

Red Hat Fuse – 7.x

A Distributed, Cloud-native Integration Platform

– **Avadhut**

- **Part – 4 : Fuse Installation and CLI**
- **Fuse Installation/Fuse on EAP installation**
- **Management HawtIO**
- **Understanding CLI/Karaf container in Action**
- **Implementation of OSGi in production**

**** Commands to interact with Karaf**





Red Hat Fuse Practicals

Management using HawtIO and CLI



What is HawtIO



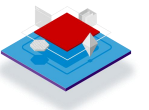
- ✓ **A modular web console for managing your Java stuffs**
- ✓ **Composed of a collection of plug-ins, each of which is an AngularJS module**
- ✓ **HawtIO has lot of plug-ins such as JMX, JVM, OSGi, Logs, ActiveMQ, Apache**
- ✓ **Camel and Spring Boot**
- ✓ **Designed by considering micro-services strategy in enterprise applications**
- ✓ **Cloud ready**
- ✓ **Available at port number 8181 for Red Hat Fuse**





Red Hat Fuse

HawtIO Monitoring





Red Hat Fuse

CLI & Karaf in Action



Enables/disables dynamic-import for a given bundle

bundle:dynamic-import <bundle-id>

Locates a specified class in any deployed bundle

bundle:find-class <class_name>

Displays OSGi headers of a given bundles

bundle:headers <bundle-id>

Displays detailed information of a given bundles

bundle:info <bundle-id>

Installs one or more bundles

osgi:install <mvn:group-id/artifact-id/version>

bundle:install <mvn:group-id/artifact-id/version>

karaf@root()| bundle:install mvn:com.kodtodya.practice/osgi-demo-example/0.0.1-SNAPSHOT

Lists all installed bundles

bundle:list <bundle-id>



Load test bundle lifecycle

bundle:load-test

Refresh bundles

bundle:refresh <bundle-id|

Displays OSGi requirements of a given bundles

bundle:requirements <bundle-id|

Restarts bundles

bundle:restart <bundle-id|

Lists OSGi services per Bundle

bundle:services <bundle-id|

Gets or sets the start level of a bundle

bundle:start-level <bundle-id| <start-level|

Bundle start|stop|status

bundle:start|stop|status <bundle-id|



Uninstall bundle

bundle:uninstall <bundle-id>

Watches and updates bundles


bundle:watch <bundle-id>

These are some sample commands

Please do refer below URL for more karaf commands and details:

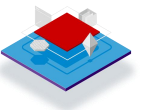
https://access.redhat.com/documentation/en-us/red_hat_fuse/7.2/html-single/apache_karaf_console_reference/





Red Hat Fuse

Implementation of OSGi in Production



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



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 – Problems

- 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 **ClassNotFoundException** in your bundles



Proper Imports

Problem

Code:

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

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

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

Manifest:

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



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 ...



OSGi Best Practices

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 plugin 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 **ClassNotFoundExceptions** since it does not have visibility to the codec classes



OSGi Best Practices

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



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



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 you 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



Let's revise

- **Part – 4 : Fuse Installation and CLI**
- **Fuse Installation/Fuse on EAP installation**
- **Management HawtIO**
- **Understanding CLI/Karaf container in Action**
- **Implementation of OSGi in production**

**** Commands to interact with Karaf**



Questions ?





Thank you..!!!

LinkedIn, GitHub, GitLab, Twitter: [@kodtodya](#)

<https://kodtodya.github.io/talks/>