OSGi Introductory Tutorial

By Peter Kriens
Content

- Section I  –  Setting Up Eclipse and the OSGi Platform
- Section II –  OSGi Background
- Section III –  OSGi Technology
- Section IV –  Fundamental OSGi Concepts
- Section V  –  Component Interaction and Collaboration
- Section VI  –  Service Components
- Section VII –  Use Case: Developing a Chat Service
- Section VIII –  Service Tracking
- Section IX  –  Finishing Touch
- Section XI  –  Final
Section I – Setting up Eclipse and the OSGi Platform
Setup: A Clean Workspace in Eclipse
Setup: Your Infrastructure

- You need to have the following software installed on your machine in a new workspace:
  - Latest Eclipse SDK (3.2 or 3.3) ([http://www.eclipse.org](http://www.eclipse.org))
  - The aQute.tutorial.runtime.zip ([http://www.aQute.biz/OSGi/Tutorial](http://www.aQute.biz/OSGi/Tutorial)). Unzip this file and import it as a project.
  - The *bnd* plugin installed (is part of the runtime, see the jar directory)
  - All examples are available as bundles and can also be installed from [http://www.aQute.biz/OSGi/Tutorial](http://www.aQute.biz/OSGi/Tutorial)

- This tutorial assumes the PC for user interface interactions. You must translate this to your environment if you run Linux or have a Mac
Install the *bnd* Plugin

- Copy in the OS the bnd.jar file to your *eclipse/plugin* directory related to your running eclipse.exe.
  - The bnd.jar file resides in the *aQute.tutorial.runtime* project in the jar directory
  - Copy this from your operating system
- Exit Eclipse and restart 😞
- Verify: Help > About > Plugin Details
- Check for “aQute Bundle Tool”
Install the *bnd* Plugin

- The *bnd* bundle/plugin provides a convenient way to make bundles.
  - It adds a menu to files that have a “.bnd” extension
  - The *bnd* file contains all the information a requires, it calculates defaults for undefined information
  - The bnd.jar is also usable in a command line form
- For more documentation, [http://www.aqute.biz/bnd](http://www.aqute.biz/bnd)
Section II - OSGi Background
What is the OSGi Service Platform?

- A Java framework for developing (remotely) deployed service applications, that require:
  - Reliability
  - Large scale distribution
  - Wide range of devices
  - Collaborative

- Created through a collaboration of industry leaders
  - IBM, Ericsson, Nokia, Sony, Telcordia, Samsung, ProSyst, Gatespace, BenQ, Nortel, Oracle, Sybase, Espial, and many more

- Spec 4.0 publicly available at www.osgi.org ...

- *Cool!*
Why the OSGi Service Platform?

- What problems does the OSGi Service Platform address?
  - The limited (binary) software portability problem
  - The complexity of building heterogeneous software systems
    - Supporting the myriad of configuration, variations, and customizations required by today’s devices
  - Managing the software on the device
Section III - OSGi Technology
The OSGi Framework Architecture

- Allows applications to share a single Java VM
- Handles all class loading in a much better defined way than standard Java
  - Versioning!
- Gives isolation/security between applications
- Mediates between communication & collaborations between applications
- Provides life cycle management (install, start, stop, update, etc).
- Policy free
  - Policies are provided by bundles
The OSGi Platform is a Service Oriented Architectures (SOA)

- Separate the contract from the implementation
- Allows alternate implementations
- Dynamically discover and bind available implementations
- Binding based on contract (interface definitions)
- Components are reusable
- Components are not coupled to implementation details of other components, only their independent interfaces have to be known
The OSGi Service Platform is divided in a number of layers:

- Execution Environment provides a defined context for applications
- The Module layer provides class loading and packaging specifications
- The Services layer provides a collaboration model
- The extensive Security layer is embedded in all layers
Layers: OSGi Execution Environment

- Define the class library available from the VM
  - `java.*` classes
  - Additional constraints
- OSGi APIs use only a subset of Sun’s J2SE and J2ME CDC/CLDC
- OSGi Minimum EE
  - Matches most Java profiles
  - Implementations can use more than the OSGi Minimum EE
- Security is not mandatory
- CLDC is possible if class loaders are added in a device specific way
Layers: OSGi Module Layer

- Packaging of applications and libraries in *Bundles*
  - Java has significant deployment issues

- Class Loading modularization
  - Java provides the Class Path as an ordered search list, which makes it hard to control multiple applications

- Protection
  - Java can not protect certain packages and classes from others

- Versioning
  - Java can not handle multiple versions of the same package in a VM
Layers: OSGi Life Cycle Layer

- System Bundle represents the OSGi Framework
- Provides an API for managing bundles
  - Install
  - Resolve
  - Start
  - Stop
  - Refresh
  - Update
  - Uninstall
Layers: OSGi Life Cycle Layer

- Bundle is started by the Bundle Activator class
- Header in the JAR manifest file refer to this class
- Bundle Activator interface has 2 methods
  - Start: Initialize and return immediate
  - Stop: Cleanup
- The Bundle Activator gets a Bundle Context that provides access to the OSGi Framework functions
- The Framework provides the Start Level service to control the start/stop of groups of applications
Layers: OSGi Service Layer

- Provides an inside-VM service model
  - Discover (and get notified about) services based on their interface or properties, no protocol required
  - Bind to one or more services by
    - program control,
    - default rules, or
    - deployment configuration

- Service Oriented Architectures (SOA) Confusion
  - Web services bind and discover over the net
  - The OSGi Service Platform binds and discovers inside a Java VM

- The OSGi Alliance provides many standardized services
Layers: The OSGi Service Layer Evolution

- R1: Framework, Http, Log, Device Access
- R2: Package Admin, Configuration Admin, Permission Admin, User Admin, Preferences, MetaType, Service Tracker
- R5: Home Automation, Vehicle, Mobile

Timeline:
- 2000
- 2001
- 2003
- 2006
Layers: The Security Layer

- The OSGi Service Platform can be configured to be one of the most secure execution environments
- The security model is dynamic, unlike normal Java
- Fully under control of the operator
  - Fully controlled systems
  - Walled gardens
  - Fully open systems
- Based on Java 2 security
  - Permissions
  - Bundle signing
- A number of extra rules to increase the overall system security
The OSGi Implementations

- Major Framework vendors are
  - ProSyst,
  - Gatespace Telematics, and
  - IBM
  - Siemens
  - Espial

- Open source implementations
  - Apache Felix
  - Eclipse Equinox
  - Gatespace Knopflerfish

- See [http://www.aqute.biz/OSGi/HomePage](http://www.aqute.biz/OSGi/HomePage) for an overview
What Did We Learn

- The OSGi Service Platform is kind of a Java Operating System
  - Execution Environment
  - Module layer
  - Service Layer
  - Security

- It simplifies:
  - Deployment Problems
  - Software composition
  - Software management

- The OSGi technologies are implemented by many vendors and open source groups.
Section IV - Fundamental OSGi concepts
Framework Entities

OSGi Framework

Bundle A
{…}

Bundle B
{…}

Bundle C
{…}

= service, defined by java interface

= bundle
Bundles

- A *bundle* is the deliverable application
  - Like a Windows EXE file
  - Content is a JAR file
- A bundle registers zero or more services
  - A service is specified in a Java interface and may be implemented by multiple bundles
  - Services are bound to the bundle life-cycle
- Searches can be used to find services registered by other bundles
  - Query language
What is in a Bundle?

- A Bundle contains (normally in a JAR file):
  - Manifest (bundle meta data)
  - Code (classes in packages)
  - Resources (other files in the JAR file)

- The Framework:
  - Reads the bundle’s manifest
  - Installs the code and resources
  - Resolves dependencies
  - Controls the bundle life cycle

- During Runtime:
  - Calls the Bundle Activator to start the bundle
  - Manages java class path for the bundle as a network of class loaders
  - Handles the service dependencies
  - Calls the Bundle Activator to stop the bundle
  - Cleans up after the bundle
Create a Hello (and Goodbye) World Bundle

- * > New > Project
- Java Project > Next
- Fill in the information, defaults are likely to be ok. As name you can use `my.world`, then > Next
- For each tab:
  - Source, just info
  - Projects, for dependent projects. We have none now
  - In Libraries > Add JARs, add a JAR from project `aQute.tutorial.runtime`, directory `work/osgi.jar`. This provides the OSGi interfaces on the class path for the bundle
- > Finish
Create a Hello (and Goodbye) World Bundle
Create a Hello (and Goodbye) World Bundle

- Create a Java source called World.java
  - Select src folder * > New > Class

- Set the package to something like my.world

- Add the BundleActivator interface to the Interfaces list box.
  - Check the “Inherited abstract methods” checkbox, this will generate the start and stop methods for the BundleActivator interface

- The remaining examples use the package names of the provided source code, no longer my.world. You should choose your own naming to not conflict with the provided source code.
The source should look similar to the source placed adjacent

We have to fill in the // TODO blocks

start – Called when the bundle is started

stop – When the bundle is stopped (no guarantee though!)

Some tips

- F3 > Will jump to the selected type (try, click BundleActivator, and then F3)
- Control-Shift-T > Will show a source of a type (class)
- Control-Shift-R > Will show a source of a resource (not a class)
Manifest code for Hello/Goodbye World

- Each JAR file has a *manifest*

- Basic format of the manifest is defined by Sun
  - [http://java.sun.com/j2se/1.5.0/docs/guide/jar/jar.html](http://java.sun.com/j2se/1.5.0/docs/guide/jar/jar.html)

- A manifest is a text file that provides meta information about the JAR file. Used for versioning and JAR file signing.

- A manifest consists of a set of headers and values.
  - For example, `Manifest-Version: 1`

- Unrecognized headers are ignored by the Java VM, this makes it possible to extend the manifest

- The OSGi specifications have declared many different headers
  - Vendor information
  - Imported and Exported packages
  - Native code
  - ...

[Image]
Real code! Hello World (and Goodbye)

- A bundle needs a **manifest**. The *bnd* plugin can create this for you (and verify its headers for validity!)

- You have to create a *bnd* file in the project directory. * > New > File
  - *aQute.world.bnd*, pick your own name if you want

- Add:
  - **Export-Package**. Defines the packages that should be included in your bundle
    - You can use wildcards
  - **Bundle-Activator**, *aQute.tutorial.world.Activator*

- Adjust the package names to your earlier choice!

- Create the bundle by selecting the *bnd* file, then * > Make Bundle

- This creates a JAR file, which is your first bundle!
Bnd Plugin Translation

- The *bnd* plugin translates the `aQute.tutorial.world.bnd` file to the manifest, and embeds it in the JAR file.
- Any header that starts with an uppercase character is copied to the manifest by *bnd*.
- Crucial headers will be filled in by *bnd* if you do not define a value.
- Import-Package is calculated from the referred classes and matched, for example:
  - `*` (all) is default
  - `!com.acme.*` means do not import
- Export package expressions are matched against all packages on the class path.
Launch a Framework

- Run -> Run ...

- Select EquinoxFw
  - The launch file is provided by the `aQute.tutorial.runtime` project.

- This starts an Equinox Framework with the `fileinstall.jar` bundle as only bundle.

- The `fileinstall.jar` bundle watches the work directory for bundles
  - Any bundle in this directory will be installed and started
  - Removed bundles will be uninstalled
  - This is not an OSGi “standard”, it is just an application like for example Eclipse!

- Installing and uninstalling is therefore simply managing the `aQute.tutorial.runtime/work` directory
Launching EquinoxFw
Run the Hello World bundle

- The launch runs a Framework console
- Just copy (drag and drop or * > Copy and > Paste) the `aQute.tutorial.world.jar` to the work directory
  - See the “Hello World” appear in the console
- Type “**ss**” (show status)
  - Look at the active bundles
  - Notice the number for the `aQute.tutorial.world.jar` bundle. This is the bundle-id.
- Type “**stop <bundle-id>**”
  - See the “Goodbye world”
  - The “Hello World” appears again because the `fileinstall.jar` bundle starts the bundle again because it is still in the work directory.
Eclipse Self Hosted Target Environment

- Eclipse also provides the Plugin Development Environment (PDE) besides the Java Development Environment (JDE) that we used so far.

- Plugins are bundles and the PDE is therefore usable to make bundles
  - Handles dependencies
  - Generates JARs
  - Launches an Equinox Framework from a Target definition

- However, I find this environment hard to use for more general bundle development (I tried!)
  - Easy to confuse the class path management
  - JAR generation is complex and manual
  - No extensive checks on generated bundles
  - Cannot include packages from other sources than project
What Did We Learn

- The unit of deployment of an OSGi Service Platform is a bundle
- How to create a bundle with the *bnd* plugin
- How to launch an Equinox environment with a defined set of bundles, and manage it with the fileinstall.jar bundle
- How to start/stop bundles
- How the Equinox console works
- Why the Eclipse Self Hosting is not used
Section V – Component interaction and collaboration
Collaborative model

- **OSGi is more** than an Applet, MIDlet, Xlet runner
- Bundles can collaborate through:
  - *service objects*
  - *package sharing*
- A dynamic registry allows a bundle to find and track service objects
- Framework fully manages this collaboration
  - Dependencies, security
Collaborative model
Collaborative model

Java Application Manager

Midlet, Xlet, or Applet

Java

Operating System

Hardware

Service Registry

Packages

No management bundles

Midlet, Xlet, or Applet

No collaboration

No package management (versions!)

No native code
Class Path Issues

- Java applications consist of *classes* placed in *packages*
- Java searches for a class in different jar files and directories
  - These are usually specified in the CLASSPATH environment variable
- An OSGi Framework is a network of class loaders that delegate by the package name. The class loaders are parameterized by the manifest headers.
- Any dependencies between bundles are resolved by the OSGi Framework
- Complicated – But an OSGi Framework makes it painless to use
  - See the *uses:* directive illustrated in this slide
OSGi Package Dependency Resolution

Framework
- org.osgi.framework
- org.osgi.service.http

Bundle A
Export
- org.osgi.service.log
- com.ibm.service.log
- com.ibm.j9

Import
- org.osgi.service.http
- javax.servlet.http

Bundle B
Export
- ericsson.osgi
- javax.servlet
- javax.servlet.http
- org.osgi.service.log

Import
- org.osgi.service.http

A resolved

B resolved
Package or Bundle Dependencies?

- The OSGi Specifications supports both Require-Bundle and Import-Package

- Require-Bundle creates a dependency on a complete bundle
  - Simple to use
  - Imports packages that are never used

- Import-Package creates a dependency on just a package
  - Creates less brittle bundles because of substitutability
  - More cumbersome to use (Tools!)
  - The bnd utility takes the pain out of this

- In almost all cases, Import-Package is recommended because it eases deployment and version migration

- The specifications detail a number of additional problems with Require-Bundle
Service Specifics

- A service is an object registered with the Framework by a bundle to be used by other bundles.
- The semantics and syntax of a service are specified in a Java interface.
- A bundle can register a service.
- A bundle can use a service (bind to) with any cardinality:
  - 1..1, 0..1, 0..n
- A service can be discovered dynamically:
  - Active search with query filter
  - Listener interface
- Services can go away at any time! This is very dynamic!

```java
class LogService {
    static final int LOG_ERROR = 1;
    static final int LOG_WARNING = 2;
    static final int LOG_INFO = 3;
    static final int LOG_DEBUG = 4;
    public void log(int level, String message);
    public void log(int level, String message, Throwable exception);
    public void log(ServiceReference sr, int level, String message);
    public void log(ServiceReference sr, int level, String message, Throwable exception);
}
```
Service Specifics

- The Framework Service Registry is available to all bundles to collaborate with other bundles
- Different bundles (from different vendors) can implement the same interface
  - Implementation is not visible to users
  - Allows operator to replace implementations without disrupting service
- OSGi defines a standard set of services
  - Other organizations can define more (AMI-C, Ertico, JCP)
- Extensive notifications for service life cycles
- Services have a unique id
- Services can require Service Permission if security is enabled
  - Fully under operator control
- Services are associated with properties
  - Powerful query language to find appropriate service
  - Bundles can update the properties of a service dynamically
Manipulating Services

- The Bundle Context provides the methods to manipulate the service registry.

- Services registrations are handled by Service Registration objects. They can be used to unregister a service or modify its properties.

- Service Reference objects give access to the service as well as to the service’s properties.

- Access to service objects is through the getService method. These services should be returned with the ungetService method.

ServiceRegistration registerService(
    String clss,
    Object srvc,
    Dictionary prprts)

ServiceReference[] getServiceReferences(
    String clss,
    String fltr)

Object getService(
    ServiceReference reference)

boolean ungetService(
    ServiceReference rfrnc);
Services and Best Practice

- Subsystem binding, like a Log Service for example
  - Separate your code in POJOs (Plain Old Java Objects) that are not coupled to OSGi technology, and an Bundle Activator that binds the POJOs in the desired way
  - Use declarative services to handle the binding (discussed later)

- Domain objects, like a Bluetooth service that is registered when a Bluetooth device comes into range
  - Use the Service Tracker class for these situations

- Handle the dynamics, dynamic things will happen
  - Start Level service is not a way to handle dynamics, believe me
  - You can not control the initialization order, believe me
What Did We Learn

- The OSGi Service Platform provides a collaboration model that is based on
  - Services
  - Package sharing

- Sharing is complicated, but the well defined specifications make it quit painless for bundle developers

- Services provide a very powerful dynamic programming model

- SOA, without the communication overhead!
Section VI – Service Components
Declarative Services Simplify Service Programming

- The dynamic nature of services make programming more complicated
- The declarative service model simplifies handling these dynamics
  - Dependencies are defined in an XML file or via `bnd` (which translates it to an XML file in the bundle, so the result is the same)
- Declarative Services:
  - Optionally Depend on one or more services
  - Optionally Provide a service
  - Optionally lazy initialized
  - Configurable
- Example shows a hello world bundle that logs “Hello” and “Goodbye” to the Log Service

```xml
<?xml version="1.0" encoding="utf-8" ?>
<component name="aQute.tutorial.component.World">
  <implementation class="aQute.tutorial.component.World"/>
  <reference name="log" interface="org.osgi.service.log.LogService" bind="setLog"/>
</component>
```

```bnd
Service-Component ::= clause ( ',,' clause ) *
clause ::= definition | <resource path>
definition ::= <class-name>
  ( ';' 'dynamic:=' list ) ?
  ( ';' 'optional:=' list ) ?
  ( ';' 'multiple:=' list ) ?
  ( ';' name '=' <interface-name> ) *

aQute.tutorial.component.bnd

Service-Component: \n  aQute.tutorial.component.World; \n  log=org.osgi.service.log.LogService
Export-Package: aQute.tutorial.component
```

OSGI-INF/aQute.tutorial.component.World.xml

```xml
<?xml version="1.0" encoding="utf-8" ?>
<component name="aQute.tutorial.component.World">
  <implementation class="aQute.tutorial.component.World"/>
  <reference name="log" interface="org.osgi.service.log.LogService" bind="setLog"/>
</component>
```
The Component Source Code

- A component can be any class. No specific interface implementation is required.
- The activate and deactivate protected methods are called when the component is activated and deactivated
  - Dependencies must be resolved. References can be bound to the component by calling the setXXX method
  - Bnd will invoke a setXXX method when the reference name is lower case. The unsetXXX method is invoked when the policy is also dynamic.
- The Component Context class provides access to referenced services by name. The locateService method finds a reference by name
- The component instance can be sure that at any moment in time between activate and deactivate there is a valid Log Service in this example

```java
package aQute.tutorial.component;
import org.osgi.service.component.*;
import org.osgi.service.log.LogService;
public class World {
    LogService log;
    protected void activate( ComponentContext context) {
        log.log(LogService.LOG_INFO, "Hello World");
    }
    protected void deactivate( ComponentContext context) {
        log.log(LogService.LOG_INFO, "Goodbye World");
    }
    public void setLog(LogService log) {
        this.log = log;
    }
}
```
Launching

- Launch the EquinoxFw launch configuration

- You can look in the log with the log command. Enter:
  - `osgi> log`
  - Last event is at bottom, so scroll back
  - Find the “Hello World”

- Stop the bundle
  - Enter in the console:
    `osgi> stop <bundle-id>`

- Run log again
What Did We Learn

- Programming with services can be complicated
- The Declarative Services model makes service programming much, much, simpler
- How the component XML is constructed
- How bnd provides a simplified syntax
- How to use the Log Service
The End

Further reading:
http://www.aqute.biz/OSGi/Resources
http://www.osgi.org
http://bundles.osgi.org
http://www.eclipse.org/osgi
http://www.ProSyst.com