Tutorial: Spring Dynamic Modules

Adrian Colyer, CTO, SpringSource
Martin Lippert, aquinet agile GmbH
BJ Hargrave, IBM & CTO, OSGi Alliance
Agenda

• What is Spring Dynamic Modules?
• Spring Dynamic Modules in Action
• Server-side Applications
• RCP Applications
• Summary
What is Spring Dynamic Modules?

• Project Objectives
• Introduction to key Spring concepts
• Bundles and module contexts
• Application design
• The extender pattern
• Who's using it?
Spring Dynamic Modules is...

- A open source project in the Spring portfolio
  - led by SpringSource
  - committers from BEA and Oracle
  - many non-code contributions from the community and from the OSGi EEG and CPEG

http://www.springframework.org/osgi
Project Objectives

• Bring the benefits of OSGi:
  ✷ modularity
  ✷ versioning
  ✷ lifecycle support

• To enterprise application development
Design considerations (raw OSGi)

• Platform dynamics
  – services may come and go at any time
  – ServiceTracker

• Asynchronous activation
  – service dependency management

• Testing

• Concurrency and thread management
Project Objectives

- The simplicity and power of Spring... with the dynamic module system of OSGi
- Modules need instantiating, configuring, decorating, assembling, ...
- Need an easy way to manage service references between modules
- Easy unit and integration testing

*Bring the benefits of OSGi to enterprise applications*
Key Spring Concepts

Simple Object

Dependency Injection

AOP

Portable Service Abstractions
The Heart of Spring

• Lightweight container
  – Full stack, simple object based application development

• Works in any environment
  – web-app, ejb, integration test, standalone

• Provides…
  – a powerful object factory that manages the instantiation, configuration, decoration and assembly of business objects
Spring-based development

• View application as a set of components
  - with clear layering
• Each component is a simple object
  - Testable in isolation

• Container manages component configuration and assembly
• Container decorates your components at runtime
Typical application layering

- **Presentation layer**
  - Other remote interfaces
  - Web interface (MVC)

- **Service layer**
  - Service interfaces

- **Data access layer**
  - DAO interfaces
  - DAO implementations

- **Domain objects**

- **RDBMS**
Typical application layering

- Other remote interfaces
- Web interface (MVC)
- Service interfaces
- Service implementations
- DAO interfaces
- DAO implementations
- Domain objects

Spring managed

RDBMS
Spring Framework

• Dependency injection
• Integration with persistence technologies (JDBC, Hibernate)
• Web application support Spring MVC, JSF and Struts
• Enterprise service abstractions
  ◦ Transactions
  ◦ Messaging
• Aspect Oriented Programming support
Without dependency injection

```java
public class TransferServiceImpl implements TransferService {
    private AccountRepository accountRepository;

    public TransferServiceImpl() {
        DataSource ds = (DataSource) ctx.lookup("myAppserverDS");
        accountRepository = new JdbcAccountRepository(ds);
    }
    ...
}
```

Tied to Jdbc implementation
Tied to application server JNDI
Hard to test. Hard to reuse
Dependency Injection

```
public class JdbcAccountRepository implements AccountRepository {
    ...
    
    Implements a service interface
}
```

```
public class TransferServiceImpl implements TransferService {
    private final AccountRepository accountRepository;

    public TransferServiceImpl(AccountRepository ar) {
        this.accountRepository = ar;
    }

    ...
}
```

Depends on service interface; conceals complexity of implementation; allows for swapping out implementation
Spring Blueprint

```xml
<beans>

  <bean id="transferService" class="app.impl.TransferServiceImpl">
    <constructor-arg ref="accountRepository"/>
  </bean>

  <bean id="accountRepository" class="app.impl.JdbcAccountRepository">
    <constructor-arg ref="dataSource"/>
  </bean>

  <bean id="dataSource" class="com.oracle.jdbc.pool.OracleDataSource">
    <property name="URL" value="jdbc:oracle:thin:@localhost:1521:BANK"/>
    <property name="user" value="moneytransfer-app"/>
  </bean>

</beans>
```
Bundles and Module Contexts

• OSGi bundle <=> Spring Application Context
  ♦ we call it a module context
• Module context created when bundle is started
• destroyed when bundle is stopped

• Module components <=> Spring beans
  ♦ instantiated, configured, decorated, assembled by Spring

• Components can be imported / exported from OSGi service registry
Application Design

• Application becomes a set of co-operating bundles
  – vertical decomposition first
  – then horizontal

• Communication via service registry
Application wiring
Spring Dynamic Modules
The Extender pattern

• “The OSGi Extender Model”
  - Peter Kriens, Feb. 2007

• [A]synchronous bundle listener
  - listen to install, update, uninstall events
  - inspect bundle content
  - Take appropriate action on behalf of the bundle

• Spring Dynamic Modules extender bundle:
  - org.springframework.osgi.bundles.extender
  - must be installed and active for module contexts to be created
Spring Dynamic Modules Users

• Oracle
  • building next generation middleware platform on OSGi and Spring DM

• BEA
  • WebLogic Event Server 2.0 built on Spring Dynamic Modules

• Over 1000 subscribers on mailing list

http://groups.google.com/group/spring-osgi
Agenda

• What is Spring Dynamic Modules?
• **Spring Dynamic Modules in Action**
• Server-side Applications
• RCP Applications
• Summary
Spring Dynamic Modules in Action

• Creating a Spring-powered bundle
• Importing and exporting services
• The whiteboard pattern
• Dynamics
• Startup and shutdown
Spring-powered bundles

• Spring module context (app context) per bundle (module)
  – created automatically for you by Spring extender bundle
  – no need to depend on any OSGi APIs

• META-INF/spring/**.xml

• or Spring-Context header in MANIFEST.MF
Demo: Spring-powered bundle

- Published interfaces
- Protected implementations
- Spring configuration files
Getting log output

• Spring uses Jakarta Commons Logging
• Commons logging doesn't behave well under OSGi
  ✷ Use SLF4J binding instead
    ▪ Simple Logging Facade for Java (http://www.slf4j.org/)
• Bundles:
  ✷ jcl104.over.slf4j (static binding of jcl to slf4j)
  ✷ slf4j.api (the slf4j API)
  ✷ slf4j.log4j12 (implementation of slf4j over log4j)
Getting log output

osgi> log4j:WARN No appenders could be found for logger (org.springframework.util.ClassUtils).
log4j:WARN Please initialize the log4j system properly.

• Where to put log4j.properties?
  ♦ which bundle is it that looks for this file?
  ♦ how do we make it visible to that bundle?
Getting log output

• Use a Fragment Bundle
  • “Fragments are bundles that are attached to a host bundle by the Framework.” - OSGi Core Specification, 3.14

Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Logging Configuration Fragment
Bundle-SymbolicName: com.springsource.logging.config
Bundle-Version: 1.0.0
Bundle-Vendor: SpringSource
Fragment-Host: org.springframework.osgi.log4j.osgi;
              bundle-version="1.2.15.SNAPSHOT"
Bundle-RequiredExecutionEnvironment: J2SE-1.5
Testing

• Unit testing is easy...
• Integration testing
  ✷ verify module behaves as expected
  ✷ running *inside* OSGi Service Platform
  ✷ kick-off tests in standard fashion
    ▪ JUnit: IDE, ant, maven, ...

• Spring Dynamic Modules integration test support...
Integration test support

OSGi bundle

Your Integration tests

OSGi Service Platform

OSGi bundle

Your Integration tests
Spring Dynamic Modules in Action

• Creating a Spring-powered bundle
• Importing and exporting services
• The whiteboard pattern
• Dynamics
• Startup and shutdown
Services

• Your application is constructed as a set of bundles, each with their own module context

• How do we reference beans in other modules?
  – use the OSGi Service Registry
    • advertise public services
    • import references to external services
Beans and services
Demo: service import/export

Exporting context:

```xml
<bean id="printService"
   class="com.springsource.osgi.print.internal.PrintServiceImpl"
   init-method="init"
   destroy-method="destroy"/>

<osgi:service ref="printService"
   interface="com.springsource.osgi.print.PrintService"/>
```

Importing context:

```xml
<bean id="printClient"
   class="com.springsource.osgi.print.client.Client"
   init-method="init">
   <property name="printService" ref="printService"/>
</bean>

<osgi:reference id="printService"
   interface="com.springsource.osgi.print.PrintService"/>
```
Exporting a service

• *any* Spring bean can be exported as OSGi service
• offers access to the ServiceRegistration object

```
<bean id="printService"
    class="com.springsource.osgi.print.internal.PrintServiceImpl"
    init-method="init"
    destroy-method="destroy"/>

<osgi:service ref="printService"
    interface="com.springsource.osgi.print.PrintService"/>
```
Importing a service

- locates the best OSGi service that matches the description
- handles the service dynamics internally

```xml
<bean id="printClient"
     class="com.springsource.osgi.print.client.Client"
     init-method="init">
    <property name="printService" ref="printService"/>
</bean>

<osgi:reference id="printService"
                interface="com.springsource.osgi.print.PrintService"/>
```
Controlling Service Exporting

• Which interface(s) should the service be registered under?
  • a single interface, use the interface attribute
  • multiple interfaces, use the nested interfaces element
  • Or... have Spring Dynamic Modules calculated the exported interface set for you automatically.

```xml
<osgi:reference id="printService" auto-export="interfaces"/>
```

• auto-export values are interfaces, class-hierarchy, or all-classes.
Controlling Service Exporting

• Service always has service property
  ✷ org.springframework.osgi.bean.name
  ✷ (set to bean name)
• Specify additional service properties explicitly if needed

```xml
<osgi:service ref="printService"
    interface="com.springsource.osgi.print.PrintService">
    <osgi:service-properties>
        <entry key="aKey" value="someValue"/>
        <entry key="aKey" value-ref="someBeanName"/>
    </osgi:service-properties>
</osgi:service>
```
Controlling Service Importing

• Use filter expressions
  - RFC 1960: A String representation of LDAP Search Filters

```xml
<osgi:reference id="printService"
  interface="com.springsource.osgi.print.PrintService"
  filter="(colour=true)"/>
```

• Special attribute **bean-name** matches on
  org.springframework.osgi.bean.name property
  - condition anded with filter expression if present
• Can specify multiple interfaces using nested **interfaces**
  element.
Spring Dynamic Modules in Action

• Creating a Spring-powered bundle
• Importing and exporting services
• The whiteboard pattern
• Dynamics
• Startup and shutdown
The Whiteboard Pattern

• “Listeners Considered Harmful: The Whiteboard Pattern”
  - OSGi Alliance Technical Whitepaper, 2004

• Lifecycle issues around listener registration
• Solution: whiteboard
  - event source is not registered as a service
  - listeners register as services using well-known interface
  - event source uses a tracker to track listener services
Importing a set of services

```xml
<bean id="printClient"
      class="com.springsource.osgi.print.client.Client"
      init-method="init">
  <property name="printService" ref="printService"/>
</bean>

<osgi:set id="printService"
          interface="com.springsource.osgi.print.PrintService"/>
```

- locates all OSGi services that match the description
- handles the service dynamics internally
- See also: `<osgi:list... />`
Spring Dynamic Modules in Action

• Creating a Spring-powered bundle
• Importing and exporting services
• The whiteboard pattern
• Dynamics
• Startup and shutdown
Dealing with dynamics

A service bundle…

Service implementation locked away

Service interface types exported [with version information]

"Passive" contribution
• types added to type space
• bundles see new version on resolution after install/refresh
Demo: update vs. refresh
Dealing with dynamics

A service bundle...

"Active" contribution
- services published in registry
- bundles see service changes immediately

Private implementation objects

Published services
Service Dynamics

• What happens when a service goes away?
  – osgi:reference cardinality=”0..1”
    • track replacement and retarget proxy when suitable target found
    • ServiceUnavailableException after timeout if invoked
  – osgi:reference cardinality=”1..1”
    • as above, plus
    • unregister any exported services that depend on the unsatisfied reference
Cardinality (single reference)
Registration management

```xml
<osgi:service id="myService" ref="exposedBean"/>

<bean id="exposedBean" class="...">
   <property name="myHelper" ref="helperBean"/>
</bean>

<bean id="helperBean" class="...">
   <property name="fooService" ref="fooService"/>
</bean>

<osgi:reference id="fooService" interface="..."/>
```
Service Dynamics

• What happens when a service goes away?
  – `osgi:set/list cardinality="0..n"`
    • service is removed from the set
    • Iterator contract is honored
  – `osgi:set/list cardinality="1..n"`
    • as above, plus
    • unregister any exported services that depend on the unsatisfied service reference
Cardinality - many
Listening

• You work with a constant reference
  – Proxy / Set / List

• Spring Dynamic Modules manages the target backing service(s) for you

• You can optionally listen to bind / unbind events

• You can optionally listen to register / unregister events
Reference listeners

```xml
<osgi:reference id="printService"
    interface="com.springsource.osgi.print.PrintService">

    <osgi:listener bind-method="onBind"
        unbind-method="onUnbind">
        <beans:bean class="MyCustomListener"/>
    </osgi:listener>

</osgi:reference>

class MyCustomListener {

    public void onBind(PrintService service, Map serviceProperties) {...}
    public void onBind(FastPrintService service, Map serviceProps) {...}
    public void onUnbind(ColorPrintService service, Map props) {...}

}
Registration listeners

```xml
<osgi:service id="printService"
    interface="com.springsource.osgi.print.PrintService">

    <osgi:registration-listener
        registration-method="registered"
        unregistration-method="unregistered"
        ref="printServiceListener"/>

</osgi:service>
```

```java
class MyCustomListener {
    
    public void registered(PrintService service, Map serviceProps) {...}
    
    public void unregistered(PrintService service, Map serviceProps) {...}
}
```
Spring Dynamic Modules in Action

• Creating a Spring-powered bundle
• Importing and exporting services
• The whiteboard pattern
• Dynamics
• Startup and shutdown
Startup

- **Context creation**
  - blocks until all mandatory service references are satisfied
  - simply start your bundles and let Spring Dynamic Modules figure it out
- **Control via Spring-Context manifest header directives**
  - wait-for-dependencies:=[true|false]
  - timeout:=[seconds]
- **E.g.**
  - `Spring-Context: *; wait-for-dependencies:=false`
Shutdown

• Module contexts disposed when bundle is stopped
• Stopping the extender bundle disposes of all module contexts created by it
  ✷ First those bundles that do not export any referenced services (in reverse bundle id order)
  ✷ Cycles broken first by ranking, then by service id
Agenda

• What is Spring Dynamic Modules?
• Spring Dynamic Modules in Action
• Server-side Applications
• RCP Applications
• Summary
Server-side Applications

• Options for using OSGi on the server-side
• Enterprise library "gotchas"
• Context class loader management
• Petclinic application
OSGi as a server platform
Embedded OSGi
Nested OSGi
Enterprise Libraries under OSGi

- class and resource-loading problems
  - class visibility
  - Class.forName
  - context class loader

- Good news: Spring 2.5 is OSGi-ready
  - modules shipped as bundles
  - all class loading behaves correctly under OSGi
Example: Class visibility
Class visibility solutions

- **Dynamic-ImportPackage**
  - a last resort, too broad a scope
  - does not affect module resolution

- **Equinox Buddy Policy**
  - In Hibernate bundle manifest:
    - Eclipse-BuddyPolicy: registered
  - In domain model bundle manifest:
    - Eclipse-RegisterBuddy: org.hibernate
    - Import-Package: org.hibernate

- **Attach a Fragment Bundle**
  - With required Import-Package headers
Class.forName

• Caches the returned class in the initiating class loader
  – native, vm-level cache
• Can cause class loading errors
• Prefer ClassLoader.loadClass
Context Class Loader

• Heavily used in enterprise Java
• Expected to have visibility of application types + classpath
• ContextClassLoader is undefined in OSGi!
  – No notion of “context”; No notion of “application”
• Solutions:
  – Eclipse Equinox: Context Finder
  – Spring Dynamic Modules : CCL management
Context ClassLoader Management

• Context ClassLoader guaranteed to have visibility of bundle classpath when the module context for a bundle is created

• Control CCL on service invocation:
  ✷ client-side (attribute of reference element)
    ▪ context-class-loader="client|service-provider|unmanaged"
  ✷ service-side (attribute of service element)
    ▪ context-class-loader="service-provider|unmanaged"
Web Applications

- OSGi HttpService (Servlet 2.1 - 1998)
  - registerServlets and resources under aliases
  - programmatic configuration

- Equinox Http Registry bundle
  - register servlets and resources using eclipse extension registry

- OPS4J
  - (http://wiki.ops4j.org/confluence/display/ops4j/Pax)
  - Pax Web (Servlet 2.5, based on Jetty)
  - Pax Web Extender – War

- Focus of Spring Dynamic Modules v1.1
Extension Registry

<plugin>

  <extension point="org.eclipse.equinox.http.registry.resources">
    <resource alias="/files" base-name="/web_files"/>
  </extension>

  <extension point="org.eclipse.equinox.http.registry.servlets">
    <servlet alias="/test" class="com.example.servlet.MyServlet"/>
  </extension>

</plugin>
Case Study: Petclinic
Petclinic under OSGi

- Spring Framework 2.5 petclinic sample
- Database: hsqldb
- Persistence: JPA (Toplink Essentials)
- Middle-tier
  - context:load-time weaving
- Web-tier: JSP, Spring-MVC
  - annotation-driven approach
- Web container: Jetty
Bundles

- database bundle
  - starts hsqldb
  - exports DataSource

- application bundle
  - exports Clinic
  - uses JPA, load-time weaving

- web bundle
  - registers DispatcherServlet
Demo: db layer

```xml
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource"
     depends-on="hsqldb">
  <property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
  <property name="url" value="jdbc:hsqldb:hsql://localhost:9001"/>
  <property name="username" value="sa"/>
  <property name="password" value=""/>
</bean>

<!-- expose the data source for other modules to use -->
<osgi:service ref="dataSource" interface="javax.sql.DataSource"/>
```
Demo: middle-tier

<!-- pull in dataSource from db bundle -->
<osgi:reference id="dataSource" interface="javax.sql.DataSource"/>

<!-- JPA EntityManagerFactory -->
<bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"
     p:dataSource-ref="dataSource">
  <property name="jpaVendorAdapter">
    <bean class="org.springframework.orm.jpa.vendor.TopLinkJpaVendorAdapter"
         p:databasePlatform="${jpa.databasePlatform}"
         p:showSql="${jpa.showSql}"/>
  </property>
  <property name="persistenceXmlLocation"
            value="classpath:org/springframework/.../jpa/persistence.xml"/>
</bean>

...
JPA class-visibility

- TopLink entity manager bundle can't see the Petclinic types

- In petclinic bundle:
  - `Eclipse-RegisterBuddy`: `oracle.toplink.essentials`

- In TopLink Essentials bundle:
  - `Eclipse-BuddyPolicy`: registered
Import what you Export

- IncompatibleClassChangeError
  - TopLink Essentials bundles javax.persistence inside its jar
- Version seen by TopLink classes different to version used by Petclinic bundle

- Solution: (in TopLink Bundle)
  - \textbf{Import-Package}: javax.persistence, javax.persistence.spi
Load-time weaving agent

- TopLink needs instrumentation agent

```xml
<context:load-time-weaver/>
```

- `-javaagent:spring-agent.jar`
- Must configure Eclipse to delegate to application classpath first
Demo: web-tier

```xml
<osgi:reference id="clinic"
  interface="org.springframework.samples.petclinic.Clinic"/>


<bean id="servletRegistration"
  class="org.springframework...registration.ServletRegistration"
  init-method="register" destroy-method="unregister">
  <property name="httpService" ref="httpService"/>
  <property name="alias" value="petclinic"/>
  <property name="jspLocation" value="/WEB-INF/jsp"/>
  <property name="resourceAliases">
    <map>
      <entry key="images" value="/WEB-INF/images"/>
      <entry key="styles" value="/WEB-INF/styles"/>
      <entry key="html" value="/WEB-INF/html"/>
      <entry key="docs" value="/WEB-INF/docs"/>
    </map>
  </property>
</bean>
```
Web application context

- Spring Extender
- Petclinic web
- Dispatcher Servlet
- OsgiBundle AppContext
- Web AppContext
- parent
Supporting JSPs

• Register JasperServlet with HttpService
• Bundles:
  – org.eclipse.equinox.jsp.jasper
  – org.apache.jasper
  – org.apache.commons.el
  – java.servlet.jsp
Agenda

• What is Spring Dynamic Modules?
• Spring Dynamic Modules in Action
• Server-side Applications
• RCP Applications
• Summary
Pure RCP Client for a Spring Backend

- Server provides REST/SOAP services, client consumes via HTTP
- Server provides services via RMI, client consumes via RMI
Evaluation

+ Unrestricted usage of Spring on the server
+ Unrestricted usage of RCP on the client

- Different deployment and programming models
  (OSGi bundles on the client, typical WAR/EAR files on the server)
  - Good for highly decoupled systems
  - Difficult for more integrated systems
RCP & Spring on the Client, Spring Backend

- Uses Spring/Remoting for remote communication
- With all the possible variations (RMI, HTTPInvoker, Hessian, Burlap, etc.)
Evaluation

+ Unrestricted usage of Spring on the client and the server
+ Unrestricted usage of RCP on the client
+ Easy remote communication via Spring/Remoting

− Still different deployment and programming models (OSGi bundles on the client, typical WAR/EAR files on the server)
  - Although most likely classes are shared between client and server
Spring & OSGi everywhere

- Equinox/OSGi can be used to implement middle-tiers
  - Same component model on both sides
  - Same extensibility for both sides
- Client and server shares components
Evaluation

+ Full OSGi power on client and server
+ Full Spring power on client and server
+ Homogeneous programming model for client and server
More Spring on the Rich Client

• Dependency injection and all other technology abstractions usable as well
  ❖ Just straight forward using Spring Dynamic Modules

• How to incorporate this with the Extension-Registry?
  ❖ For example, inject dependencies into views and editors?
Alternative 1: Views with dependencies

• Define the View in the Spring context
  ♦ Using Spring for dependency injection

• Define the Extension using an extension factory
  ♦ Which delegates the creation to the Spring context

+ Dependency injection for general extensions
  – Cumbersome manual programming for each extension
Alternative 2: Auto wiring

• Define the View in the Spring context
  ✷ Using Spring for dependency injection

• Add a call to the auto wiring factory from the views constructor

+ Dependency injection for general extensions
  – Still some manual extra code for each extension
Alternative 3: @Configurable

• Define the View in the Spring context
  ♦ Using Spring for dependency injection
• Add the @Configurable annotation to the view implementation

+ Dependency injection for general extensions
+ No additional code necessary
  – Does not work out of the box
  – Adds load-time weaving overhead
Summary
Summary

- Spring Dynamic Modules brings the familiar Spring model to the OSGi platform
- Associates module context with a bundle
- Import and export of services with management of dynamics
- A new approach for constructing enterprise applications
- ... and Rich Client Platform applications
Backup Materials
Versioning

- Packages are imported
  - optionally with version information
- Can have multiple versions of same package concurrently
Try it: versioning

Versioning Demo

Import-Package: org.osgi.framework;version="1.3.0",
com.springsource.printing.lib;version="2.0",
com.springsource.datetime

Printing Lib v2

Export-Package: com.springsource.printing.lib;
version="2.0"

Date/Time

Export-Package: com.springsource.datetime

Import-Package: com.springsource.printing.lib;
version="[1.0.0,2.0.0)"

Printing Lib v1

Export-Package: com.springsource.printing.lib;
version="1.0"
Compendium Services

- supported by the “osgix” namespace
- currently only Configuration Admin service

```xml
<osgix:property-placeholder id="osgi-props"
    persistent-id="com.springsource.osgi.print"/>

<bean id="printService" ... >
    <property name="queueSize" value="${queue.size}"/>
</bean>
```

- Support will be extended in future releases