CHAPTER 24



# OSGi and Spring

OSGi and Spring are, in many ways, a very natural technology combination. They approach different problems from different directions, but they do so in a similar spirit. It's only natural, then, that SpringSource, the company that stewards the Spring framework, should turn its eye to OSGi.

OSGi—which was formerly known as the Open Services Gateway initiative, though the name's obsolete now—has its roots in the embedded space, where dynamic service provisioning is far more important than it is in the gridiron world of enterprise applications. It provides a services registry as well as an application life cycle management framework. Beyond this, OSGi provides such features as granular component visibility via a highly specialized class-loading environment, service versioning and reconciliation, and security. OSGi provides a layer on top of the JVM's default class loader. The deployment unit for OSGi is a bundle, which is essentially a JAR with an augmented `MANIFEST.MF`. This manifest contains declarations that specify, among other things, on what other services the bundle depends, and what service the bundle exports.

OSGi has gained some notoriety because of Eclipse, which uses it for the plug-in model. This is a natural choice, because Eclipse needs to allow plug-ins to load and unload, and to guarantee that certain resources are made available to plug-ins. Indeed, the hope of an enhanced "module" system for Java has loomed large for many years, manifesting in at least a few JSRs: JSR-277, "Java Module System," and JSR-291, "Dynamic Component Support for Java SE." OSGi is a natural fit because it's been around for many years, matured, and has been improved on by many more vendors still. It is already the basis of the architecture of a few application servers.

OSGi is important today, more than ever, in the enterprise space because it represents a solution that can be gracefully layered on top of the Java Virtual Machine (JVM) (if not existing application servers) that can solve problems frequently encountered in today's environments. "`.jar` hell," the collision of two different versions of the same JAR in the same class loader, is something most developers have encountered. Application footprint reduction provides another compelling use of OSGi. Applications today, be they `.war` or `.ear`, are typically bundled with numerous `.jars` that exist solely to service that application's requirements. It may be that other applications on the same application server are using the same jars and services. This implies that there are duplicated instances of the same libraries loaded into memory. This situation's even worse when you consider how large typical deployed `.wars` are today. Most `.wars` are 90% third-party JARs, with a little application-specific code rounding out the mix. Imagine three `.wars` of 50 MBs, or 100 MBs, where only 5 MBs are application-specific code and libraries. This implies that the application server needs to field 300 MBs just to meet the requirements of a 15-30 unique MBs. OSGi provides a way of sharing components, loading them once, and reducing the footprint of the application.

Just as you may be paying an undue price for redundant libraries, so too are you likely paying for unused application server services, such as EJB1.x and 2.x support, or JCA. Here again, OSGi can help by providing a "server à la carte" model, where your application is provisioned by the container only with the services it needs.

OSGi is, on the large, a deployment concern. However, using it effectively requires changes to your code, as well. It affects how you acquire dependencies for your application. Naturally, this is where Spring is strongest and where dependency-injection in general can be a very powerful tool. SpringSource has made several forays into the OSGi market, first with Spring Dynamic Modules, which is an enhanced OSGi framework that provides support for Spring and much more. Then, on top of Spring Dynamic Modules, SpringSource built SpringSource dm Server, which is a server wired from top to bottom with OSGi and Spring. SpringSource dm Server supports dynamic deployment, enhanced tooling, HTTP, and native `.war` deployment. It also sports superb administrative features.

OSGi is a specification, not a framework. There are many implementations of the specification, just as there are many implementations of the Java EE specification. Additionally, OSGi is not a user component model, like Spring or EJB 3. Instead, it sits below your components, providing life-cycle management for Java classes. It is, conceptually, possible to deploy to an OSGi runtime in the same way that you deploy to a Java EE runtime, completely unaware of how Java consumes your `.jar` files and manifests and so on. As you'll see in this chapter, however, there's a lot of power to be had in specifically targeting OSGi and exploiting it in your application. In this chapter, we will discuss Spring Dynamic Modules, and to a lesser extent, Spring dm Server.

# 24-1. Getting Started with OSGi

## Problem

You understand OSGi conceptually, but you want to see what a basic, working example with raw OSGi looks like. It's hard to appreciate the sun, after all, if you've never seen the rain.

## Solution

In this solution, we'll build a simple service and then use it in a client. Remember, in OSGi, anything used by something else is a service. "Service" doesn't imply any concrete inheritance; it doesn't imply transactional qualities, and it doesn't imply RPC. It's merely a class on whose concrete, black-box functionality and interface your class relies.

## How It Works

In this example, we'll use Eclipse's OSGi distribution, Eclipse Equinox. There are many distributions to choose from. Popular ones include Apache's Felix and Eclipse's Equinox. You may use any distribution you want, but for this example, the instructions will be for Felix. The concepts should be the same across implementations, but the specifics may vary wildly in both commands and mechanism.