An Introduction to OSGi

Alexandre Alves
June 24, 2008
Short Bio

- Employed by Oracle Corp.
  - Previously at BEA Systems

- Architect for WebLogic Event Server (rebranded into Oracle CEP)
  - Light-weight application server just for event processing
  - Completely built on top of Equinox/OSGi and completely modular

- OASIS BPEL 2.0 spec committee
Agenda

- History
- Benefits
- Architecture
- Bundles
- Services
- Conclusion
History

- The OSGi Alliance is an independent non-profit corporation
  - Deutsche Telekom, Nokia, Samsung, etc
  - IBM, Oracle, IONA, etc

- OSGi technology is the *dynamic module system for Java*
  - First release in May 2000
  - Latest version 4.1 was released in May 2007

- OSGi technology provides a
  - service-oriented,
  - component-based environment for developers
  - and offers standardized ways to manage the software lifecycle.
Agenda

- History
- Benefits
- Architecture
- Services
- Summary
Benefits

Problem Domain

- In large and complex systems, different components need to evolve separately
  - Developed by different teams
  - Re-used from other products
  - Some components need more patches than others

Solution Domain

- Organize components as independent versioned modules
  - Modules define public interface and dependencies
  - Design and implement for re-use!
- Bind modules dynamically and verify constraints
Benefits

- **Dynamic module** system for Java
  - Java does not define the concept of a module
  - Closest to it would be a JAR
    - Has no clear definition of its interfaces, dependencies, or version
- **Dynamic module** system for Java
  - One can load new classes into a Class-Loader, but cannot un-load
  - No standard way of loading new features into a running platform
    - Different technology/vendors have different approaches (e.g. JBI, J2EE)
Agenda

- History
- Benefits
- Architecture
- Bundles
- Services
- Conclusion
The Framework is split up into different layers

- Execution Environment – the VM
- Module Layer – Module system for the Java Platform
- Lifecycle Layer – Dynamic support
- Service Layer – Module collaboration
Execution Environment

- The VM used to launch the Framework
- The OSGi specification originated on the J2ME platform
- Framework implementations can scale down to small devices and scale up to large server environments
Module Layer

- Module system for the Java Platform
  - Enforces visibility rules
  - Dependency management
  - Supports versioning of bundles, the OSGi modules

- Sophisticated modularity framework
  - Provides for class space consistency for bundles
  - Supports multiple versions of packages and bundles
Lifecycle Layer provides API to manage bundles:

- Installing
- Starting
- Stopping
- Updating
- Uninstalling

All dynamically supported at runtime.
Service Layer

- Provides an in-VM service model
  - Services can be registered and consumed inside a VM
  - Again all operations are dynamic
  - Extensive support for notification of the service lifecycle
Key Concepts

- For most users, there are really just two main concepts to learn:
  - Bundles
    - Supported by Execution Environment, Module, and Lifecycle layers
  - Services
    - Supported by the Lifecycle and Service layers
Agenda

- History
- Benefits
- Architecture
- Bundles
- Services
- Conclusion
Bundle as Module

- OSGi technology’s modularity unit
  - Or, in enterprise terms, OSGi technology’s deployment unit
  - Again, main advantage of bundles is to achieve better re-use

- Regular JAR file
  - Java code
  - Resources
  - OSGi specific entries in MANIFEST.MF
Bundle Definition

- MANIFEST.MF

- Bundle-SymbolicName:
- Bundle-Version:
- Bundle-Classpath:
- Bundle-Activator:

- Import-Package:
- Export-Package:
Importing and Exporting Packages

- Import-Package/Export-Package
  - Explicit dependency model
    - Rigid documentation of public interface of module, which can be shared amongst development teams
    - Helps with build automation (don’t under-estimate the effort of building large systems)
  - Allows dynamic selection (i.e. resolve) of dependencies
    - Allows framework to find best suitable provider of a feature
    - Allows framework to dynamically change provider, useful for patching system
Bundle Versioning

Versioning

- Import-Package: com.acme.foo;version="[1.0.0.1, 2.1]"
  ==> 1.0.0.1 <= version < 2.1

- Import-Package: com.acme.foo;version="1.0.0.1"
  ==> 1.0.0.1 <= version < ∞

- Import-Package: com.acme.foo;version="1.0"
  ==> 1.0.0.0 <= version < ∞
Importing and Exporting Packages

- Attribute matching
  - Declarative way of influencing resolving

- Example:
  - Bundle A: Import-Package: com.acme.foo;company=ACME
  - Bundle B: Export-Package: com.acme.foo
  - Bundle C: Export-Package: com.acme.foo; company="ACME";
Bundle Life-cycle

- INSTALLTED:
  - Framework has bits installed

- RESOLVED:
  - Framework has resolved all dependencies successfully

- STARTING:
  - Framework is starting bundle, and invokes registered activators in the process

- ACTIVE:
  - Bundle is running

- STOPPING:
  - Framework is shutting down bundle, and invokes registered activators in the process
Bundle Activation

- Use Bundle Activator to:
  - Contribute to start and stop of bundle
  - Allows bundle to manage resources (e.g. start thread, read file)
  - Specify Bundle-Activator and import org.osgi.framework
  - Should perform work async, or return quickly
  - Provides bundle implementer access to BundleContext object

- Note-worthy: there is no standard way of installing/un-installing bundle from remote agent
Bundle Activation

Bundle-SymbolicName: example.mybundle
Bundle-Version: 1.0.0
Bundle-Activator: example.MyBundleActivator
Import-Package: org.osgi.framework

public class MyBundleActivator implements BundleActivator {
    public void start(BundleContext c) {
        // Initialize
    }
    public void stop(BundleContext c) {
        // Shutdown
    }
}
Another approach is to use Spring-DM

- Specify bundle as a Spring-DM application context
  - Spring-Context: META-INF/spring-context.xml

- Use standard Spring-bean life-cycle interfaces
  - InitializingBean
  - DisposableBean

- By default, context is created asynchronously

IMO, cleaner and simpler
Bundle-Activation

Bundle-SymbolicName: example.mybundle
Bundle-Version: 1.0.0
Spring-Context: META-INF/spring-context.xml
Import-Package:

<bean id="bundleBean" class="example.myBundleBean"
    init-method="init" destroy-method="destroy" />
Agenda

- History
- Benefits
- Architecture
- Bundles
- **Services**
- Conclusion
Services

- SOA deals with programming-in-the-large
  - Interaction between system components (e.g. WS-clients and WS-providers through WSDL)

- OSGi Service Layer allows one to bring SOA concepts (e.g. re-use, implementation abstraction) into the system component implementation level (e.g. programming-in-the-small)

- Main benefit: de-coupling of interface and implementation allows the selection of different implementation providers
  - Authentication/Authorization providers: LDAP, file-system
Service Definition

- Services are regular Java classes
  - No need to implement technology-specific interfaces

- A Service is made of three components:
  - Service name(s)
    - “example.AuthenticationService”
  - Service implementation
    - example.LDAPAuthenticationServiceImpl
  - Service (reference) properties (optional)
    - String property type = (‘file-system” | ‘ldap”)
Service Interaction

- **Service-provider bundles:**
  - Register service name(s), implementation, and properties into a Service Registry

- **Service-consumer bundles:**
  - Query Service Registry for a particular service name(s)
    - May do additional filtering by properties
    - Communicates through returned *class/interface*, does not see implementation

- **Service Registry:**
  - Similar to a map of services
Service Registration

AuthenticationService serviceImpl = new LDAPAuthenticationServiceImpl();
Dictionary properties = new Dictionary();
properties.put("type", "LDAP");
ServiceRegistration reference =
    bundleContext.registerService(
        new String [] {AuthenticationService.class.getName()},
        serviceImpl,
        properties);
Service Registration

• Or alternatively using Spring-DM:

```
<bean name="ldapService"
  class="LDAP AuthenticationServiceImpl"/>
<osgi:service ref="ldapService"
  interface="example.AuthenticationService">
  <osgi:service-properties>
    <beans:entry key="type" value="LDAP"/>
  </osgi:service-properties>
</osgi:service>
```
Referencing Services

```java
ServiceReference reference = bundleContext.getServiceReference(
    AuthenticationService.class.getName());

AuthenticationService service = (AuthenticationService)
    bundleContext.getService(reference);
```
Referencing Services

Or

<osgi:reference id="authenService"
    interface="example.AuthenticationService"/>
Services are Dynamic

- Services are dynamic, they may come and go
  - Service reference/service may be null/stale
  - Should not cache references

- ServiceListener used to keep track
  - ServiceTracker raises the ServiceListener abstraction

- Spring-DM proxies services, and will do the right thing
Agenda

- History
- Benefits
- Architecture
- Bundles
- Services
- Conclusion
Challenges

Mind-set:

- Understand that it is more work to create a modular solution, but it pays off long-term

Design-time:

- Very large Import-Packages
  - Error-prone
- Non-intuitive Import-Packages
  - Hard to get correct when reflection is used (e.g. Kodo)
Challenges

- **Runtime:**
  - Hard to debug complex class-path resolving
    - `instanceof` just fails sometimes…
  - Service availability race-conditions
    - Client applications referencing to services that have not been bound it
    - Particularly a problem during start-up

- **Certain features are missing or too hard to use:**
  - Security, Configuration support, Transaction support
Many framework implementations
- Equinox – Open source
- Felix – Open source
- Knopflerfish – Open source
- Concierge – Open source
- ProSyst

Spring Dynamic Modules for OSGi

All Eclipse-based systems run on Equinox
- Runtimes (e.g., RAP, Swordfish, Riena, ECF, EclipseLink)
- RCP, eRCP
- Tooling
Adoption

- Equinox OSGi as a component runtime
- Consistent programming model from embedded to server
- Reuse components across the spectrum

<table>
<thead>
<tr>
<th>Embedded</th>
<th>Rich Client</th>
<th>Tooling</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>eRCP</td>
<td>Nokia</td>
<td>Rational Suite</td>
<td>RAP</td>
</tr>
<tr>
<td>Sprint</td>
<td></td>
<td></td>
<td>Swordfish</td>
</tr>
<tr>
<td></td>
<td>NASA</td>
<td>Swiss Rail</td>
<td>Riena</td>
</tr>
<tr>
<td></td>
<td>JPMorgan</td>
<td></td>
<td>WAS</td>
</tr>
<tr>
<td></td>
<td>Lotus</td>
<td></td>
<td>BEA</td>
</tr>
<tr>
<td></td>
<td>Jazz</td>
<td></td>
<td>Jazz</td>
</tr>
<tr>
<td></td>
<td>SAS</td>
<td></td>
<td>Spring</td>
</tr>
<tr>
<td></td>
<td>SAS</td>
<td></td>
<td>Jazz</td>
</tr>
<tr>
<td></td>
<td>Daimler</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Riena</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

© 2008 by IBM Corp and Code 9; made available under the EPL v1.0 | March 2008
Lessons Learned when using OSGi

- There are always opportunities for re-use
  - Re-use within organization
  - Re-use of standard services
    - HTTP Service
    - Service Tracker
    - Initial Provisioning
    - Declarative Services using Spring-DM
    - Start Level Service
- Modularize at all levels
  - WL-EvS programming model itself is a separate bundle, de-coupled from other services, which means WL-EvS could in theory support other programming models, such as SCA, etc.
Conclusion

- **Standard**
  - Several different implementations are available

- **Mature**
  - Proven technology
  - Over 8 years-old (versus JSR-277/294)

- **Key-concepts**
  - Bundles: re-usability
  - Service: flexibility, extensibility