OSGi Best Practices!

BJ Hargrave
OSGi Alliance CTO
IBM Lotus

Peter Kriens
OSGi Alliance Technical Director
aQute

Session TS-1419

OSGi Best Practices!

Learn how to prevent common mistakes and build robust, reliable, modular, and extendable systems using OSGi™ technology

Agenda

Introduction to OSGi Technology

Module Layer Best Practices

Lifecycle Layer Best Practices

Service Layer Best Practices

General Best Practices

Conclusion

Q&A

Introduction to OSGi Technology

The Dynamic Module System for Java™ Platforms

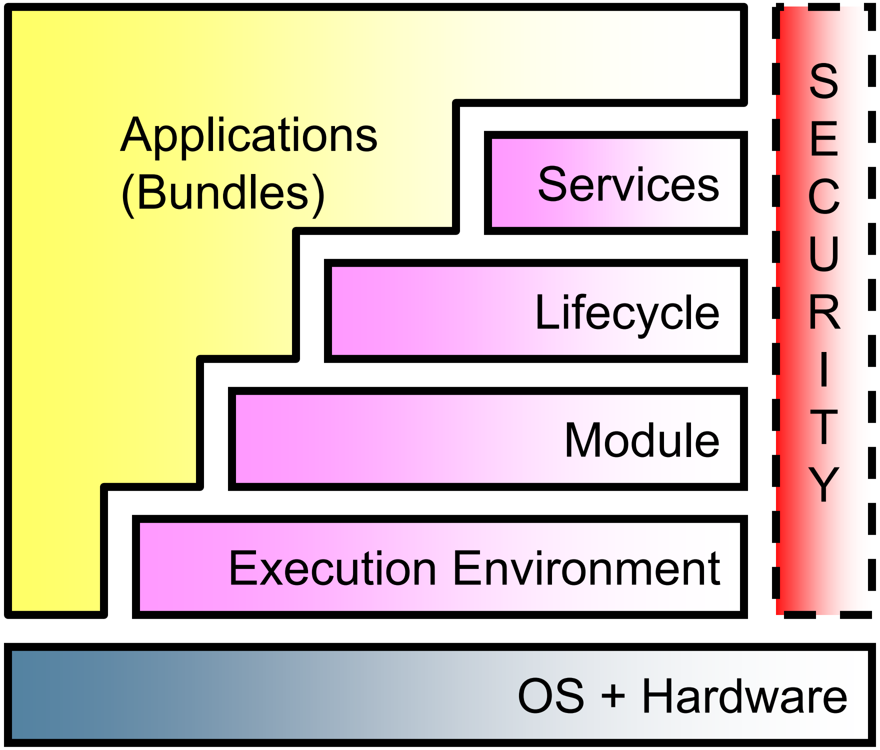
- It's a module system for the Java platform
  - Includes visibility rules, dependency management and versioning of bundles, the OSGi modules
- It's dynamic
  - Installing, starting, stopping, updating, uninstalling bundles, all dynamically at runtime
- It's service-oriented
  - Services can be registered and consumed inside a VM, again all dynamically at runtime
- A specification of the OSGi Alliance, a non-profit organization http://www.osgi.org

OSGi Technology Key Benefits

The Dynamic Module System for Java™ Platforms

- Avoids Java Archive (JAR) file hell
- Reuse code “out of the box”
- Simplifies multi-team projects
- Enables smaller systems
- Manages deployments local or remotely
- Extensive tool support
- No lock in, many providers of core technology including many open source
- Very high adoption rate

OSGi Layering



Agenda

Introduction to OSGi Technology

Module Layer Best Practices

Lifecycle Layer Best Practices

Service Layer Best Practices

General Best Practices

Conclusion

Q&A

Portable Code

Problem

- You compile your code using source level 1.3 on a Java 5 platform compiler, assuming you are safe to run on older VMs
- But then it fails to run when you deploy to a Java platform 1.3 or CDC/Foundation 1.0 environment
- It turns out that despite your 1.3 source level, you were still linked to new parts in the Java 5 class library

```
java.lang.NoSuchMethodError: java.lang.StringBuffer: method
append(Ljava/lang/StringBuffer;)Ljava/lang/StringBuffer;
not found
```

Portable Code

Best Practice

- Compile your code against the minimum suitable class libraries
- OSGi specification defines Execution Environments (EE)
  - OSGi minimum—Absolute minimum, suitable for API design
  - Foundation—Fairly complete EE, good for most applications; used for Eclipse
  - JAR files available from OSGi website
- Java platforms are backward compatible so you should always compile against the lowest version you are comfortable with
  - New features are good, but there is a cost!
  - At least think about this

Proper Imports

Problem

- You develop and test your bundles on an OSGi Service Platform that you have configured yourself
- Your colleague tries these bundles on another OSGi Service Platform and complains of a `ClassNotFoundError` in your bundles

Proper Imports

Problem

```
Code:
   import org.osgi.framework.*;
   import javax.xml.parsers.*;

   public class Activator implements BundleActivator {
    public void start(BundleContext ctxt) {
        SAXParserFactory factory =
            SAXParserFactory.newInstance();
        SAXParser parser = factory.newSAXParser();
        ...
    }
   }

Manifest:
   Import-Package: org.osgi.framework
```

Missing an import for javax.xml.parsers in the manifest

Proper Imports

Best Practice

- Do not assume that everything in the Java Runtime Environment (JRE) will be available to your bundle
  - Only java.* packages are reliably available from the boot class path
- Your bundle must import all packages that it needs
  - Except: java.* does not need to be imported
- Why?
  - Enables bundles to provide substitute implementations of JRE implementation release software version packages
- The `org.osgi.framework.bootdelegation` system property may be set differently on different configurations, so you should never rely on its setting

Minimize Dependencies

Problem

- You find an interesting bundle and want to use it
- You install it in an OSGi framework
- You find it has dependencies on other bundle
- So you find and install those bundles
- Those bundles end up depending on still other bundles…
  - Ad nauseum…

Minimize Dependencies

Best Practice

- Use `Import-Package` instead of `Require-Bundle`
  - Require-Bundle can have only one provider—the named bundle
  - Import-Package can have many providers
  - Allows for more choices during resolving
  - Has a lower fan out, which gain adds up quickly
- Use version ranges
  - Using precise version numbers gives the dependency resolver less choice
- Design your bundles
  - Don't put unrelated things in the same bundle
  - Low coupling, high cohesion

Hide Implementation Details

Problem

- You wrote a bundle that has a public API and associated implementation code
  - This implementation code defines public classes because it needs to make cross-package calls and references
- You exported all the packages in your bundle
- In the future, you release an update to the bundle with the same public API but a vastly different implementation
- You then get an angry call because you broke some customer’s code
  - And you told them not to use the implementation packages…

Hide Implementation Details

Best Practice

- Put implementation details in separate packages from the public API
  - `org.example.foo` – exported API package
  - `org.example.foo.impl` – private implementation package
- Do not export the implementation packages
  - Export and/or import the public details while keeping the implementation details private
  - `Export-Package: org.example.foo; version=1.0`

Avoid Class Loader Hierarchy Dependencies

Problem

- You are designing a multimedia system and want to allow other bundles to provide plug-in codecs
- Your design requires them to pass names of the codec classes which you load via `Class.forName`
  - Either by method call or configuration file
- This design works in a traditional tree-based class loader model since the multimedia system’s class loader has visibility to the codec classes
- However, in an OSGi environment, the multimedia system gets `ClassNotFoundErrors` since it does not have visibility to the codec classes

Avoid Class Loader Hierarchy Dependencies

Best Practice

- Better to use a safe OSGi model like services or the Extender Model to have bundles contribute codecs
  - More dynamic, you can add new services on the fly by installing bundles
- Workaround for using **`Class.forName`**
  - Use **`DynamicImport-Package: *`** and have the contributing bundles export their codec package
  - This may work but can result in unintended side effects since your bundle may import packages it did not expect

Agenda

Introduction to OSGi Technology

Module Layer Best Practices

Lifecycle Layer Best Practices

Service Layer Best Practices

General Best Practices

Conclusion

Q&A

Avoid Start Ordering Dependencies

Problem

- You develop a bundle that uses the Http Service and get the service in your BundleActivator

```
public class Activator implements BundleActivator {
 HttpService http;
 public void start(BundleContext ctxt) {
     ServiceReference
        ref = ctxt.getServiceReference(
          HttpService.class.getName());
     http = ctxt.getService(ref);
     http.registerServlet(); }}
```

- Your bundle works fine on your workstation but fails with a NullPointerException on the call to getService when integrated into the build

Avoid Start Ordering Dependencies

Best Practice

- Do not assume that you can always obtain a service during initialization
  - Bundles can start in different orders on different systems and you usually do not have control over the order
- Use **`ServiceTracker`** to track services and respond to their publication by subclassing or via a **`ServiceTrackerCustomizer`**
- Use a declarative service model like OSGi Declarative Services or Spring OSGi

Agenda

Introduction to OSGi Technology

Module Layer Best Practices

Lifecycle Layer Best Practices

Service Layer Best Practices

General Best Practices

Conclusion

Q&A

Handle Service Dynamism

Problem

- You develop a bundle with a servlet
- You get the HttpService and register your servlet
- After deployment, you receive problem reports that your servlet seems to vanish after working for a while
- It turns out the HttpService was unpublished temporarily when the HttpService bundle was stopped and restarted during an update
- Your bundle did not react and re-register the servlet

Handle Service Dynamism

Best Practice

- A service is a dynamic entity and can be unpublished after you get it
  - A bundle must respond to the lifecycle of a dependent service
- The OSGi framework provides an API to handle these dynamics but they are rather low level
- There are helpers, based on this API, like:
  - Service Tracker and Service Activator Toolkit (SAT)
  - Declarative models like Declarative Services, iPOJO, and Spring OSGi

Whiteboard Pattern

Problem

- You design a service provided by your bundle to use the familiar addListener and removeListener methods
- In practice, you find that other bundles forget to call removeListener when they stop or you stop, or forget to call addListener when you restart
- Both bundles need special code to track the other bundle or events are not properly delivered
- The OSGi LogReaderService design is an example of this problem ☹

Whiteboard Pattern

Best Practice

- Design your API to have the listener registered as a service
  - Simple
  - More robust
  - Leverages the OSGi service model and its lifecycle model awareness
- The event source tracks the listener services and calls them when there is an event to deliver
- This is called the Whiteboard Pattern
  - It can be considered an Inversion of Control pattern
- The OSGi **`EventAdmin`** design is an example of this best practice

Extender Model

Problem

- You design a Help System where other bundles contribute help content to your bundle
- The other bundles need to track the Help System bundle and contribute their Help content
- The Help System bundle must clean up when the bundles that contribute Help content are stopped
- This problem of tracking bundle lifecycles is much like the one solved by the Whiteboard Pattern
  - But there is a another pattern to address this use case
- The OSGi HttpService design is an example of this problem ☹

Extender Model

Best Practice

- The bundle being “extended” specifies a data schema
- Contributing bundles define this data in their bundle
- The extender bundle will track the bundles via certain lifecycle event and process the data, if present
    - This can include loading classes from the contributing bundle
- Extenders have more advantages
    - Lazy: less time pressure on startup and less memory later
    - More robust in case of failures: extender bundle can make consistent and policy driven choices
- Many bundles use this pattern
    - Declarative Services, iPOJO, Spring OSGi, and Eclipse Extension Point Registry

Agenda

Introduction to OSGi Technology

Module Layer Best Practices

Lifecycle Layer Best Practices

Service Layer Best Practices

General Best Practices

Conclusion

Q&A

Avoid OSGi Framework API Coupling

Problem

- You wrote your code and packaged it in a bundle
- Your code publishes an OSGi service for other bundles to use and also uses services provided by other bundles
- Your code uses the OSGi service layer API in quite a number of classes and is now coupled to the OSGi API
- You no longer can easily use your code in a non-OSGi environment

Avoid OSGi Framework API Coupling

Best Practice

- Write your code as POJOs (Plain Old Java Objects)
- Program against interfaces, not concrete classes
- Isolate the use of OSGi API to a minimal number of classes
- Let these coupled classes inject dependencies into the POJOs
- Make sure none of your domain classes depend on these OSGi coupled classes
- Use an OSGi-ready IoC container like Declarative Services or Spring OSGi to express these dependencies in a declarative form
    - Let the IoC containers handle all of the OSGi API calls

Return Quickly from Framework Callbacks

Problem

- You work in a large team building an enterprise OSGi based system
- Each developer develops their part of the system in a modular fashion and does extensive and continuous unit testing
- When all bundles are put together for integration test, a week before deadline, it takes too long to bring up the whole system
- It turns out that each bundle spent a long time in their activator and the cumulative effect on the complete system was significant

Return Quickly from Framework Callbacks

Best Practice

- Bundle developers have a tendency to do too much up front activation
- 1s per bundle (think DNS name lookup)
    - → 1 minute with 60 bundles
    - → 5 minutes with 300 bundles
- Lazy is good
    - See new lazy activation features in release 4 Version 4.1
- Framework callbacks need to return quickly
- If you need to do something that takes some time then either:
    - Use eventing
    - Spin off a background thread to perform the long running work

Thread Safety

Problem

- You develop a bundle and test it extensively
- However when deployed in the field with a set of other bundles, your bundle fails with exceptions in strange places
- Ultimately your realize that these other bundles are triggering events
  - Which your bundle receives and processes
  - But the events are being delivered on many different threads
- Time to consult a concurrency expert…

Thread Safety

Best Practice

- In an OSGi environment, framework callbacks to your bundle can occur on many different threads simultaneously
- Your code must be thread-safe!
  - Callbacks are likely running on different threads and can occur really simultaneously
  - Do not hold any locks when you call a method and you do not know the implementation, they might call back to bite you
  - Java platform monitors are intended to protect low level data structures; use higher level abstractions with time-outs for locking entities
  - In multi-core CPUs, memory access to shared mutable state must always be synchronized

Conclusion

- We have presented a number of pitfalls and showed the best practices to prevent those pitfalls
    - Some are common sense and apply to other Java environments as well
    - Some are needed because of the characteristics of the OSGi environment
- Despite these pitfalls, OSGi technology provides a robust environment for software development that gives a tremendous amount of advantages
    - Many OSGi mechanisms were designed to prevent common pitfalls in traditional Java technology programming

OSGi Service Platform

For More *Effective* Software Development!

For More Information

- If you have further question on these or want to discuss other issues in developing for OSGi
  - Please try the **`osgi-dev@www2.osgi.org`** mail list
  - http://www2.osgi.org/mailman/listinfo/osgi-dev
- OSGi Developer website
  - http://www2.osgi.org/

Q&A

BJ Hargrave

Peter Kriens

OSGi Best Practices!

BJ Hargrave
OSGi Alliance CTO
IBM Lotus

Peter Kriens
OSGi Alliance Technical Director
aQute

Session TS-1419