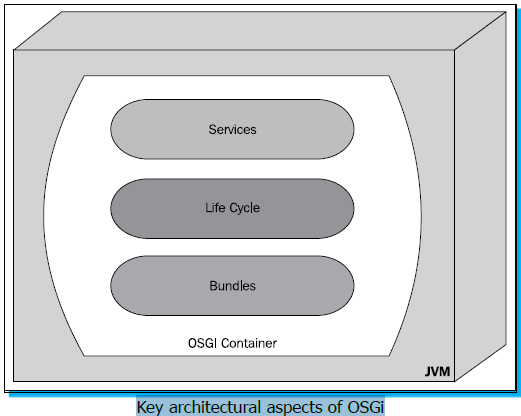
OSGi is a modular runtime for applications requiring a life cycle, deployment into a running container as well as re-use of services and libraries.

The basic function of OSGi is to provide Java developers with a component model that regulates code identity, versioning, interaction between code, life cycle, and making code requirements strictly enforceable.

This model is encapsulated by its three key aspects; **bundles**, **life cycle**, and **services**—all within a single OSGi container.



**Bundles**:

Developers are able to encapsulate functional portions of code into single deployable units, with explicitly stated imported and exported packages.

This allow for simplified management since the bundle has a clear function and environment.

A **bundle** is a jar with additional information. This information allows you to control what packages are

Imported, exported, as well as which ones are private and/or hidden.

In contrast to a regular classloading environment, you as a developer can control explicitly what you expose, share and how it is versioned.

OSGi provides these bundles with a **life cycle**. Once a bundle is resolved it can be freely started or stopped, updated, or removed remotely via an API. The framework handles the heavy lifting, ensuring that an installed bundle has all of its requirements met.

Finally, OSGi allows developers to take advantage of **micro-services**; these are services provided by one bundle to another in a dynamic fashion. The wiring between service producers and consumers is managed such that bundles adapt to the changing environment.

**How can you use OSGi within your existing applications?**

When we look closer at OSGi applications we'll see that very little or no additional code is required to make aspects of OSGi available to your application.

The essential addition is a few extra headers added to Java Archive manifest files. These headers are used to identify the bundle, its version, its imported dependencies from the environment, and any exported packages the bundle provides. In fact, just adding these few OSGi headers to a JAR manifest will create a valid OSGi bundle.

To take advantage of more advanced OSGi features, however, we'll have to introduce some additional code.

**Installation**

**Step 1 – downloading an OSGi core**

Download Felix : <http://felix.apache.org/site/downloads.cgi>.

(Alternatively install equinox <http://download.eclipse.org/equinox>.)

**Step 2 – starting up an OSGi core directly**

Run felix.jar present inside bin.

**$java –jar bin/felix.jar**

Enter **help** to view available commands. Enter **help <command\_name>** to view command manual. Enter **Ctrl+C** to exit.

**Step 3 – downloading Apache Karaf**

Using a bare OSGi framework can be an unwieldy experience for a first time OSGi developer. As such, we encourage new users to try an OSGi environment such as **Apache Karaf** or **Eclipse Virgo**. Apache Karaf, by default, uses Apache Felix as it is an OSGi framework while Eclipse

Virgo uses Eclipse Equinox. Both runtime containers provide an enhanced experience when working with OSGi environments.

A few highlights of using Apache Karaf include: Hot Deployment of OSGi bundles, Dynamic Configuration via OSGi's Configuration Admin service, a centralized logging system, built-in provisioning mechanisms to simplify gathering resources, native OS integration, an extensible shell console, remote access, security framework, and OSGi instance management, among other great features!

**Step 4 – starting Apache Karaf**

Starting Apache Karaf is as simple as executing the Karaf start script in the bin folder of your Karaf distribution.

You get a tab for completion of commands, a greatly expanded repertoire of commands and tooling that will help you develop, monitor, and deploy your projects. Note here that Apache Karaf can be installed as a system service, allowing the OSGi container to be started and stopped as any other service.

**Step 5 – obtaining the BND tool (Maven Bundle plugin)**

Building OSGi bundles is a relatively straightforward operation consisting of adding **OSGi headers** to your Java Archive's Manifest file. Our recommendation, however, is to use tools to handle generating the file entries. The easiest way to do this is to use the **Maven Bundle plugin**.

<dependency>

<groupId>org.apache.felix</groupId>

<artifactId>maven-bundle-plugin</artifactId>

<version>${felix.plugin.version}</version>

<!-- <scope>provided|compile|test</scope> -->

</dependency>

**Step 6 – obtaining Blueprint**

**Blueprint** is a dependency injection framework for OSGi, its job is to handle the wiring of JavaBeans, and instantiation and life cycle of an application. Blueprint also helps with the dynamic nature of OSGi where services can come and go at any time.

Blueprint is expressed as an XML file that defines and describes how the various components are assembled together, instantiated, and wired together to build an executing module.

<?xml version="1.0" encoding="UTF-8"?>

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

</blueprint>

**Step 8 – obtaining Pax Exam**

In a container, testing your OSGi applications is easy—just grab Pax Exam! This tool allows you to launch an OSGi framework, build bundles as defined in your test cases, and inject them into the container.

<dependency>

<groupId>org.ops4j.pax.exam</groupId>

<artifactId>pax-exam</artifactId>

<version>${pax.exam.version}</version>

</dependency>