servlet specs. Additionally, the war file must have the necessary OSGi manifest headers.

- **Bundle-ManifestVersion: 2** - This header defines that the bundle follows the rules of R4 specification.
- **Bundle-SymbolicName** - This header specifies a unique, non-localizable name for this bundle.

There are also a number of other OSGi manifest headers that are processed by the WAR Extender. Please have a look at *OSGify your WAR* [http://wiki.ops4j.org/display/paxweb/OSGify+your+WAR] for details.

An example of how a bundle uses the WAR Extender to register servlet and resources is given in *WebApp Example*.

7.10. 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*** **[http://java.sun.com/javase/6/docs/api/javax/xml/parsers/SAXParserFactory.html]**
- ***javax.xml.parsers.DocumentBuilderFactory*** **[http://java.sun.com/javase/6/docs/api/javax/xml/parsers/DocumentBuilderFactory.html]**

Please see *XMLParserActivator* [http://www.osgi.org/javadoc/r4v41/org/osgi/util/xml/XMLParserActivator.html] for details.

7.11. XML Binding Services

The JBoss OSGi **jboss-osgi-xml-binding.jar** bundle provides an *JBossXB* [http://www.jboss.org/community/wiki/JBossXB] unmarshaller service.

The service is registered with the Framework under the name

- ***org.jboss.osgi.jbossxb.UnmarshallerService*** **[../../apidocs/org/jboss/osgi/jbossxb/UnmarshallerService.html]**

Provided Examples

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* [http://www.jboss.org/community/docs/DOC-13275] and *Hudson QA Environment* [http://www.jboss.org/community/docs/DOC-13420].

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
```

Suported target container values are:

- runtime
- jboss501
- jboss510
- jboss600
- jboss601

To run the examples against a different OSGi Framework, you need to define the **framework** system property.

```
mvn -Dframework=felix test
```

Suported framework values are:

- jbossmc
- equinox
- felix

8.2. Event Admin Example

The **example-event.jar** bundle uses the *EventAdmin* [http://www.osgi.org/javadoc/r4v42/org/osgi/service/event/EventAdmin.html] 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[] { 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 registerd 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* [http://www.osgi.org/javadoc/r4v41/org/osgi/service/http/HttpService.html].

```
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. JAXB Service

The **example-xml-jaxb.jar** bundle gets the JAXBContext from the JAXBService and unmarshalls an XML document using JAXB

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

JAXBService service = (JAXBService)context.getService(sref);
JAXBContext jaxbContext = service.newJAXBContext(getClass().getPackage().getName());
Unmarshaller unmarshaller = jaxbContext.createUnmarshaller();

URL resURL = context.getBundle().getResource("booking.xml");
JAXBElement rootElement = unmarshaller.unmarshal(resURL.openStream());
assertNotNull("root element not null", rootElement);
```

8.6. JMX 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"));
}
```

Note

Please note that access to the MBeanServer from the test case is part of the *OSGiRuntime* [../../apidocs/org/jboss/osgi/spi/testing/OSGiRuntime.html] abstraction.

8.7. 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
  }
}
```

Note

Please note that access to the InitialContext from the test case is part of the *OSGiRuntime* [../../apidocs/org/jboss/osgi/spi/testing/OSGiRuntime.html] abstraction.

8.8. JTA Service

The **example-jta.jar** bundle gets the *javax.transaction.UserTransaction* [http://java.sun.com/javaee/5/docs/api/javax/transaction/UserTransaction.html] service and registers a transactional user object (i.e. one that implements *Synchronization* [http://java.sun.com/javaee/5/docs/api/javax/transaction/Synchronization.html]) with the *javax.transaction.TransactionManager* [http://java.sun.com/javaee/5/docs/api/javax/transaction/TransactionManager.html] 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;
 }

 ...
}
```

8.9. Lifecycle Interceptor

The interceptor example deploys 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* [www.osgi.org/wiki/uploads/Links/whiteboard.pdf].

```
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");
    }
  }
}
```

8.10. Microcontainer Service

The **example-microcontainer.jar** bundle calls a service from an MC bean and vica versa. The MC bean gets the MBeanServer injected and registeres itself as an MBean.

The test accesses the registered MBean.

```
@Test
public void testServiceRoundTrip() throws Exception
{
   SomeBeanMBean someBean = MBeanProxy.get(SomeBeanMBean.class, MBEAN_NAME,
 runtime.getMBeanServer());
  assertEquals("hello", someBean.echo("hello"));
  assertEquals("hello", someBean.callSomeService("hello"));
}
```

8.11. 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"));
}
```

8.12. ServiceLoader Example

The ServiceLoader example uses three bundles - **example-serviceloader-api.jar, example-serviceloader-impl.jar, example-serviceloader-client.jar**. The implementation bundle contains a traditional service defined in META-INF/services. This service definition gets picked up by the ServiceLoader Interceptor and is automatically registered with the OSGi Framework.

For details and more background information, please hava a look at *ServiceLoader and how it relates to OSGi* [http://jbossosgi.blogspot.com/2010/01/suns-serviceloader-and-how-it-relates.html].

8.13. 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());
```

8.14. XML Unmarshaller Service

The **example-xml-binding.jar** bundle unmarshalls an XML document through the UnmarshallerService. This example is very similar to the *JAXB Example*. However, it uses JBossXB to do the unmarshalling.

```
ServiceReference sref = context.getServiceReference(UnmarshallerService.class.getName());
UnmarshallerService unmarshaller = (UnmarshallerService)context.getService(sref);

Bundle bundle = context.getBundle();
URL xsdurl = bundle.getEntry("booking.xsd");
URL xmlurl = bundle.getEntry("booking.xml");

unmarshaller.registerSchemaLocation("http://org.jboss.test.osgi.jbossxb.simple/booking.xsd",
 xsdurl.toExternalForm());
unmarshaller.addClassBinding(CourseBooking.NAMESPACE_XML_SIMPLE,
 CourseBooking.class);

CourseBooking booking = (CourseBooking)unmarshaller.unmarshal(xmlurl.toExternalForm());
```

References

Resources

- *JBoss OSGi Wiki* [http://www.jboss.org/community/wiki/JBoss OSGi]
- *JBoss OSGi Diary* [http://jbossosgi.blogspot.com]
- *Issue Tracking* [https://jira.jboss.org/jira/browse/JBOSGI]
- *Hudson QA* [http://jbmuc.dyndns.org:8280/hudson]
- *Subversion* [https://anonsvn.jboss.org/repos/jbossas/projects/jboss-osgi]
- *Fisheye* [http://fisheye.jboss.com/browse/JBoss OSGi]
- *User Forum* [http://www.jboss.org/index.html?module=bb&op=viewforum&f=257]
- *Design Forum* [http://www.jboss.org/index.html?module=bb&op=viewforum&f=256]

Authors

- *Thomas Diesler* [email:thomas.diesler@jboss.com]

Getting Support

We offer free support through the *JBoss OSGi User Forum* [http://www.jboss.org/index.html?module=bb&op=viewforum&f=257].

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* [http://www.jboss.com/services].