Modular Java Applications with Spring, dm Server and OSGi
Topics in this session

• Introduction
• OSGi basics
• OSGi & Spring
• Enterprise OSGi
• Modularization Best Practices
• dm Server 2
• Summary
Introduction

- OSGi: a dynamic module system for Java
- Modular:
  - Bundles, JAR files with meta-data
  - Strict visibility for types, based on packages
  - Versioning of both packages and bundles
- Dynamic:
  - Add, remove, start, stop bundles at runtime
  - Use shared services to share objects, not just types
History of OSGi

• Open Specification managed by OSGi Alliance
  – Founded in March 1999

• Based on the realized need for light weight dynamic platform
  – Initially targeted network and embedded devices
  – Since 2006, server side adoption

• Member companies
  – IBM, SpringSource, Motorola, Oracle, Tibco etc. . .
Topics in this session

- Introduction
- **OSGi basics**
- OSGi & Spring
- Enterprise OSGi
- Modularization Best Practices
- Summary
Bundle MANIFEST.MF

- Bundles use JAR Manifest for meta-data

```
Manifest-Version: 1.0
Bundle-Version: 1.0.0
Bundle-Name: My First Bundle
Bundle-ManifestVersion: 2
Bundle-SymbolicName: my.first.bundle
...
```

- By default, such a bundle is a black box
  - Its types are invisible to other bundles
  - It can't see any types besides its own
Inter-bundle Visibility: Exports

- To make types available to other bundles, *export* their packages

  **Export-Package**: my.first.bundle.api

- Can also *version* the package(s)

  **Export-Package**: my.first.bundle.api;version=1.0.0, my.first.bundle.util;version=1.2.3
Exporting Best Practices

• **Best practice:** separate interfaces from implementations
  – Put in different packages
  – Only export public API, hide internal details
  – Expose implementations as services

• **Best Practice:** apply versions to your packages
  – Allows multiple versions in the same runtime
  – Clients can pick the version they need
Inter-bundle Visibility: Imports

• To access types from other bundles, *import* their packages

Bundle-SymbolicName: some.client.bundle
Import-Package: my.first.bundle.api

– Can also specify version range

Import-Package: my.first.bundle.api;version="[1.0.0, 2.0.0)"

– Single version means 'at least'
– [] for inclusive, () for exclusive boundaries

• Best practice: pick good version range
– What versions will work? Too narrow or wide?
OSGi Visibility

• Adds true encapsulation and versioning to your applications
  – Preserves modularity at runtime
• No longer restricted to a single, linear classpath
  – Each bundle gets its own ClassLoader
• All managed by the OSGi container
OSGi Containers

- Provide the runtime for OSGi bundles
- Small core with additional services
  - Typically very lightweight
- Well-known OSS implementations:
  - Equinox
  - Apache Felix
  - Knopflerfish
OSGi Dynamics

- Bundles can be installed into container
- Have a managed lifecycle
  - Installed (just present, missing dependencies)
  - Resolved (stopped, all dependencies satisfied)
  - Starting
  - Started (services now also available)
  - Stopping
  - Uninstalled (gone after restart or refresh)
• Bundles can receive callbacks when started or stopped

```java
public class MyActivator implements BundleActivator {
    public void start(BundleContext context) throws Exception {
        // ...
    }
    public void stop(BundleContext context) throws Exception {
        // ...
    }
}
```

• Register in manifest

```
Bundle-Activator: some.client.bundle.MyActivator
```
OSGi BundleContext

• BundleContext is API of OSGi runtime
• Work with bundles and services
  – (un)install, start/stop bundles
  – Register services and obtain references
• Register listeners to be notified of interesting events
DEMO

Using plain OSGi bundles to share types and services
Topics in this session

- Introduction
- OSGi basics
- **OSGi & Spring**
- Enterprise OSGi
- Modularization Best Practices
- dm Server 2
- Summary
OSGi & Spring

• OSGi provides nice runtime
  – but lacks component model
• Spring provides component model
  – but does not define the runtime
• Spring Dynamic Modules marries the two
  – Use familiar Spring programming model in OSGi!
• (or Spring-DM for short)
• Removes most OSGi-dependencies from your code
  – **Best Practice:** Proper Separation of Concerns
• Creates ApplicationContext per bundle
• Declarative service management
• And much more
• Exposes Spring beans as services
  – Full control over interface, properties, etc.
• Creates proxies for service references
  – No more manual management of service dynamics!
  – Saves lots of plumbing code in a typical Spring fashion
Spring-DM Configuration

- Spring config files go in META-INF/spring
  - Will be picked up automatically
- osgi: namespace for service export/ref

```xml
<bean id="myService" class="some.bundle.internal.MyServiceImpl">
  <property name="serviceDependency" ref="serviceDependency"/>
</bean>

<!-- Creates dynamic proxy for OSGi service with given interface -->
<osgi:reference id="serviceDependency" interface="other.bundle.ServiceDependency"/>

<!-- Exposes our Spring bean as OSGi service under its interfaces -->
<osgi:service ref="myService" interface="some.bundle.MyService"/>
```
Spring-DM Configuration

- **Best Practice:** Separate normal Spring config files from Spring-DM config files
  - Test or even reuse of modules without OSGi
  - Esp. if ids of beans backing services are the same as `<osgi:reference>` ids

Bundle A, file module-context.xml:

```xml
<bean id="myService"
     class="samples.internal.MyServiceImpl"/>
```

Bundle A, file osgi-context.xml:

```xml
<osgi:service ref="myService"
              interface="samples.MyService"/>
```

Bundle B, file module-context.xml:

```xml
<bean id="myClient" class="...">
    <constructor-arg ref="myService"/>
</bean>
```

Bundle B, file osgi-context.xml:

```xml
<osgi:reference id="myService"
                interface="samples.MyService"/>
```
DEMO

Using Spring Dynamic Modules to share types and services
Topics in this session

- Introduction
- OSGi basics
- OSGi & Spring
- **Enterprise OSGi**
- Modularization Best Practices
- dm Server 2
- Summary
Enterprise OSGi

• OSGi features are appealing to Enterprise Java developers as well
  – True modules instead of monolithic deployments
  – True dynamics that don't require constant restarts
• Most applications servers already use OSGi internally
Enterprise OSGi Issues

• Pure OSGi doesn't mix with Enterprise Java very well
  – Incompatible classloading models
  – OSGi has hardly any web support
  – Bunch of bundles is not a good deployment model
  – Enterprise Libraries not available as bundles

• New products and standards are emerging to address this
Enterprise OSGi Products

• SpringSource dm Server
  – Pioneered OSGi in an Enterprise Java setting
  – Web Support, Thread Context Classloader mgmt, PAR deployment format, Bundle provisioning, ...

• Paremus Service Fabric
  – SCA Support
  – Advanced clustering / cloud capabilities

• Various Open Source Projects
  – Apache Aries
  – OPS4J PAX has several relevant projects
• Enterprise Expert Group produces new specifications
  – RFC 66: Web Support (RI: dm Server 2.0)
  – RFC 112: Bundle Repository
  – RFC 119: Distributed OSGi
  – RFC 124: Blueprints (RI: Spring-DM 2.0)
  – RFC 139: JMX interface for OSGi
  – RFC 142: JNDI integration

• At the same time, much innovation is happening
Enterprise OSGi

• **Best Practice:**
  Enterprise OSGi is harder than you think, don't build your own platform
  – Getting e.g. JPA libraries to work reliably is very challenging
  – Think about runtime management as well

• **Check your options and choose for yourself based on your requirements**
  – Obviously we prefer dm Server
DEMO

Developing a multi-bundle web application with SpringSource dm Server
Topics in this session

- Introduction
- OSGi basics
- OSGi & Spring
- Enterprise OSGi
- **Modularization Best Practices**
- dm Server 2
- Summary
Best Practices

• So much for the tech talk, but how do you apply this?
• Some best practices were given already
• Here follows some more high-level advice on how to design your modules
How To Modularize

• How to split up your application in bundles is not an easy question to answer
• Question is really how to partition and what granularity to use
Partitioning can be done in different ways:

- **Vertical**: *functional* partitioning
  - For example orders, warehouse, billing and CRM

- **Horizontal**: *technical* partitioning
  - Web, services, repositories, infrastructure

- A combination of the two
Vertical Partitioning

- Bundles represent functional modules
- Preferred approach for big enough applications
  - Single module assignable to a team of developers
  - Encapsulates internals like repositories
    - OrderRepository only needed in 'order' module
  - Minimizes module's “surface area”
    - Only needs to expose its business interfaces
- Might not work well for small applications
  - Might not need partitioning in the first place
Horizontal Partitioning

• Bundles represent architectural layers
• Natural approach to many developers
  – Tend to think of layers as modules already
• Allows for replacing layers easily
  – For testing or during early development
  – Deploy stubbed repository bundle without changing services module
• Typically means more maintenance
  – Use cases spread across multiple bundles
  – So changes often span bundle boundaries
Web Modules

- Web Resources like JSPs cannot be split across multiple bundles in dm Server 1.0
  - Not for single ServletContext / HttpSession at least
- Must use single WAR / web module
  - Even when using vertical partitioning!
- dm Server 2.0 will offer slices support
  - Allowing for truly modular web applications
Infrastructure Bundles

• Most applications use shared infrastructure across functional, vertical slices
  – Same DataSource, transaction manager, JMS ConnectionFactory, etc.

• Creating infrastructure bundle(s) makes sense
  – Even when using vertical partitioning: there's no JNDI registry with globally defined resources!
  – Simply expose resources as OSGi services
  – By application developers or operations team
Bundle Granularity (1)

- How much stuff goes in one bundle
- Use same rules as for object orientation
  - Bundles need to have a clear responsibility
  - High cohesion within a bundle
  - Loose coupling between bundles
- Works well with vertical partitioning
  - Horizontal tends to increase dependencies between bundles
Bundle Granularity (2)

• Easy to make modules too fine-grained
  – Often seen in samples and labs
    • To show how OSGi works
  – Doesn't necessarily represent best practice!
  – Better to extract extra bundle later if desired

• Typically shouldn't create bundle if non-OSGi application wouldn't have dedicated jar for the same code
Topics in this session

• Introduction
• OSGi basics
• OSGi & Spring
• Enterprise OSGi
• Modularization Best Practices
• **dm Server 2**
• Summary
See “dm Server 2” Presentation
Topics in this session

- Introduction
- OSGi basics
- OSGi & Spring
- Enterprise OSGi
- Modularization Best Practices
- dm Server 2

Summary
Summary

• OSGi brings true modularity and dynamics to your applications
• Many potential benefits, but not always easy to gain these
• Enterprise OSGi is an upcoming area of great interest to many developers
• New products and standards new emerging