Hints and Tips for Modularizing Existing Enterprise Applications

Graham Charters
Agenda

- Motivation for a Modular Enterprise
- Approaches to OSGi Adoption
- A Staged Approach in Detail
- Summary
Modularity Issues in Java EE

Java EE modularity is inadequate

**Across apps - each** archive typically contains *all* required libraries
- Common libraries/frameworks get installed with each application
- Multiple copies of libraries in memory

**Within apps - 3rd party libraries consume other 3rd party libraries leading to conflicting versions on the application classpath.**
Enterprise OSGi

Specifications are growing coverage

- First specification released 2010 added:
  - Web Applications (WAR -> WAB)
  - JNDI (including OSGi Service scheme)
  - Transactions
  - Blueprint (inspired by Spring)
  - Remote Services

- Second specification released 2012, added:
  - Subsystems (standard application model)
  - Repository
  - Resolver

- Work in the pipeline
  - EJB
  - CDI
  - Blueprint transactions
  - Blueprint enhancements

- Platforms are plugging the gaps: Apache Aries, Eclipse Virgo, Apache Geronimo, WebSphere Application Server, JBoss AS, Glassfish, *your mileage may vary...*
A Staged Approach

- Lots of good reasons to improve modularity but don’t have to achieve all in one go
  - “Big bang” =often=> “damp squib”
- Incremental successes are successes that help build and sustain momentum
- A bad practice may still be a better practice (a means to an end)

How?
Useful Tools

- Many tools help with OSGi development, including bundle generation from simple configuration
  - Rational Application Developer
  - IBM WebSphere Application Server V8.5 Developer Tools V8.5
  - BND
  - BNDTools
  - Maven-bundle-plugin
  - BND Ant Task
  - ...

- Example taken from modularizing the Apache Geronimo DayTrader application.

- DayTrader is built using maven and so maven-bundle-plugin snippets are used throughout.
Strategies

- A number of possible approaches:
  - Hargrave & Kriens: “Single Bundle”
  - Feng & Charters: “One to One”
  - Nottingham & Charters: “Fragments” (a “work in progress”)

© IBM Corporation 2012
Hargrave & Kriens


- Approach Summary:
  1. Single bundle project with all dependent jars on Bundle-Classpath
  2. Over time, pull out dependent jars as bundles

- Focuses on Java SE:
  - Does not consider Java EE classloaders
  - Does not consider Java EE component models

- Surfaces OSGi slower, but delivers incremental benefits
Feng & Charters


- Approach Summary:
  1. Convert each build module directly to an OSGi Bundle
  2. Analyze and refactor

- Focused on Java SE
  - No consideration for Java EE classloaders
  - No consideration for Java EE component models

- Motivated by need to deliver assembly subsets

- Surfaces OSGi early, but longer lead-time to first success
Nottingham & Charters

- EclipseCon 2012: this deck
- Approach Summary:
  1. Replicate existing classloading in OSGi using bundle fragments
  2. Incrementally separate out individual bundles
  3. Adopt OSGi best practices
- Considers Java EE classloading and component models
- Surfaces OSGi early, and incrementally - a staged approach
Stage 1: Understanding the starting point

- Java EE prescribes a hierarchical classloading model
- Assuming “parent first” and “multiple”:
  - Each Application has own classloader
  - Each WAR has own class loader
  - WAR has visibility to Application classes
  - WAR prefers Application classs,
    Application prefers System classes, etc...
- OSGi modularity enforced through class visibility using classloaders
- Migration strategies need to consider the impact of this change
  - e.g. replicate visibility relationships of existing application in OSGi
Replicating Java EE classloading: Classloaders

- Preserve Application and WAR roles
- Application -> Application Host Bundle
  - Add application modules to fragments of host bundle
- Web App Archive -> Web Application Bundle
  - Add WEB-INF/classes to Bundle- Classpath
  - Extract WEB-INF/lib jars and add as fragments of Web Application Bundle
- We now have two classloaders just as we did in Java EE 😊
- We also have full visibility of the modules
Maven-bundle-plugin - Application Host Bundle Example

... <parent> ...
</parent>

<packaging>bundle</packaging>

<groupId>org.apache.geronimo.daytrader.modules</groupId>
<artifactId>app-host</artifactId>
<version>0.1</version>
<name>DayTrader :: Modules - Application</name>
<description>DayTrader Application Module</description>

<dependencies>
...
</dependencies>

<build>
<plugins>
<plugin>
<groupId>org.apache.felix</groupId>
<artifactId>maven-bundle-plugin</artifactId>
<configuration>
<instructions>
...
</instructions>
</configuration>
</plugin>
</plugins>
</build>
</project>

- bundle artefact type to be built
- plugin uses this as default bundle metadata
- the dependencies to build against
- extra bundle configuration (e.g. exports, imports, etc)
Maven-bundle-plugin - Fragment Bundle Example

<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
    <instructions>
      <Fragment-Host>org.apache.geronimo.daytrader.modules.daytrader-app-host</Fragment-Host>
    </instructions>
  </configuration>
</plugin>

Fragment-Host: org.apache.geronimo.daytrader.modules.daytrader-app-host
Replicating Java EE classloading: Delegation

- App and wab can’t see the container apis from the System Classloader
- Wab can’t see app classes
- Solution:
  - Add package imports for container packages to app and wab
  - Add package exports to app-host fragments required by wab
  - Add require bundle from wab to app

- Note: delegation semantics are not identical to Java EE
  - WAS provides all imported packages (close to “parent-first”)
  - Bundle provides everything else
  - Now have control over what wab & app-host see from the System classloader
Determining the Container APIs (System Classloader)

**Warning:** This step will vary depending on target server

- `<was_install>/dev` contains the APIs in `was_public.jar` and `JavaEE/jee.jar`
- Take Export-Package entries and add as Import-Package entries on app and wab bundles
- If using BND-based tools then can wildcard values (e.g. `com.ibm.*`, `javax.*`)
- `<was_install>/dev` includes a pom for WAS API for maven development (available since v8.0)
- Applications should not rely on packages not defined in `<was_install>/dev`
maven-bundle-plugin: Container delegation

mvn install:install-file -DgroupId=com.ibm.websphere.appserver -DartifactId=was_public -Dversion=8.5 -Dpackaging=jar -Dfile=was_public-8.5.0.jar

maven dependency on WAS API
<dependency>
    <groupId>com.ibm.websphere.appserver</groupId>
    <artifactId>was_public</artifactId>
    <version>8.5</version>
</dependency>

maven-bundle-plugin (pom.xml)

<Import-Package>
    <!-- from was_public.jar -->
    com.ibm.*,
    ...
    <!-- from jdk & jee.jar -->
    javax.*
</Import-Package>

daytrader-app.jar META-INF/MANIFEST.MF

Import-Package:
com.ibm.websphere.ActivitySession; version="1.1.0",
...
javax.xml.rpc.server; version="1.1.0",
javax.xml.rpc.soap; version="1.1.0"
Maven-bundle-plugin: WAB to App Delegation

- Fragment exports (daytrader package naming allowed wildcarding)

```xml
<instructions>
  <Fragment-Host>org.apache.geronimo.daytrader.modules.daytrader-app-host</Fragment-Host>
  <Export-Package>org.apache.geronimo.samples.daytrader.core*</Export-Package>
</instructions>
```

- Wab Require-Bundle

```xml
<instructions>
  <Require-Bundle>org.apache.geronimo.daytrader.modules.daytrader-app-host</Require-Bundle>
</instructions>
```
A Note on Require-Bundle

- **Uses**
  - Merge split package
  - When there’ll only ever be one provider (e.g. extension points)

- **Considered bad practice because:**
  - The required bundle can change what it provides - brittle in the face of refactoring
Extenders: Enabling Component Models

- Web Components, EJBs, Persistence Contexts, Blueprint Beans all handled by extenders
- Extenders look inside bundles for code/metadata to process
- Extenders typically require a bundle to ‘opt in’ using a manifest header
  – host must opt-in
- JPA entities not permitted to come from fragments: “…any entity classes must originate in the bundle’s JAR, it cannot come from a fragment.”

```xml
<!-- maven-bundle-plugin configuration snippet--><instructions>

  <!-- Opt in for EJB Extender -->
  <Export-EJB>&lt;&lt;EMPTY&gt;&gt;</Export-EJB>
</instructions>

<!-- maven-bundle-plugin configuration snippet--><instructions>
  <!-- Opt in for Web Extender -->
  <Web-ContextPath>/daytrader</Web-ContextPath>
</instructions>
```

Tip: &lt;&lt;EMPTY&gt;&gt; allows empty header in BND (needs a recent version of maven-bundle-plugin)
OSGi Application

**Warning:** this step will vary depending on target server

- Servers typically enable a deployment artefact and build tools for an application
  - Example was targeting WebSphere, so used a .eba (a zip file with an application manifest)
    - RAD/WDT tools make development simple
    - Ant zip task and eba-maven-plugin (Apache Aries) can also be used in builds

```xml
<plugin>
  <groupId>org.apache.aries</groupId>
  <artifactId>eba-maven-plugin</artifactId>
  <version>0.4-SNAPSHOT</version>
  <extensions>true</extensions>
  <configuration>
    <generateManifest>true</generateManifest>
    <instructions>
      <Application-SymbolicName>${project.artifactId}</Application-SymbolicName>
    </instructions>
  </configuration>
</plugin>
```

**Application-Name:** DayTrader Application
**Application-SymbolicName:** org.apache.aries.samples.daytrader
**Application-ManifestVersion:** 1
**Application-Version:** 0.1
**Manifest-Version:** 1.0
**Application-Content:**
- daytrader.app; version="[0.1,1.0)",
- daytrader.wab; version="[0.1,1.0)"

```
© IBM Corporation 2012
```
Stage 2: Factoring out Bundles

- Now the application’s running in OSGi we can start to split out the fragments as independent, re-usable bundles

- Strategy:
  - Do one fragment at a time
  - Start with the leaf dependencies
    - Wab bundle contents first, then the app bundle
    - Project build dependencies help with identification
    - Third-party libraries
    - “shared libraries” (if runtime supports this concept)
  - Detach from host and calculate the package imports & exports

```plaintext
Fragment-Host: wab

Import-Package: daytrader.web
Export-Package: daytrader.web
```
Split Package Strategies

- A split package is a package exported by more than one provider where the contents differ
  - Sub-set/superset relationship - e.g. javax.transactions
  - Intersecting or non-intersecting subsets
  - Not different releases

- Split packages hint at poor modularity or naming

- Consumers often need access to all providers, but OSGi Import-Package will only wire to one

- Solution:
  - Combine jars with split packages into a single bundle
    - Multiple jars on single Bundle-Classpath or
    - Fragment Bundles attached to a single host

- This approach is tactical and is addressed in Stage 3
Converting Jars to Bundles

Using RAD/WDT?
- Convert Java projects to OSGi Bundle Projects (also works for EJB and WAR)
- Or import “Java Archive into OSGi Bundle”
  - Wraps Jar in a Bundle

Using maven or ant?
- Modify build to produce bundles (e.g. maven-bundle-plugin or bnd in ant task)
- Use wild-cards to generate initial imports/exports then refine...

```xml
<packaging>bundle</packaging>

<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
    <instructions>
      ...
    </instructions>
  </configuration>
</plugin>
```
Third-party Libraries

- Applications often use third-party libraries, such as open source
  - DayTrader uses commons.logging
- Many now come with OSGi metadata
- Most others have been bundlized by repository projects
  - Eclipse Orbit - bundlized, pedigree reviewed for Eclipse projects
  - Apache Felix Commons
  - SpringSource Enterprise Bundle Repository
- or by other OSGi-based projects
  - Apache Tuscany
  - Apache ServiceMix
  - Ops4J Pax *
- Use these if available
- But what if you can’t find a bundlized library...?
Bundlizing third-party libraries

- Use tools to calculate manifest
  - RAD/WDT - Import -> Java Archive into an OSGi Bundle
  - BND
  - Refine Imports/Exports

- If library required to work inside and outside OSGi?
  - inject OSGi manifest into existing jar
  - Doesn’t work for signed Jars
  - Can’t aggregate multiple Jars

- If library only required to work in OSGi
  - Wrap jar in bundle (add jar to Bundle-Classpath)
  - Works for signed jars
  - Can aggregate multiple jars, if necessary
The story so far...

- The first two stages have:
  - Taken Java EE application and run in OSGi
  - Decomposed single deployment artefact into multiple bundles

- Now have the potential to:
  - Understand the application architecture; modules used and their relationships (OSGi Application Console)
  - Remove duplicate deployment binaries (single bundle in repository)
  - Remove duplicate runtime binaries (share bundle in server)
  - Reduce operational costs and errors - deploy shared binaries based on application needs
  - Determine which binaries are in use (e.g. to apply critical fix)

- But there’s more...
Stage 3: Incrementally Adopt Best Practices

- Stage 2 documents the ‘as-is’ architecture, warts-‘n’-all
- OSGi Best Practices show how to make the most of OSGi
  - Hopefully you attended Emily Jiang’s session
- Adopting best practices leads to:
  - High cohesion - bundles with clear and distinct roles in the architecture
  - Loose coupling - flexible, but necessary, dependencies
- …which all leads to greater agility, flexibility and re-use
  - Development teams can understand and explain the architecture and are no longer afraid to change the code
  - Applications can evolve and new application can be created at a pace that enables, rather than inhibits, the business
Summary

- Moving enterprise applications to OSGi helps:
  - increase understanding of application architecture
  - increase module cohesiveness
  - reduce module coupling

- Leads to greater application agility through re-use and understanding of impact of change

- A staged approach to adoption delivers value sooner and in-line with investment

- With the wealth of tools, runtimes, and existing assets, it’s never been easier to adopt OSGi

- What OSGi could do to better:
  - Enable ‘fragments’ to define everything except a classloader
  - Enable JPA entities in fragments
Questions
Trademarks

- Eclipse and the Eclipse logo are trademarks of Eclipse Foundation, Inc.

- Apache, Apache Felix, Felix, the Apache Felix project logo, Apache Tuscany and the Apache Tuscany project logo are trademarks of The Apache Software Foundation.

- Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

- IBM and the IBM logo are trademarks of International Business Machines Corporation, registered in many jurisdictions.

- Other marks may be trademarks or registered trademarks of their respective owners.