JavaOne℠

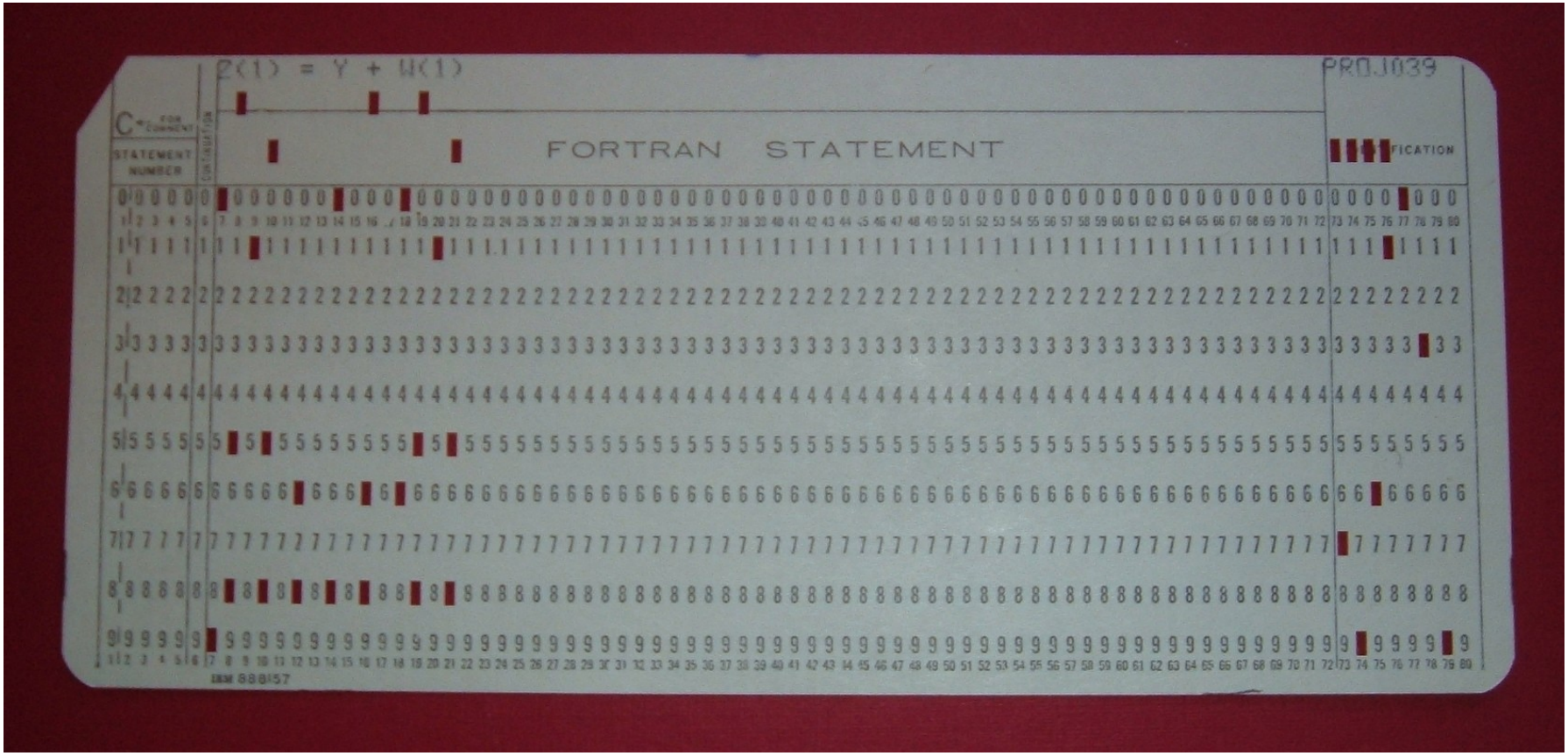
OSGi on the Web: Here's How

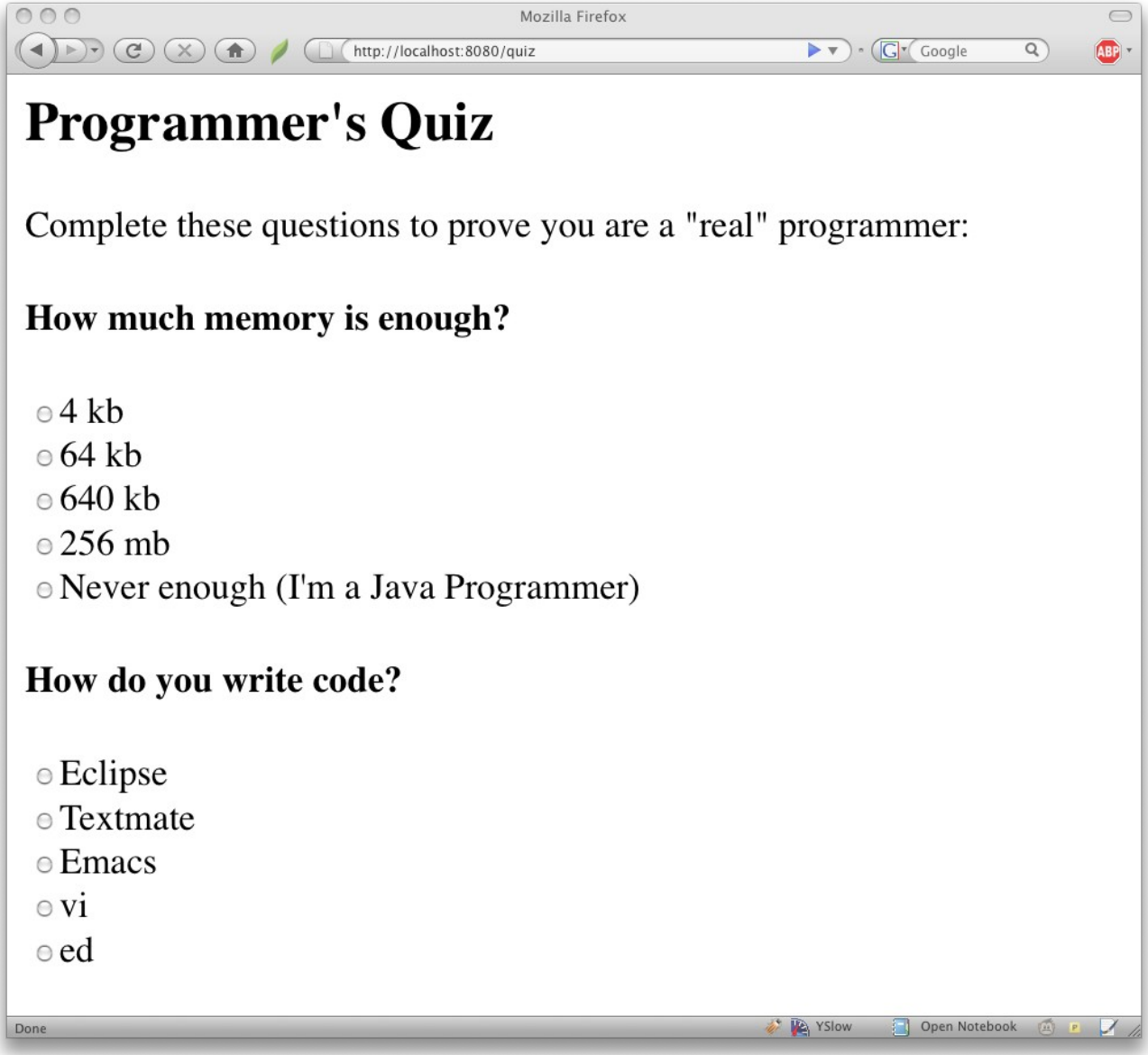
Don Brown

Atlassian

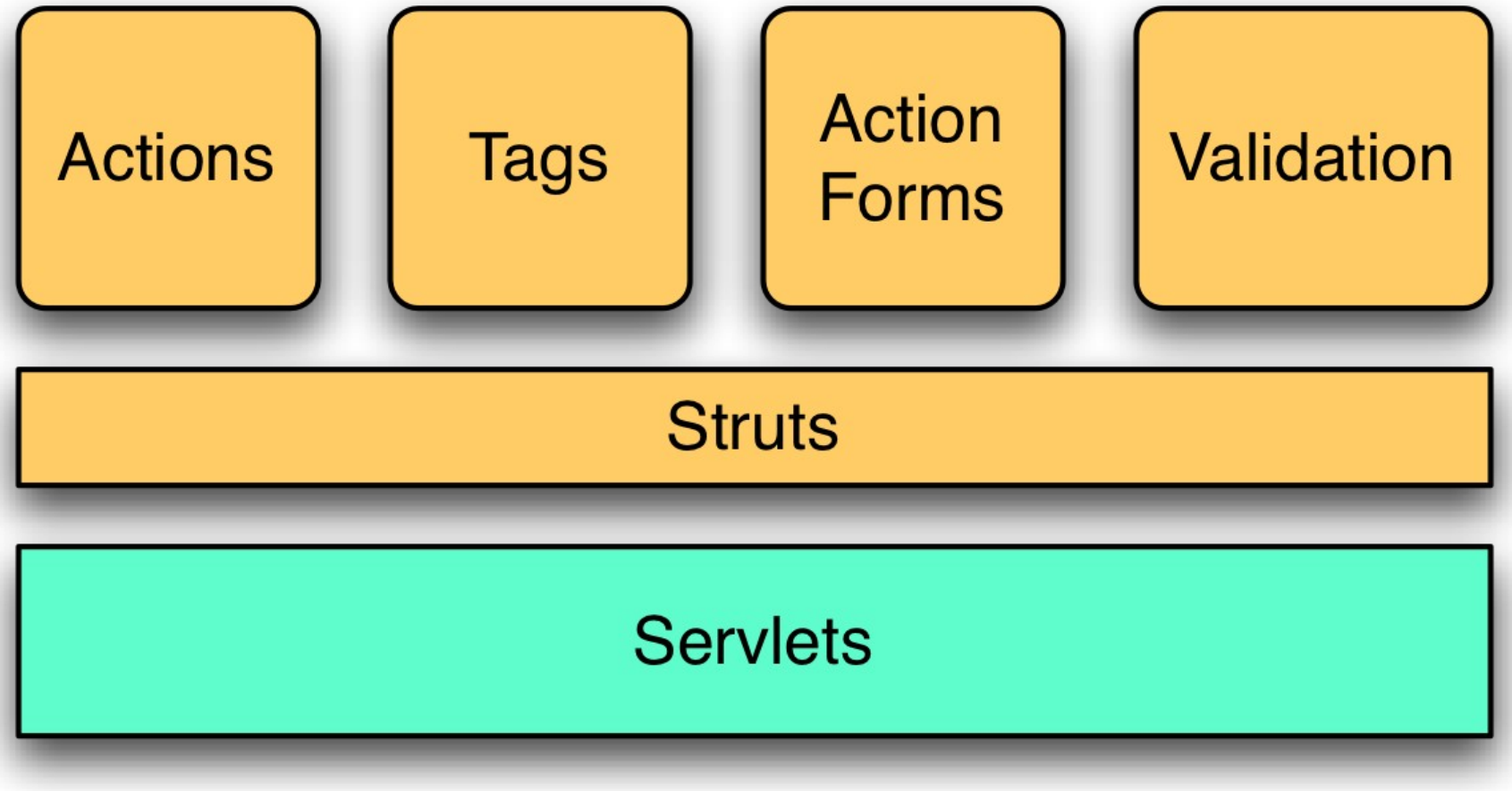

IBM
IBM
IBM


TIMEX sinclair 1000
TIMEX® Module 16K RAM Module
TIMEX sinclair 1016
TIMEX User Manual

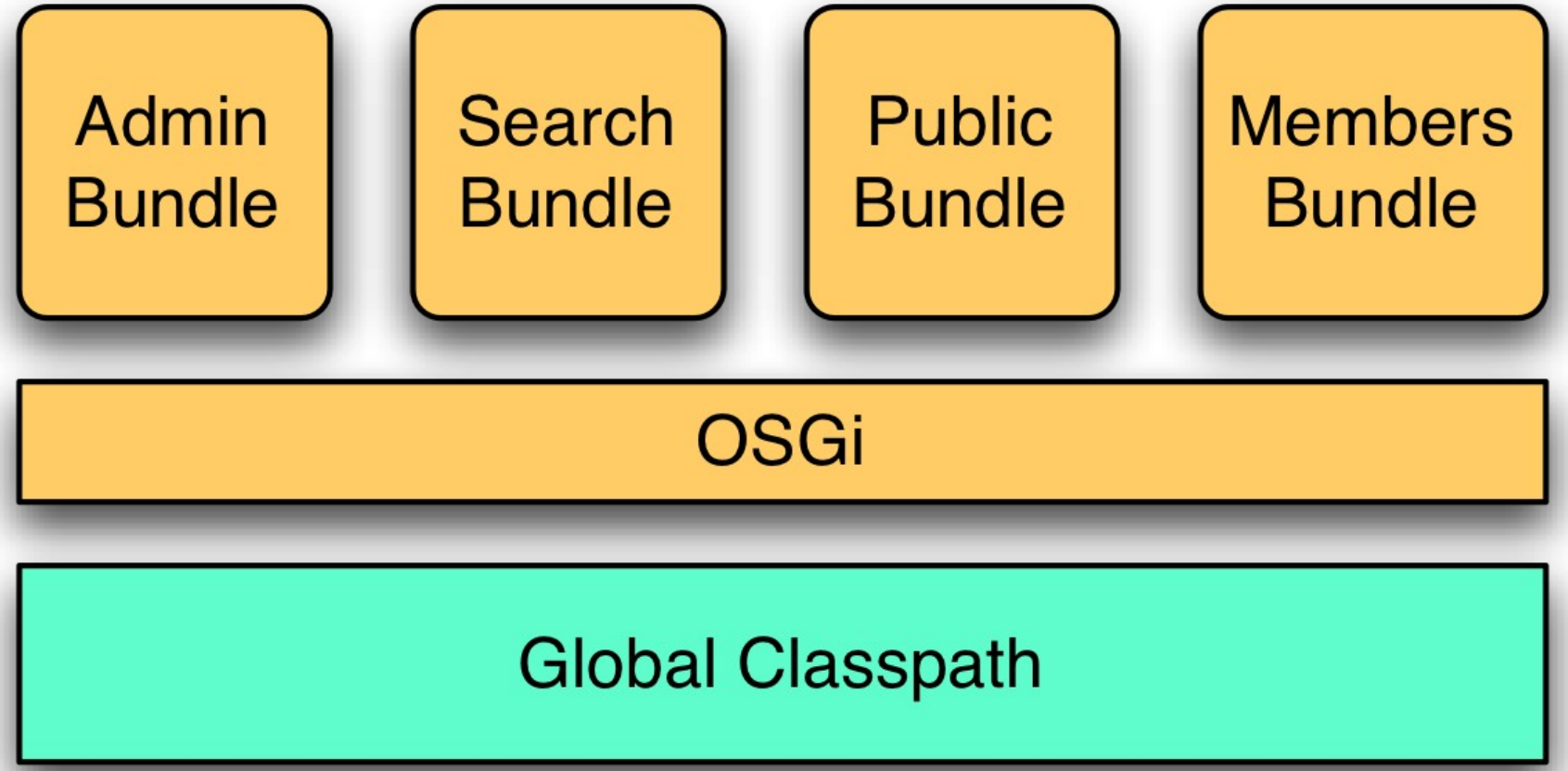

Z(1) = Y + W(1)
PROJ039
FORTRAN STATEMENT
STATEMENT NUMBER
CONTINUATION
IBM 888157


Mozilla Firefox
http://localhost:8080/quiz
Google
Programmer's Quiz
Complete these questions to prove you are a "real" programmer:
How much memory is enough?
4 kb
64 kb
640 kb
256 mb
Never enough (I'm a Java Programmer)
How do you write code?
Eclipse
Textmate
Emacs
vi
ed
Done
YSlow
Open Notebook

Raising the abstraction with Struts



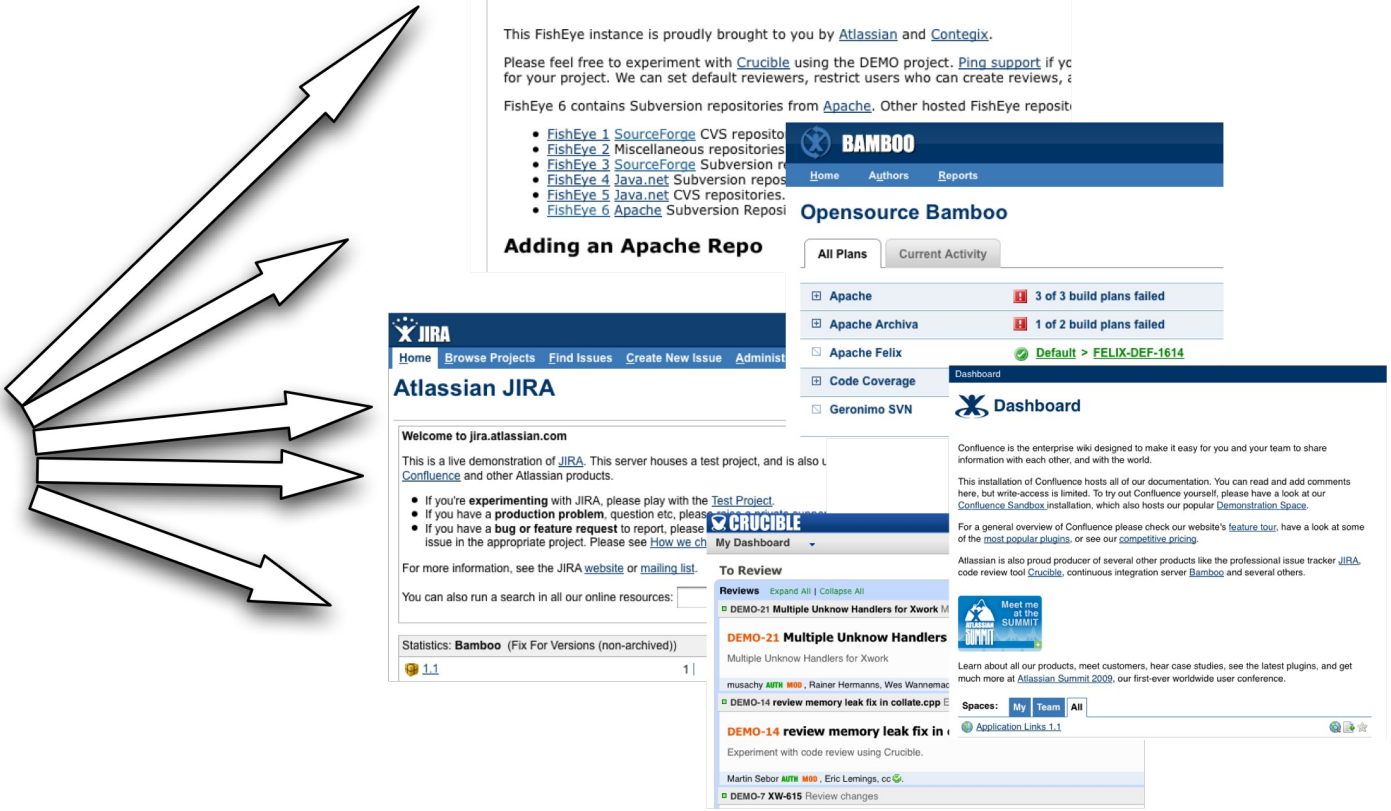
Raising the abstraction with OSGi



Why we needed OSGi

One feature * five products =

Dashboard



Multiple teams across the globe

Gdańsk, Poland

San Francisco, USA

Kuala Lumpur, Malaysia

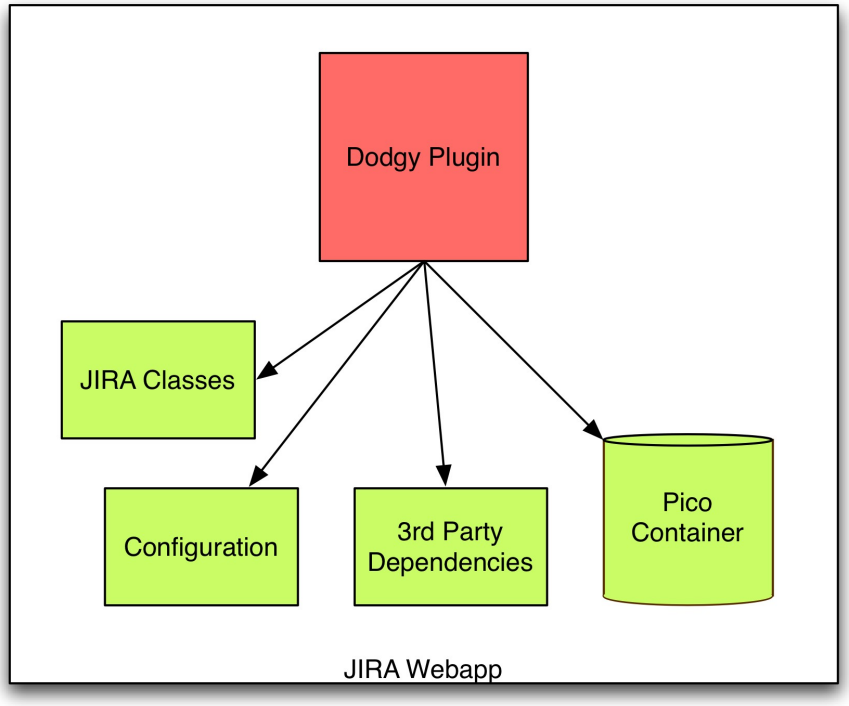
Sydney, Australia

Plugin development slow

- Write plugin code
- Build plugin
- Copy plugin to WEB-INF/lib
- Start app
- Discover bug
- Wash, rinse, repeat

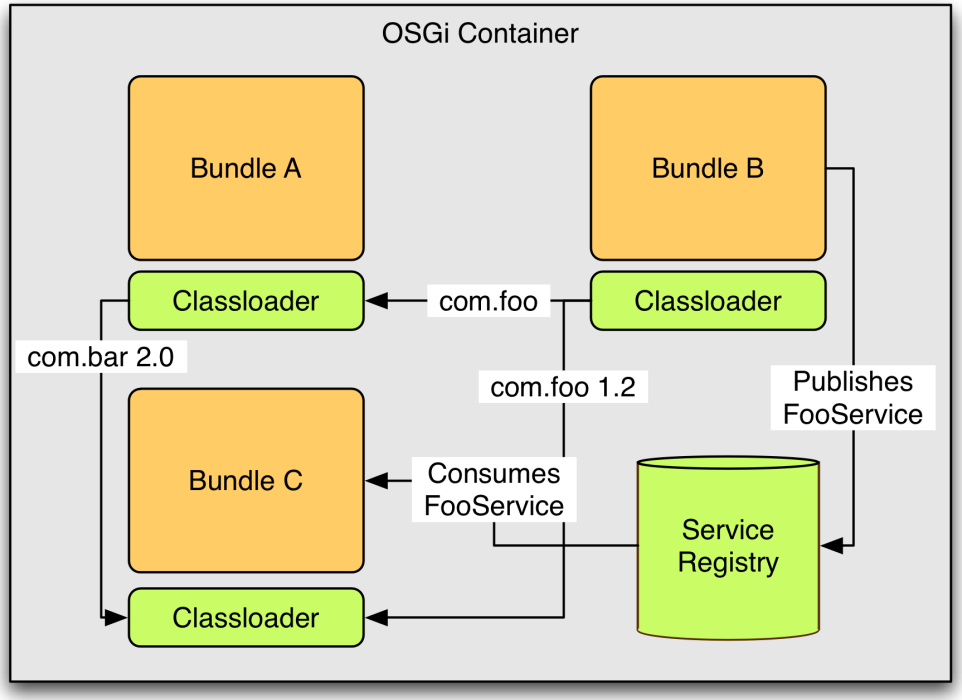
Plugins break on product upgrade

- Plugins have unrestricted access to application classes, objects, and configuration
- Broken plugins after a product upgrade make us look bad



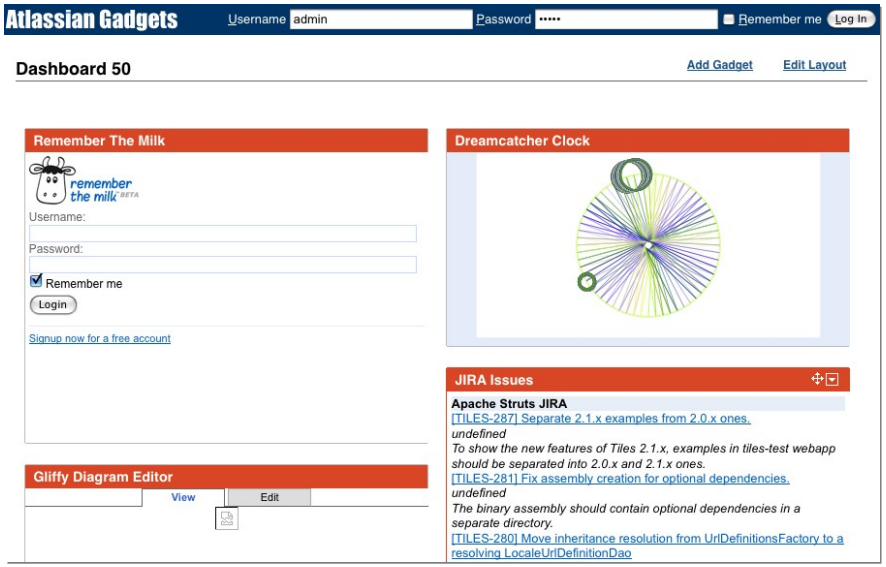
OSGi in one slide

- Bundles contain code, configuration, manifest metadata
- Runtime dependencies at Java package, service, and bundle levels
- Supports multiple versions of code
- Can share dynamic service objects
- Lifecycle: install, resolve, active, uninstall



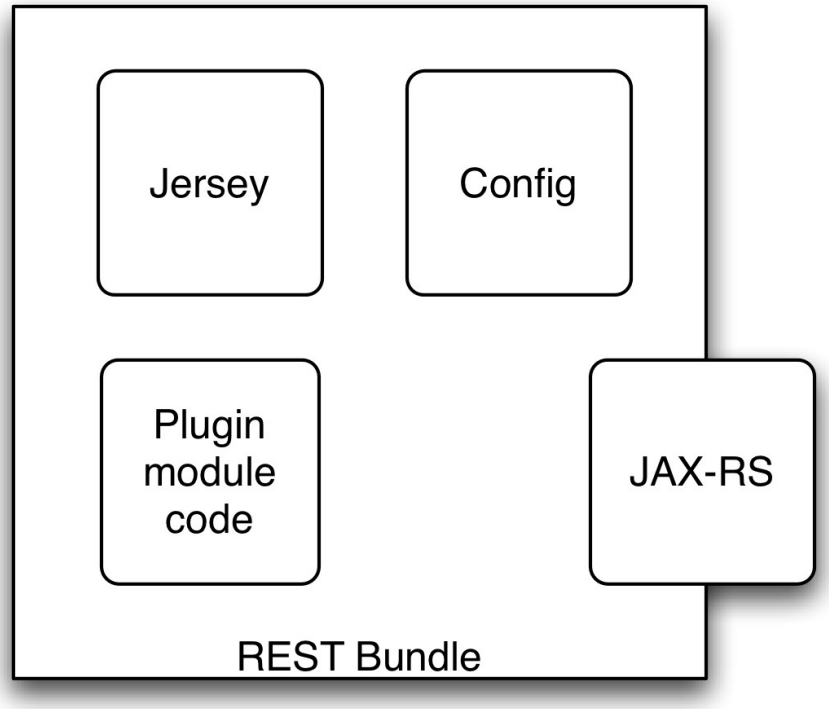
Features written once

- Example: OpenSocial-based dashboard as an OSGi plugin
- Written and owned by San Francisco team
- Contains UI, Shindig, internal services, SPI, and API



Each team can “own” a bundle

- Only JAX-RS exposed
- Complete freedom to switch to another JAX-RS implementation
- Can run multiple versions of the bundle side-by-side

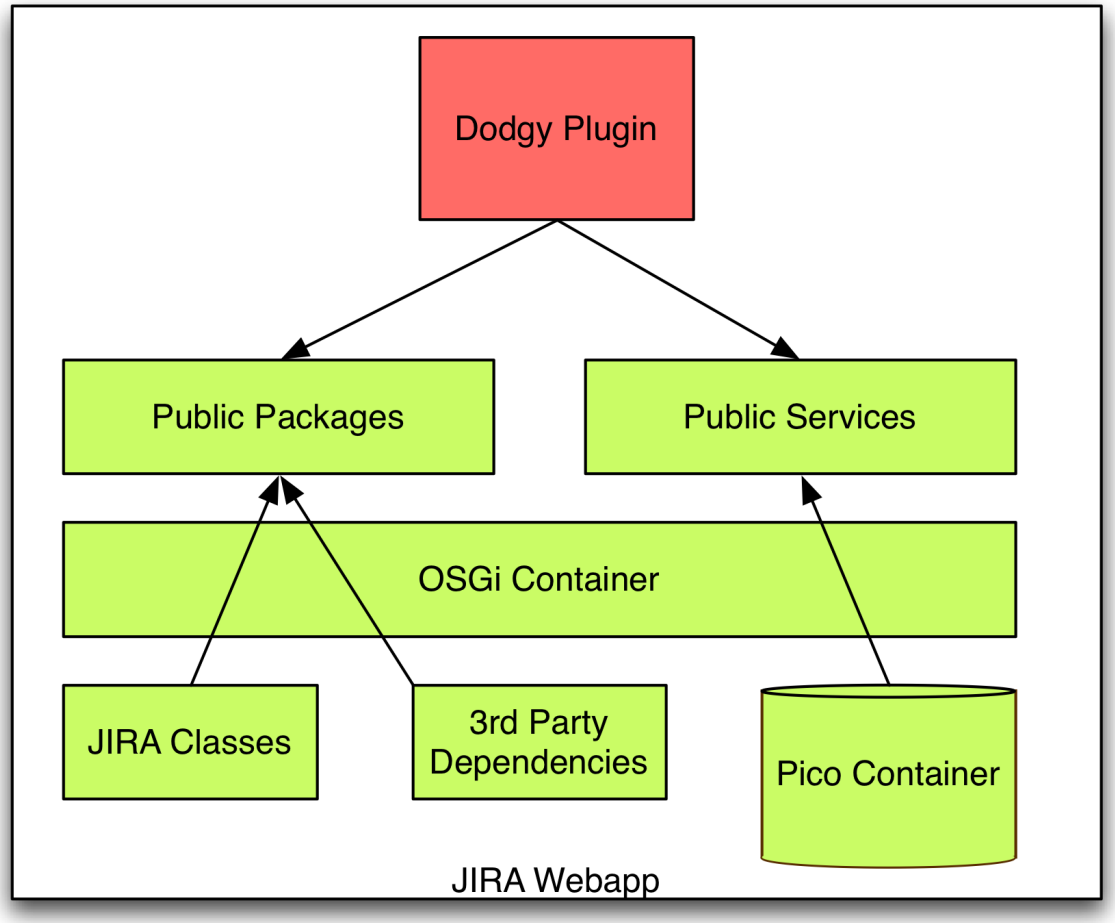


Dynamic deployment = faster dev cycle

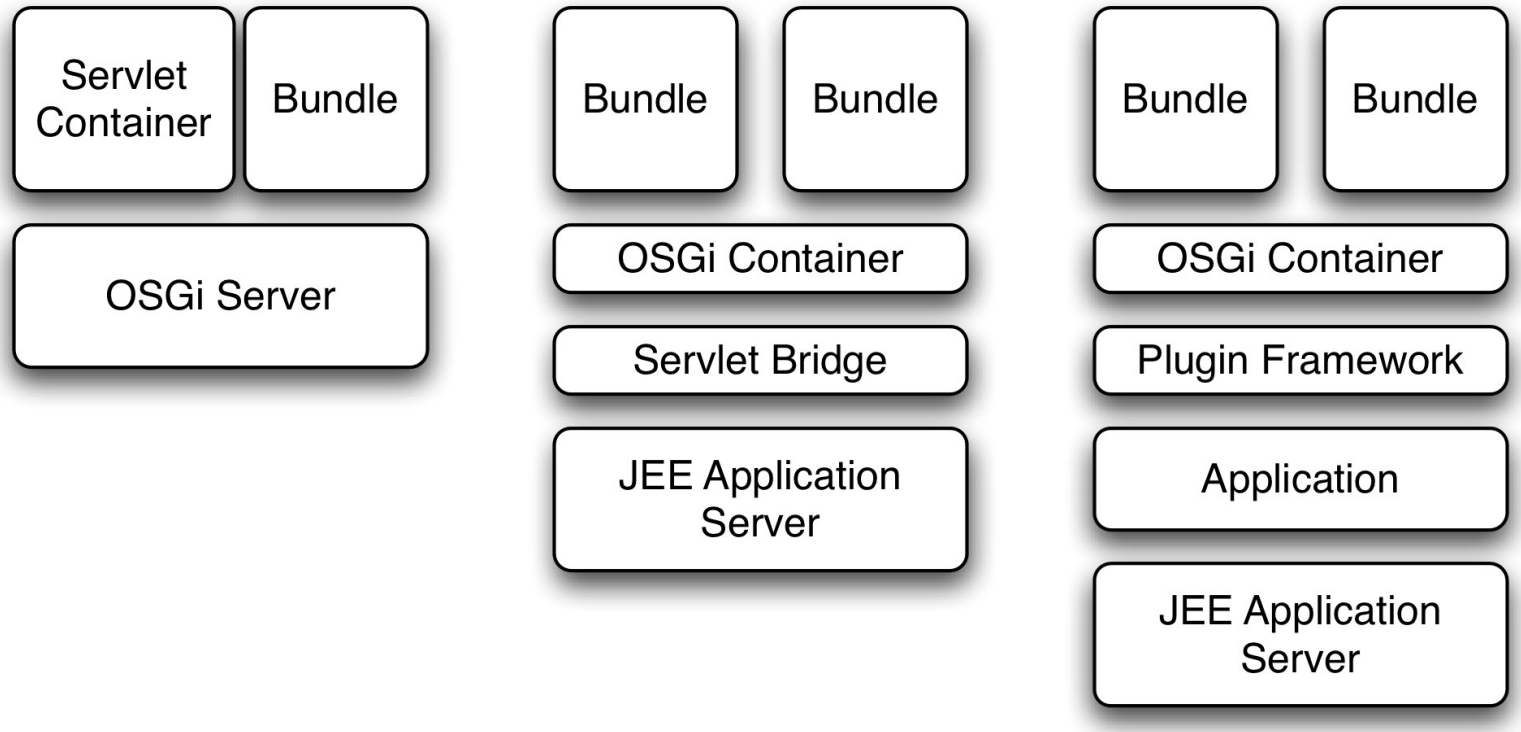
- Without OSGi
  1. Code
  2. Compile
  3. Copy to WEB-INF/lib
  4. Restart application
  5. Test in browser
- With OSGi
  1. Code
  2. Build and push to running web application
  3. Test in browser

. . . from code to browser in one or two seconds

Sandboxed plugins



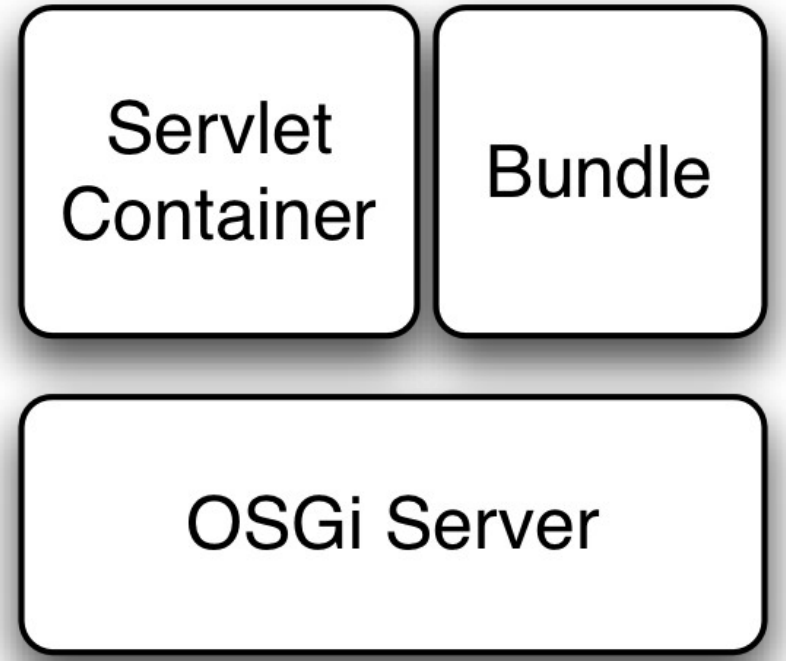
Architecture options



Server **Embedded** **Plugin**

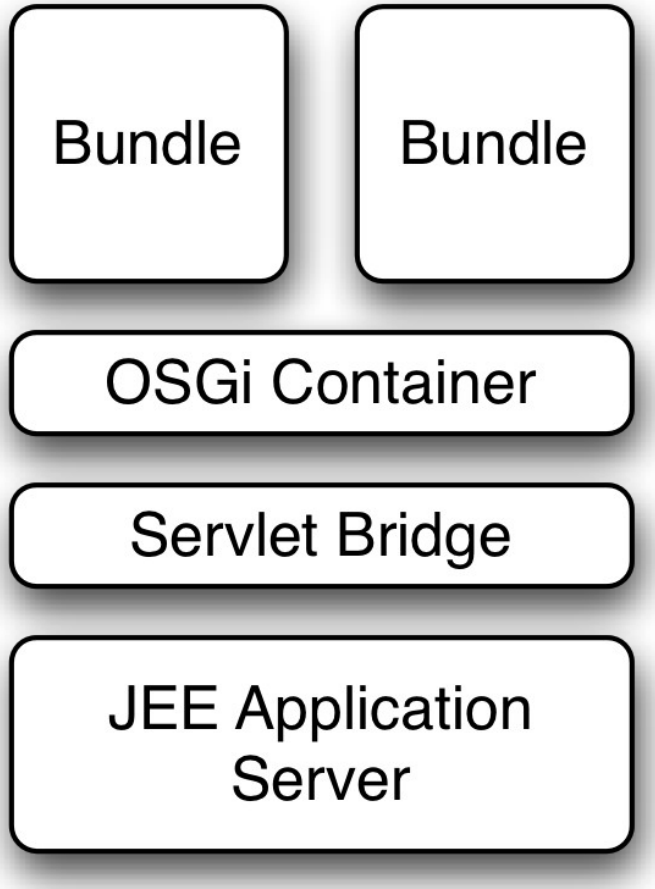
Architecture: OSGi server

- Options
  - Spring DM Server
  - Roll your own with Apache Felix, Equinox, etc.
- Advantages
  - Few deployment hassles
  - Consistent OSGi environment
- Disadvantages
  - Cannot use existing JEE infrastructure



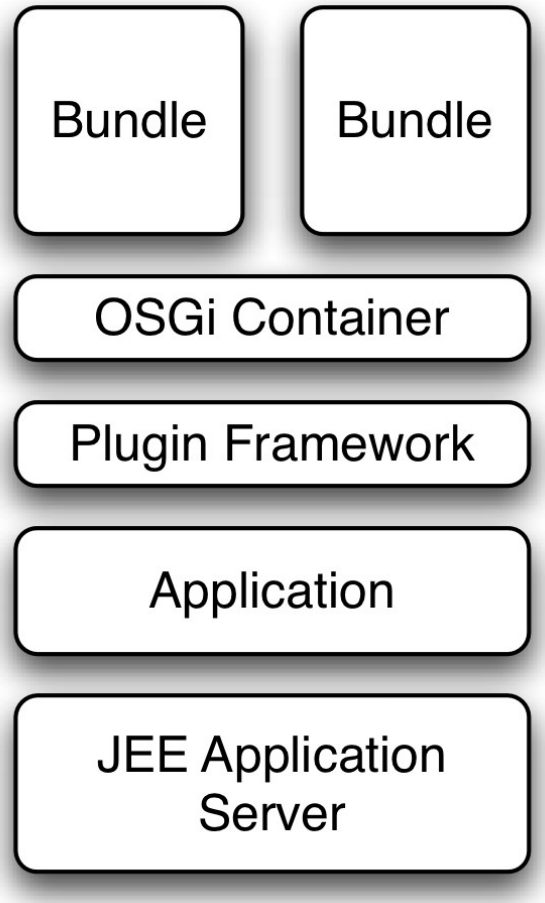
Architecture: Embed OSGi via bridge

- Options
  - Equinox Servlet bridge
- Advantages
  - Can reuse JEE servers
  - Application still pure OSGi
- Disadvantages
  - Susceptible to deployment issues
  - Poor framework support



Architecture: Embed OSGi via plugins

- Options
  - Atlassian Plugins
- Advantages
  - Can reuse JEE servers
  - Easier migration
  - Fewer library hassles
- Disadvantages
  - More complicated
  - Susceptible to deployment issues



Development experience

- How to share and consume services?
- What if bundles I depend on go down?
- How do I configure my manifest?
- What tools are available?

Services: What the tutorials show

- Registering a service:

```
MovieFinder finder = new BasicMovieFinderImpl();
registration = context.registerService(
                MovieFinder.class.getName(),
                finder, null);
```

- Consuming a service

```
ServiceReference ref = bundleContext.getServiceReference(
                         MovieFinder.class.getName());
MovieFinder finder = (MovieFinder)
                     bundleContext.getService(ref);
```

Services: The old school reality

```
public class MovieFinderTracker extends ServiceTracker {

        private final MovieListerImpl lister = new MovieListerImpl();
        private int finderCount = 0;
        private ServiceRegistration registration = null;

        public MovieFinderTracker(BundleContext context) {
                super(context, MovieFinder.class.getName(), null);
        }

        private boolean registering = false;
        public Object addingService(ServiceReference reference) {
                MovieFinder finder = (MovieFinder) context.getService(reference);
                lister.bindFinder(finder);

                synchronized(this) {
                        finderCount ++;
                        if (registering)
                                return finder;
                        registering = (finderCount == 1);
                        if (!registering)
                                return finder;
                }

                ServiceRegistration reg = context.registerService(
                MovieLister.class.getName(), lister, null);

                synchronized(this) {
                        registering = false;
                        registration = reg;
                }

                return finder;
        }

        public void removedService(ServiceReference reference, Object service) {
                MovieFinder finder = (MovieFinder) service;
                lister.unbindFinder(finder);
                context.ungetService(reference);

                ServiceRegistration needsUnregistration = null;
                synchronized(this) {
                        finderCount --;
                        if (finderCount == 0) {
                                needsUnregistration = registration;
                                registration = null;
                        }
                }

                if(needsUnregistration != null) {
                        needsUnregistration.unregister();
                }
        }
}
```

Spring Dynamic Modules

- **Great if:** You are already using Spring and like it
- **Sucks if:** You deploy 50+ and need high performance
- **Example:**

```
<service ref="beanToBeExported" interface="com.xyz.MyServiceInterface">
  <service-properties>
    <beans:entry key="myOtherKey" value="aStringValue"/>
    <beans:entry key="aThirdKey" value-ref="beanToExposeAsProperty"/>
  </service-properties>
</service>
```

```
<reference id="asyncMessageService" interface="com.xyz.MessageService"
  filter="(asynchronous-delivery=true)"/>
```

Peaberry using Guice

- **Great if:** You need speed and like Guice
- **Sucks if:** You need full lifecycle support (doesn't use the extender pattern)
- **Example:**

```
bind(export(StockQuote.class)).toProvider(
service(myQuoteImpl).attributes(names("Currency=GBP")).export());
```

```
bind(StockQuote.class).toProvider(
service(StockQuote.class).filter(ldap("(Currency=GBP)")).single());
```

Declarative Services

- **Great if:** You want to use the built-in OSGi standard
- **Sucks if:** You don't like XML and want to use constructor injection or construct your own services
- **Example:**

```
<component name="stockQuote">
  <implementation class="com.xyz.internal.MyStockQuote"/>
  <service>
    <provide interface="com.xyz.StockQuote"/>
  </service>
  <reference name="RUNNABLE"
             interface="java.lang.Runnable"
             bind="setRunnable"
             unbind="unsetRunnable"
             cardinality="0..1"
             policy="dynamic"/>
</component>
```

Bundle packaging

This is what you are up against in MANIFEST.MF:

```
APSHOT",com.atlassian.sal.api.scheduling;version="2.0.1.SNAPSHOT",com
 .atlassian.sal.api.search;version="2.0.1.SNAPSHOT",com.atlassian.sal.
 api.search.parameter;version="2.0.1.SNAPSHOT",com.atlassian.sal.api.s
 earch.query;version="2.0.1.SNAPSHOT",com.atlassian.sal.api.transactio
 n;version="2.0.1.SNAPSHOT",com.atlassian.sal.api.upgrade;version="2.0
 .1.SNAPSHOT",com.atlassian.sal.api.user;version="2.0.1.SNAPSHOT",com.
 atlassian.sal.jira.scheduling,com.atlassian.sal.spi,com.atlassian.sec
 urity.auth.trustedapps,com.atlassian.seraph.auth,com.atlassian.seraph
 .config,com.opensymphony.module.propertyset,com.opensymphony.user,jav
 ax.servlet.http,org.apache.commons.httpclient,org.apache.commons.http
 client.auth,org.apache.commons.httpclient.methods,org.apache.commons.
 httpclient.params,org.apache.commons.httpclient.util,org.apache.commo
 ns.lang,org.apache.log4j
Spring-Context: *;timeout:=60
Embed-Transitive: false
Bnd-LastModified: 1239159384564
Export-Package: com.atlassian.sal.jira.scheduling;uses:="org.apache.lo
 g4j,com.atlassian.sal.api.scheduling,com.opensymphony.module.property
 set,com.atlassian.configurable,com.atlassian.sal.api.component,com.at
 lassian.jira.service"
Bundle-Version: 2.0.1.SNAPSHOT
Bundle-Name: Shared Application Access Layer JIRA Plugin
Bundle-Description: Shared Application Access Layer JIRA 4.0 Plugin
Bundle-ClassPath: .,META-INF/lib/sal-core-2.0.1-SNAPSHOT.jar
Build-Jdk: 1.5.0_16
Private-Package: com.atlassian.sal.core.auth,com.atlassian.sal.core.co
 mponent,com.atlassian.sal.core.executor,com.atlassian.sal.core.lifecy
 cle,com.atlassian.sal.core.message,com.atlassian.sal.core.net,com.atl
 assian.sal.core.net.auth,com.atlassian.sal.core.pluginsettings,com.at
 lassian.sal.core.scheduling,com.atlassian.sal.core.search,com.atlassi
 an.sal.core.search.parameter,com.atlassian.sal.core.search.query,com.
 atlassian.sal.core.transaction,com.atlassian.sal.core.trusted,com.atl
 assian.sal.core.upgrade,com.atlassian.sal.core.util,com.atlassian.sal
 .jira,com.atlassian.sal.jira.executor,com.atlassian.sal.jira.license,
 com.atlassian.sal.jira.lifecycle,com.atlassian.sal.jira.message,com.a
 tlassian.sal.jira.pluginsettings,com.atlassian.sal.jira.project,com.a
 tlassian.sal.jira.search,com.atlassian.sal.jira.trusted,com.atlassian
 .sal.jira.upgrade,com.atlassian.sal.jira.user
Bundle-DocURL: http://www.atlassian.com/
Embed-Directory: META-INF/lib
Bundle-ManifestVersion: 2
Bundle-Vendor: Atlassian Pty Ltd
Bundle-SymbolicName: com.atlassian.sal.jira
Tool: Bnd-0.0.255
Require-Bundle: system.bundle
Embed-Dependency: *;scope=compile|runtime;inline=false
```

Using the bundle tool BND

- Maven and Ant support
- Maven 2 plugin also does packaging

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <extensions>true</extensions>
  <configuration>
    <instructions>
      <Export-Package>org.foo.myproject.api</Export-Package>
      <Private-Package>org.foo.myproject.*</Private-Package>
      <Bundle-Activator>org.foo.myproject.impl1.Activator
      </Bundle-Activator>
    </instructions>
  </configuration>
</plugin>
```

SpringSource's Bundlor

- Maven 2 and Ant support
- Handles manifest generation only

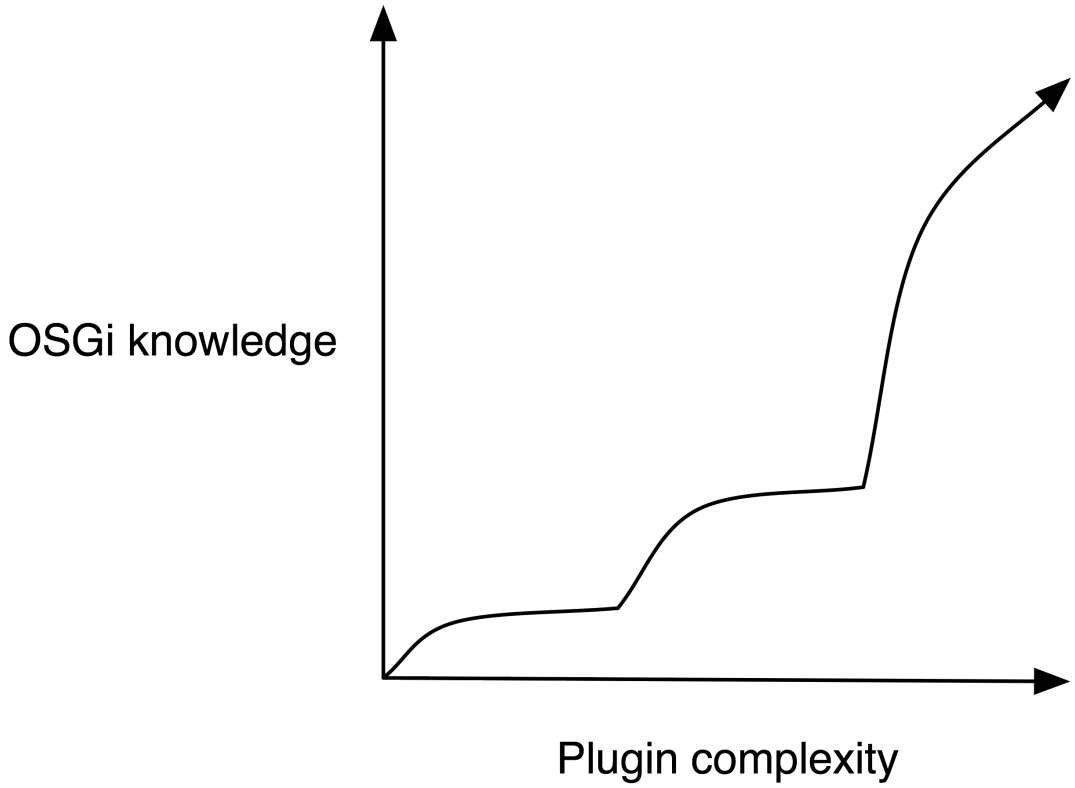
```
Bundle-ManifestVersion: 2
Bundle-SymbolicName: org.springframework.binding
Bundle-Name: Spring Binding
Bundle-Vendor: SpringSource
Import-Package:
ognl;version="[2.6.9, 3.0.0)";resolution:=optional,
org.jboss.el;version="[2.0.0, 3.0.0)";resolution:=optional
Import-Template:
org.springframework.*;version="[2.5.4.A, 3.0.0)",
org.apache.commons.logging;version="[1.1.1, 2.0.0)",
javax.el;version="[2.1.0, 3.0.0)";resolution:=optional
```

How we did it - Atlassian Plugins

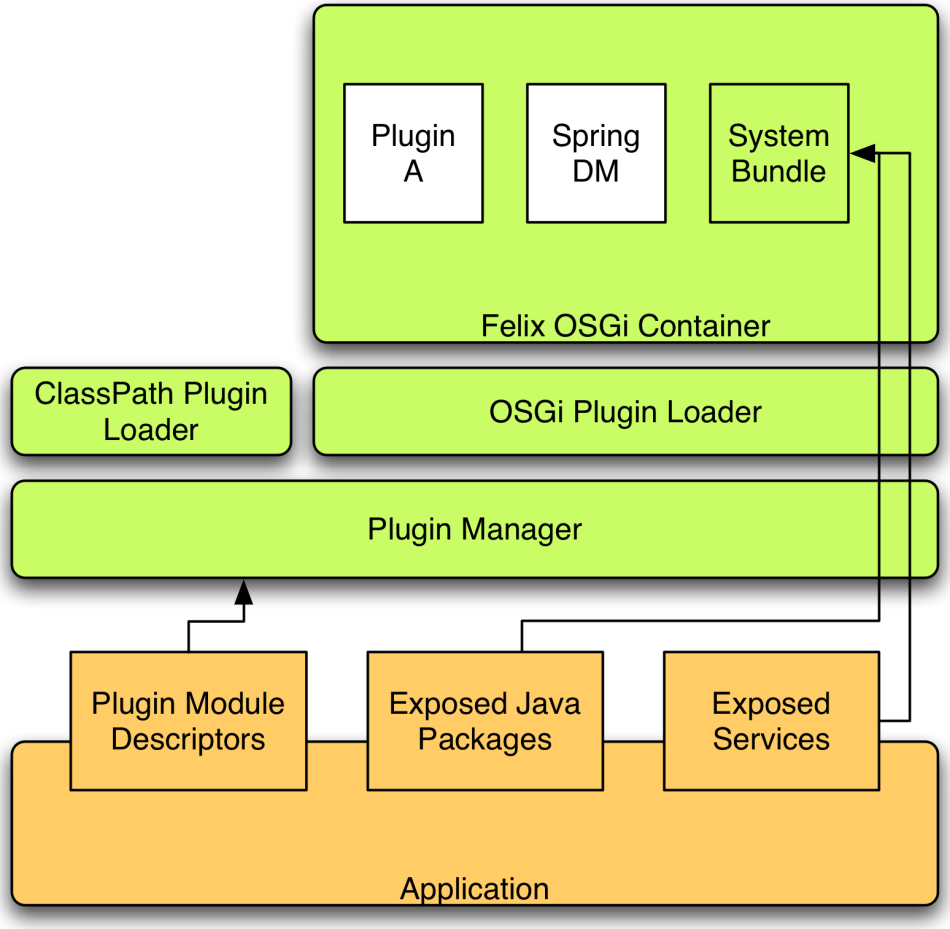
- Constraints
  - Must run on any JEE server
  - Must co-exist with millions of lines of legacy code
  - Must not require developers to be OSGi experts
- Selections
  - Felix OSGi embedded as part of our existing plugin framework
  - Spring DM to manage services
  - Automatic bundle transformation for simple plugins

One more thing - Minimal OSGi required

- Can we scale the learning curve to keep the easy plugins easy?



Plugins architecture



Plugin descriptor

- atlassian-plugin.xml

```
<atlassian-plugin key="com.xyz.example"
                  name="Example Plugin"
                  plugins-version="2">
  <plugin-info>
    <description>A sample plugin</description>
    <version>1.0</version>
    <vendor name="Atlassian"
            url="http://www.atlassian.com/"/>
  </plugin-info>
  <servlet key="test" name="Test Servlet"
           class="com.xyz.TestServlet">
    <description>An example servlet</description>
  </servlet>
</atlassian-plugin>
```

Available modules

General	Confluence	JIRA
Servlet	Job	Search
Servlet filter	Macro	Custom field
Link	Theme	Issue tab
Link section	Workflow	Portlet
REST (JAX-RS)	Language pack	RPC
Component	Editor	Workflow
Component import	Event listener	User format
Module type	RPC	Report

Plugin descriptor - Hidden OSGi

- atlassian-plugin.xml

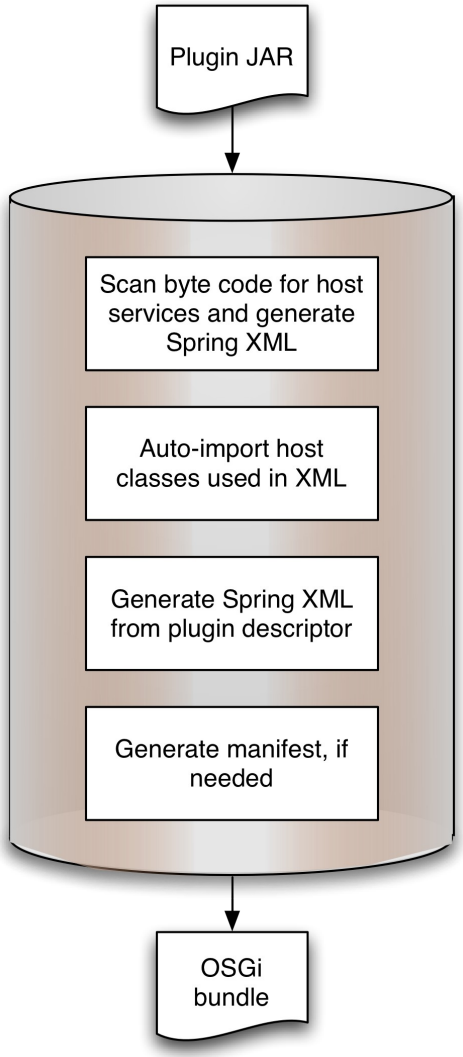
```
<atlassian-plugin key="com.xyz.example"
                  name="Example Plugin"
                  plugins-version="2">
  …
  <component key="myComponent" class="com.xyz.MyComponent"
             public="true">
    <interface>com.xyz.Component</interface>
  </component>

  <component-import key="otherComponent"
                    interface="com.abc.OtherComponent" />

</atlassian-plugin>
```

Plugin descriptor - Hidden OSGi

- Generates atlassian-plugin-spring.xml

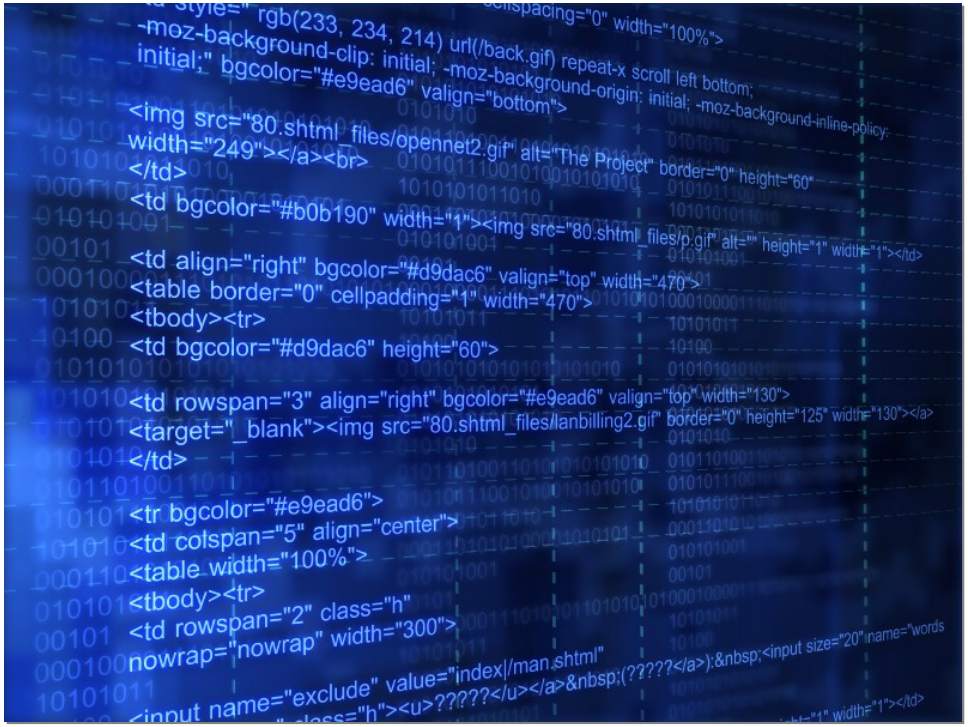
```
<beans …>
  <bean id=“myComponent class=“com.xyz.MyComponent” />

  <osgi:service id=“myComponent_service” ref=“myComponent”
                interface=“com.xyz.Component” />

  <osgi:reference id=“otherComponent"
                  interface="com.abc.OtherComponent” />
</beans>
```

Plugin to bundle process



- Goal: Allow simple plugins with no OSGi knowledge
- Three types of plugins:
  - **Simple** - no OSGi
  - **Moderate** - OSGi via plugin descriptor
  - **Complex** - OSGi via Spring XML directly

DEMO: Using Atlassian Plugins



http://atlassian.com/opensource

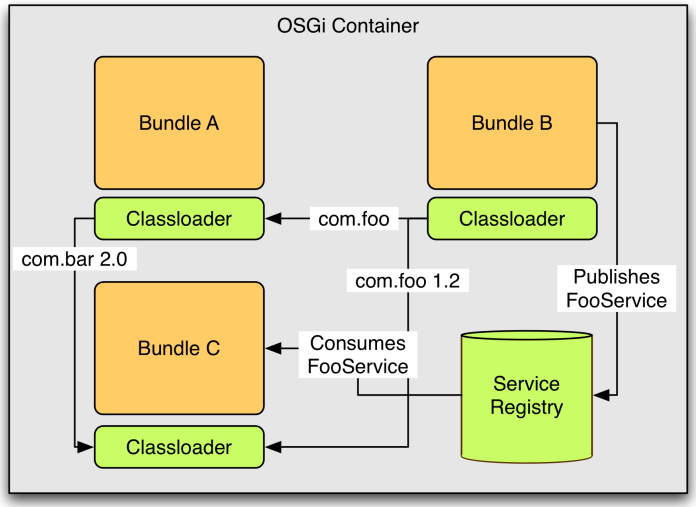
Lessons learned

Training key

- Critical topics
  - How classloaders work
  - OSGi (obviously)
  - How to develop quickly
  - Debugging

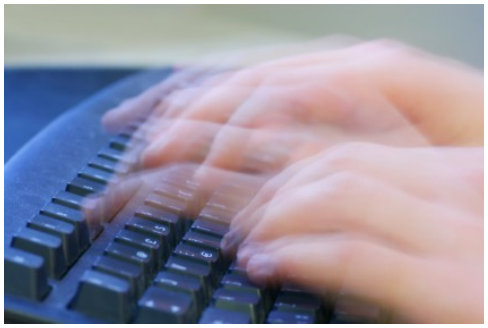
Be prepared for classloader hell

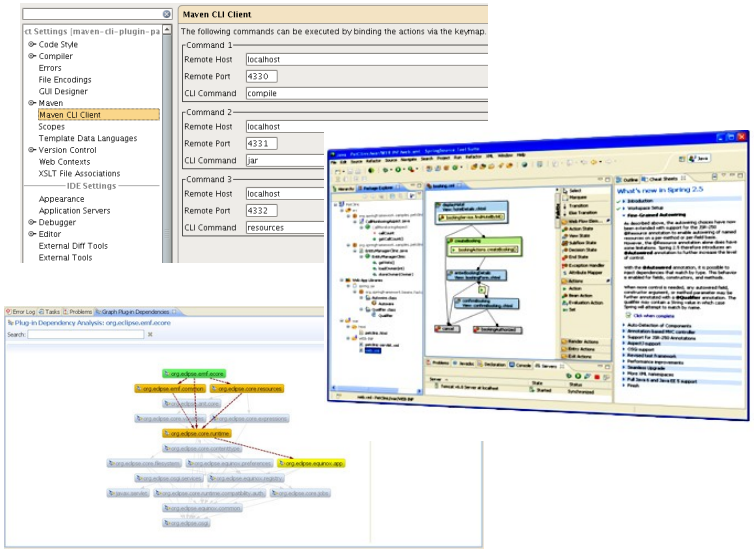
- Your new playmates
  - ClassNotFoundException
  - NoClassDefFoundError
  - ClassCastException
- Know what the thread context classloader is and how/where it is used
- Avoid Class.forName()
- Know your wires




```
- Installed bundle com.atlassian.plugin.osgi.logging (2)
- Installed bundle org.springframework.bundle.osgi.extender (3)
- Installed bundle com.springsource.org.aopalliance (4)
- Installed bundle org.springframework.bundle.spring (5)
- Installed bundle org.springframework.bundle.osgi.extensions.annotations (6)
- Installed bundle org.springframework.bundle.osgi.core (7)
- Installed bundle org.springframework.bundle.osgi.io (8)
- WIRE: 6.0 -> org.osgi.framework -> 0
- WIRE: 6.0 -> org.apache.commons.logging -> 2.0
- WIRE: 6.0 -> org.springframework.osgi.service.importer -> 7.0
- WIRE: 6.0 -> org.springframework.osgi.util -> 7.0
- WIRE: 6.0 -> org.springframework.osgi.extender -> 3.0
- WIRE: 6.0 -> org.springframework.core.annotation -> 5.0
```

Focus on development experience early





- Evaluate tools like Eclipse, the Intellij IDEA plugin Osmorc, and SpringSource Tool Suite
- Be prepared to be under-whelmed - tool support lacking in general
- Look at building custom Ant/Maven/Buildr plugins to optimize build/deploy/test cycle
- If using Maven 2 to deploy bundles, consider the Maven CLI Plugin

Deployment environments matter

- WebSphere
  - Uses OSGi but R4.0. Ensure child-first classloading
  - Classloader scanning for packages to expose to OSGi may not work. Be prepared to scan WEB-INF/lib contents directly.
- WebLogic
  - Uses own XML parser, so ensure weblogic.* is available via boot classpath delegation for JAXP
- Profiling
  - Add com.yourkit.* and com.jprofiler.* to boot delegation

Takeaway:

- Web on OSGi can rock, but beware the cliffs

Questions

Thank You

Don Brown
don@atlassian.com