In Pieces
Breaking down monolithic applications with Spring-DM and OSGi

Agenda

- The problem: Lack of modularity
- OSGi Basics
- OSGi without Spring-DM
- Introducing Spring-DM
- Spring-DM and the web
- Blueprint Services
- So what?
- Moving to OSGi
Modularity is...

- High cohesion
  - Modules are focused in purpose
- Low coupling
  - Modules have minimal/no direct dependency on each other
- Not a new idea...

"A well-defined segmentation of the project effort ensures system modularity. Each task forms a separate, distinct program module. At implementation time each module and its inputs and outputs are well-defined, there is no confusion in the intended interface with other system modules. At checkout time the integrity of the module is tested independently; there are few scheduling problems in synchronizing the completion of several tasks before checkout can begin. Finally, the system is maintained in modular fashion; system errors and deficiencies can be traced to specific system modules, thus limiting the scope of detailed error searching."

Modularity promotes...

- Testability
- Comprehensibility
- Flexibility
- Reusability
- Plugability

Modularity in Java?

- Java lacks some essential features to support modular development
  - Classes encapsulate data and functionality
    - They're too fine-grained for practical modularity
  - Packages only contain classes
    - They're only an organizational mechanism
  - JAR files only contain packages/classes
    - Their boundaries dissolve when placed on the classpath
A false sense of modularity

JAR Hell
The solution...OSGi

- Modular framework for the OSGi platform
- Classpath++
- POJO-oriented
- Dynamic Runtime
  - Modules can be installed, started, stopped, updated, and uninstalled...in a live program
- Ends JAR Hell
  - Multiple versions of a class can reside in OSGi simultaneously

You think you’re modular?

Then why do you deploy your web applications as a single large WAR file?
Yeah, but...

Isn’t OSGi hard?

OSGi Basics
**Bundles**

- Just JAR files
- Contain special metadata in MANIFEST.MF
- All content is private by default
  - May export packages to be imported by other bundles
  - May publish services
- May be versioned

**Fragments**

- Just JAR files, like bundles, but...
- Must be hosted by another bundle
- Physically separate, logically united
- Used to add content (classes, resources, etc) to a hosting bundle
Bundle Lifecycle

OSGi Manifest

- Defines the content of a bundle

  Manifest-Version: 1.0
  Built-By: wallsc
  Created-By: Apache Maven Bundle Plugin
  Bundle-Activator: com.osgiknowhow.hello.consumer.internal.HelloConsumerActivator
  Import-Package: com.osgiknowhow.hello.service;version="1.0.0.SNAPSHOT",
                  org.osgi.framework, org.osgi.util.tracker
  Bnd-LastModified: 1236686261405
  Bundle-Version: 1.0.0.SNAPSHOT
  Ignore-Package: com.osgiknowhow.hello.consumer.internal
  Bundle-Name: com.osgiknowhow.hello.consumer
  Bundle-Description: Generated using Pax-Construct
  Build-Jdk: 1.5.0_16
  Private-Package: com.osgiknowhow.hello.consumer.internal
  Bundle-ManifestVersion: 2
  Bundle-SymbolicName: com.osgiknowhow.hello.consumer
  Tool: Bnd-0.0.255

- Don’t ever write this file yourself
  - Generate it
**Activators**

- Perform some functionality when a bundle is started and stopped
- Typically used to publish and/or consume services
- Should be quick...or else they’ll hold up the starting/stopping of a bundle

**Versioning**

- Bundles and packages can be versioned
  - Can even be versioned independent from each other
- Multiple versions can be available simultaneously
- Packages can be imported by specifying a specific version, a version range, or no version at all (implying an infinite version range)
Services

- Bundle functionality encapsulated behind services
- Bundles can publish/consume services
  - Identified by their interface(s) and optional parameters
  - Publish programatically using bundle context
  - Consume programatically using bundle context and service tracker
- Can be published/consumed declaratively
  - Declarative Services, iPOJO, Spring–DM

OSGi without Spring–DM
**Publishing services**

- Done from an activator, via the bundle context:

```java
public final class HelloServiceActivator implements BundleActivator {
    public void start(BundleContext bc) throws Exception {
        Dictionary props = new Properties();
        bc.registerService(HelloService.class.getName(),
                           new HelloServiceImpl(), props);
    }
    public void stop(BundleContext bc) throws Exception {
    }
}
```

**Consuming services**

- Done through an activator, via the bundle context and service activator:

```java
public final class HelloConsumerActivator implements BundleActivator { 
    private ServiceTracker serviceTracker;
    public void start(BundleContext bc) throws Exception { 
        serviceTracker = new ServiceTracker(bc, 
                           HelloService.class.getName(), null); 
        serviceTracker.open();
        HelloService service = 
                           (HelloService) serviceTracker.waitForService(10000);
    } 
    ... 
}
```
Yeah, but...

• We had to write an activator class
• We had to import org.osgi.*
• We had to work with the OSGi API directly
• There’s a lot of boilerplate code

Wouldn’t it be great if...

• ...we could eliminate the boilerplate?
• ...we could declare services for publication and consumption?
Introducing Spring-DM

What Spring-DM offers...

- Declarative service model for OSGi
- The full facilities of the Spring framework
- Simplified OSGi web development
  - with easy Spring MVC integration
- OSGi API...optional
Not the only game in town

- OSGi Declarative Services
- Apache Felix iPOJO
- DynamicJava’s ServiceBindingUtils
- Peaberry

The Spring-DM Extender

- “Allows other bundles to extend the functionality in a specific domain”
- Watches for bundles to be installed in OSGi
- Creates a Spring application context for Spring-enabled bundles
  - By default, looks in META-INF/spring/*.xml
  - Can be configured with Spring-Context: header
- Publishes Spring context as a service
  - Can be disabled with “;public-context:=false”
**Spring-DM Extender**

Adding Spring-DM

- Your code typically doesn’t depend on Spring-DM
- Only need the Spring-DM extender bundle and a few Spring bundles
- Add them all with Pax Construct:

  ```
  % pax-import-bundle -g org.springframework.osgi \
  ?   -a spring-osgi-extender -v 1.2.0 \
  ?   -- -DimportTransitive -DwidenScope
  ```
Spring-DM context XML

- Best kept separate from non-OSGi XML
- Supports OSGi-free testing of beans
- Best used as default namespace

```xml
<beans:beans xmlns="http://www.springframework.org/schema/osgi"
    xmlns:beans="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/osgi/
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
    ...
</beans:beans>
```

Registering services

- Simplest form:
  ```xml
  <service ref="helloBean" interface="com.osgiknowhow.beans.HelloService" />
  ```

- Publish under multiple interfaces:
  ```xml
  <service ref="helloBean">
    <interfaces>
      <beans:value>com.osgiknowhow.beans.HelloService</beans:value>
      <beans:value>com.osgiknowhow.beans.GoodbyeService</beans:value>
    </interfaces>
  </service>
  ```

- With service properties
  ```xml
  <service ref="helloBean" interface="com.osgiknowhow.beans.HelloService" >
    <service-properties>
      <beans:entry key="mode" value="test" />
    </service-properties>
  </service>
  ```
Consuming services

- Simplest form:
  ```xml
  <reference id="helloBean"
           interface="com osgiknowhow beans HelloService" />
  ```
- Optional service reference:
  ```xml
  <reference id="helloBean"
           interface="com osgiknowhow beans HelloService"
           cardinality="0..1" />
  ```
- Specifying a filter:
  ```xml
  <reference id="helloBean"
           interface="com osgiknowhow beans HelloService"
           filter="(mode=testing)" />
  ```

Spring-DM and the web
Web Bundles

- Just WAR files
  - With an OSGi manifest
  - Should be thin...no/few JARs in WEB-INF/lib
  - And Bundle-ClassPath: header
    
    Bundle-ClassPath: .,WEB-INF/classes,WEB-INF/lib/hello.jar

Web Extenders

- Pax Web Extender
  - Parses web.xml file
  - Automatically registers servlets/filters/etc with HttpService

- Spring-DM Web Extender
  - Doesn’t parse web.xml file
  - Hands web bundle over to Tomcat/Jetty to deploy
A tale of two Spring contexts

● Spring–DM extender will create a web context for the web bundle
● Spring MVC will create another web context for the DispatcherServlet
● The controllers in the Spring MVC context won’t be able to see the service references in the DM–created context
● Unless...

```xml
<web-app>
  <context-param>
    <param-name>contextClass</param-name>
    <param-value>
      org.springframework.osgi.web.context.support.OsgiBundleXmlWebApplicationContext
    </param-value>
  </context-param>
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/dude-osgi.xml</param-value>
  </context-param>
  <listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
  </listener>
  <servlet>
    <servlet-name>dude</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
      <param-name>contextClass</param-name>
      <param-value>
        org.springframework.osgi.web.context.support.OsgiBundleXmlWebApplicationContext
      </param-value>
    </init-param>
  </servlet>
</web-app>
```
You’ll need a few more bundles

● You’ll need the Spring-DM web extender and Spring MVC

● Using Pax Construct:

```
pax-import-bundle -g org.springframework.osgi -a spring-osgi-web-extender -v 1.2.0
pax-import-bundle -g org.springframework.osgi -a spring-osgi-web -v 1.2.0
pax-import-bundle -g org.springframework -a spring-web -v 2.5.6
```
OSGi 4.2’s Blueprints

- AKA RFC-124: A component model for OSGi
- Suspiciously similar to Spring-DM
- Spring-DM 2.0.0 is the reference implementation
  - Also available outside of Spring

Blueprint Example

```xml
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:ext="http://geronimo.apache.org/blueprint/xmlns/blueprint-ext/v1.0.0"
  default-activation="lazy">
  <bean id="numberService"
    class="com.habuma.numbers.internal.NumberToEnglishServiceImpl" />
  <service ref="numberService"
    interface="com.habuma.numbers.NumberToEnglishService" />
</blueprint>
```
So what?

What did OSGi buy us?

Each module can be...

- Developed independently
- Tested independently
- Understood independently
- Updated independently
- Swapped out for alternate implementations
Moving to OSGi

Or...what do I do with this WAR file I already have?

Strategy

1. Convert full app into one large bundle
   a. Embed JARs (Use Bundle-ClassPath)
   b. Add Bundle-SymbolicName
2. Peel off dependencies one-by-one
   a. Start with 3rd party libraries
   b. Pull out proprietary modules
Converting Big Apps

Dealing with Legacy JARs

- BND
- Pax Construct:
  - pax-wrap-jar or pax-embed-jar
- Find them in a bundle repository
  - OBR
    - http://www.osgi.org/Repository
  - SpringSource Enterprise Bundle Repository
    - http://www.springsource.com/repository