OSGi Bundles and the maven-bundle-plugin

Bundles separate contract from implementation and allow for modularized development and deployment of functionality. For that to be effective, they must be defined and used correctly so inadvertent coupling does not occur. Good bundle definition and usage leads to a more flexible environment.

All the examples in this tutorial can be found in this Github repository.

1. Basic Overview

Service Overview

Before diving into bundles, it is worth a brief discussion on the structure and architecture of an OSGi service.

Services consist of

- **An API Bundle.** The API, written as Java interfaces, defines the contract of the service and should, to the extent possible, reference only those concrete classes that are loaded by the root classloader. These classes being in the java.* packages. These exceptions to the java.* rule can be made:
  - Extra interfaces can be declared by the API and used as input parameters and return values from API methods.
  - Because of their complexity and relative permanence, generated JAXB classes can be exported from an API bundle for use by its consumers.
- **At least one implementation bundle.** As the intent of loosely coupled services is to allow a variety of implementations to be deployed into the container, it is common for there to be more than one concrete implementation of a service. However, that is not a requirement. A single implementation can suffice. It should include a `blueprint.xml` associating the implementation class(es) with interface(s), providing any other wiring of beans, services, and metadata necessary, and registering with the container.

At its most basic, a service is not an especially complex construct. It is an API bundle + an Implementation bundle.

Bundles 101

Bundles are jar archives where the `META-INF/MANIFEST.MF` defines the properties of the bundle. The best resource for learning about the structure and headers in the manifest definition is in section 3.6 of the OSGi Core Specification. Additionally, the tools we use to generate our manifests - the Apache Felix Maven Bundle Plugin and the underlying BND tool - have concise and valuable documentation.

There are a few simple rules about the definition and use of bundles that will make developing with them much less stressful:

- **Embedding is an implementation detail**. Using the `Embed-Dependency` instruction provided by the `maven-bundle-plugin` will insert the specified jar(s) into the target archive and add them to the `Bundle-ClassPath`. These jars and their contained packages/classes are *not* for public consumption; they are for the internal implementation of this service implementation only.
- **Bundles should never be embedded**. Bundles expose service implementations; they do not provide arbitrary classes to be used by other bundles.
- **Bundles should expose service implementations**. This is the corollary to the previous rule. Bundles should not be created when arbitrary concrete classes are being extracted to a library. In that case, a library/jar is the appropriate module packaging type.
- **Bundles should generally only export service packages**. If there are packages internal to a bundle that comprise its implementation but not its public manifestation of the API, they should be excluded from export and kept as private packages.
- **Concrete objects that are not loaded by the root classloader should not be passed in or out of a bundle.** This is a general rule with some exceptions (JAXB generated classes being the most prominent example). Where complex objects need to be passed in or out of a service method, an interface should be defined in the API bundle.

2. Your first bundle

All the tutorials in this session will start with an implementation of a simple `Calculator` service. The API and all the service implementations and helper classes can be found here in the repository, in the `calculator` module.

The first thing to do is define the API in its own submodule.

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
    <modelVersion>4.0.0</modelVersion>

    <parent>
        <groupId>com.connexta</groupId>
        <artifactId>calculator</artifactId>
        <version>1.0-SNAPSHOT</version>
    </parent>

    <artifactId>calculator-api</artifactId>
    <name>Calculator Service :: API</name>
    <packaging>bundle</packaging>


    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.felix</groupId>
                <artifactId>maven-bundle-plugin</artifactId>
                <extensions>true</extensions>
                <configuration>
                    <instructions>
                        <Bundle-SymbolicName>${project.artifactId}</Bundle-SymbolicName>
                        <Bundle-Name>${project.name}</Bundle-Name>
                    </instructions>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

Of special note are the `packaging` directive on line 15 and the `maven-bundle-plugin` configuration.

As this is the API of the service and has no implementation classes, the instructions for the bundle plugin are quite simple.

And here is the service's interface:

Calculator.java

```
package com.connexta.calculator;

public interface Calculator<T> {
    T adder(int x, int y);

    T multiplier(int x, int y);
}
```

And that is it. The entirety of the service's API.

Note that for the purposes of these tutorials, the `Calculator` is defined with a generic return value so different permutations can be demonstrated. This is not common and in a real-world setting would require a way for consumers to specify the return types they were able to work with. The simplest way to do that would be for service implementors to provide a capability test by LDAP filter that consumers could they use. The examples in this session do not do that; however, it is a simple matter to implement and test.

3. Two clean implementations

The implementations provided in the `rootclass` module and the `rootclass-withembed` module provide the simplest implementation of the `Calculator` service. The only difference between the two is that the latter embeds the `guava` library for its internal use. Both return `Long` objects as their results.

Pom files

pom.xml - rootclass module

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
    <modelVersion>4.0.0</modelVersion>

    <parent>
        <groupId>com.connexta</groupId>
        <artifactId>calculator</artifactId>
        <version>1.0-SNAPSHOT</version>
    </parent>

    <artifactId>rootclass</artifactId>
    <name>Calculator Service :: Returns class managed by root loader</name>
    <packaging>bundle</packaging>

    <dependencies>
        <dependency>
            <groupId>com.connexta</groupId>
            <artifactId>calculator-api</artifactId>
            <version>${project.version}</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.felix</groupId>
                <artifactId>maven-bundle-plugin</artifactId>
                <extensions>true</extensions>
                <configuration>
                    <instructions>
                        <Bundle-SymbolicName>${project.artifactId}</Bundle-SymbolicName>
                        <Bundle-Name>${project.name}</Bundle-Name>
                    </instructions>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

pom.xml - rootclass-withembed module

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
    <modelVersion>4.0.0</modelVersion>

    <parent>
        <groupId>com.connexta</groupId>
        <artifactId>calculator</artifactId>
        <version>1.0-SNAPSHOT</version>
    </parent>

    <artifactId>rootclass-withembed</artifactId>
    <name>Calculator Service :: Returns class managed by root loader/embeds library for
        implementation
    </name>
    <packaging>bundle</packaging>

    <dependencies>
```

```
        <dependency>
            <groupId>com.connexta</groupId>
            <artifactId>calculator-api</artifactId>
            <version>${project.version}</version>
        </dependency>

        <dependency>
            <groupId>com.google.guava</groupId>
            <artifactId>guava</artifactId>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.felix</groupId>
                <artifactId>maven-bundle-plugin</artifactId>
                <extensions>true</extensions>
                <configuration>
                    <instructions>
                        <Bundle-SymbolicName>${project.artifactId}</Bundle-SymbolicName>
                        <Bundle-Name>${project.name}</Bundle-Name>

                        <!--
                            This is an implementation detail which should have no impact on
                            consumers of this service; no classes from the embedded library are
                            used as inputs/outputs of the service implementations.

                            However, if consumers incorrectly embed this bundle they will become
                            tightly coupled to the implementation details and fail to load becaus
                            this dependency will be missing.

                            Don't do that. Don't ever embed a bundle.

                            Don't.
                        -->
                        <Embed-Dependency>guava</Embed-Dependency>
                    </instructions>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

A quick glance between the two files will show that the only substantive difference is the `Embed-Dependency` instruction in the latter pom. As the verbose comment indicates, this is an internal implementation detail and *should* have no effect on users of the service. Only if another service incorrectly embeds the `rootclass-withembed` bundle will an issue present itself.

Implementations

CalcImpl.java - rootclass module

```
package com.connexta.rootclass;

import com.connexta.calculator.Calculator;

/**
```

```
 * This is a well-formed service implementation.
 * <p>
 * Its inputs and outputs are loaded by the root classloader and guaranteed to be available to al
 * consumers. It does not require coupling between consumers and implementors.
 */
public class CalcImpl implements Calculator<Long> {
    @Override
    public Long adder(int x, int y) {
        return (long) x + y;
    }

    @Override
    public Long multiplier(int x, int y) {
        return (long) x * y;
    }
}
```

CalcImpl.java - rootclass-withembed module

```
package com.connexta.rootclasswithembed;

import com.connexta.calculator.Calculator;
import com.google.common.math.LongMath;

public class CalcImpl implements Calculator<Long> {
    @Override
    public Long adder(int x, int y) {
        return LongMath.checkedAdd(x, y);
    }

    @Override
    public Long multiplier(int x, int y) {
        return LongMath.checkedMultiply(x, y);
    }
}
```

The difference here is again small: the latter implementation uses the Guava `LongMath` class internally for overflow checking.

Blueprints

Finally, here are the blueprints for the two implementations, also extremely similar:

blueprint.xml - rootclass module

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">

    <service interface="com.connexta.calculator.Calculator">
        <service-properties>
            <entry key="id" value="rootclass"/>
        </service-properties>

        <bean class="com.connexta.rootclass.CalcImpl"/>
    </service>

</blueprint>
```

blueprint.xml - rootclass-withembed module

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">

    <service interface="com.connexta.calculator.Calculator">
        <service-properties>
            <entry key="id" value="rootclass-withembed"/>
        </service-properties>

        <bean class="com.connexta.rootclasswithembed.CalcImpl"/>
```

```
    </service>

</blueprint>
```

These implementations are interchangeable from the perspective of their callers, even accounting for the generic typing of the return types. A consumer of the `calculator` service could request either of these implementations by ldap filter and operate the same. If the use of the return value were simply for output and display by invoking its `toString()` method, a consumer could request a `Calculator` from the container and accept any one. For our contrived examples, adding a capability filter with a `service-property` would allow consumers to request `Calculators` that returned, for example, `Long`. In that case, the consumers could accept either of these implementations and interact with them safely.

4. Don't do that

Don't Embed Bundles

The first two guidelines provided in our **Bundles 101** discussion regard embedding. Let's quickly review them:

- **Embedding is an implementation detail**.
- **Bundles should never be embedded**.

So it follows that the instructions to the `maven-bundle-plugin` for this `embed-bundle` module are wrong.

pom.xml fragment

```
<plugin>
    <groupId>org.apache.felix</groupId>
    <artifactId>maven-bundle-plugin</artifactId>
    <extensions>true</extensions>
    <configuration>
        <instructions>
            <Bundle-SymbolicName>${project.artifactId}</Bundle-SymbolicName>
            <Bundle-Name>${project.name}</Bundle-Name>

            <Embed-Dependency>rootclass-withembed</Embed-Dependency>
        </instructions>
    </configuration>
</plugin>
```

In fact, because this bundle embeds a bundle that itself embeds another library causes the bundle to fail to load. Had this embedded the {{rootclass}} bundle instead, this bundle would load and the problem would not manifest at this time; however, it would still be a grave error. This creates a pernicious link between the consuming bundle and the one that is incorrectly embedded.

Were this bundle changed to embed the `rootclass` bundle instead so that it works today, future changes to the *internal implementation details* of the `rootclass` could break its consumers.

Don't Pass Embedded Objects Between Bundles

The `library-class` bundle incorrectly returns a concrete class from an embedded library.

CalcImpl.java

```
package com.connexta.libraryclass;

import com.connexta.calculator.Calculator;
import com.connexta.commons.Wrapper;

public class CalcImpl implements Calculator<Wrapper> {
    @Override
    public Wrapper adder(int x, int y) {
        return new Wrapper((long) x + y);
    }

    @Override
    public Wrapper multiplier(int x, int y) {
        return new Wrapper((long) x * y);
```

```
    }
}
```

This forces consuming bundles to also embed that library.

However, because each bundle uses its own isolated `ClassLoader`, this will not work; rather, a `ClassCastException` will be thrown that can appear confusing:

```
com.connexta.commons.Wrapper cannot be cast to com.connexta.commons.Wrapper
```

This is caused because the fully qualified definition of a class is Loader::Package::Class; in this case, the classes have been loaded by different `ClassLoaders`.

Quoting section 3.6.6 of the OSGi spec in its entirety:

> Bundles that collaborate require the same classloader for types used in the collaboration. If multiple bundles export packages with collaboration types then they will have to be placed in disjoint classspaces, making collaboration impossible. Collaboration is significantly improved when bundles are willing to import exported packages; these imports will allow a framework to substitute exports for imports.
>
> Though substitution is recommended to increase collaboration, it is not always possible. Importing exported packages can only work when those packages are pure API and not encumbered with implementation details. Import of exported packages should only be done when:
>
> - The exported package does not use private packages. If an exported package uses private packages then it might not be substitutable and is therefore not clean API.
> - There is at least one private package that references the exported package. If no such reference exist, there is no purpose in importing it.
>
> In practice, importing exported packages can only be done with clean API-implementation separation.
> OSGi services are carefully designed to be as standalone as possible. Many libraries intertwine API and implementation in the same package making it impossible to substitute the API packages.
>
> Importing an exported package must use a version range according to its compatibility requirements, being either a consumer or a provider of that API. See Semantic Versioning on page 45 for more information.

Don't Embed Bundles, Redux

Taking a look at the `local-concrete` and `use-bundles-export1` bundles demonstrates another issue that combines the prior two issues. In this case, instead of a shared library the concrete wrapper class is defined alongside the service implementation that uses it. The consuming bundle embeds that where it should `Import-Package`. However, the export of concrete classes (other than service implementations) from bundles should be avoided wherever possible.

5. Do do this instead

The `contract` bundle demonstrates a better, safer way to pass concrete implementations between services. Its concrete class seen below...

Wrapper.java

```
package com.connexta.contract;

import com.connexta.calculator.GoodWrapper;

public class Wrapper implements GoodWrapper {
    private final long value;

    public Wrapper(long value) {
        this.value = value;
    }

    @Override
    public long getValue() {
        return value;
```

```
    }
}
```

...implements an interface declared in the `calculator-api` bundle itself. As the calculator's API is imported by all consumers of the service, they all have access to the `GoodWrapper` interface through their bundle classloaders. There is no cross-loader mismatch when attempting to reference an implementation of it.

The `GoodWrapper` interface also includes a `default` method which can be invoked by all callers.

An acceptable alternative is to put an abstract base class into the API bundle; this approach has been used with the [`abstract-contract` bundle].

6. Demo commands

Command	Consumer Command Module	Consumer Service Module	Calculator Module	Description
bundemo: embedbundle	embed-bundle	NO INJECTION	NO INJECTION	This command bundle directly embeds the rootclass-withembed calculator bundle and uses the contained implementation directly. This bundle will not install correctly as it has an unresolved constraint.
bundemo: cast	class-cast	embed-library	library-class	Calculator and consumer embed a common library and attempt to share a concrete object from it. This results in a runtime exception as the class does not share a classloader space.
bundemo: export1	use-bundles-export1	NO INJECTION	local-concrete	Creates a lambda (could use an anonymous inner class for greater verbosity) to process the concrete value returned from the calculator. Which concrete value was defined in the calculator bundle and then incorrectly embedded into the command bundle.
bundemo: imported	shared-concrete	import-concrete	exported-concrete	This calculator bundle includes a concrete result class in a package it exports. The consumer bundle imports that package; the command bundle includes it as a private package import. This allows the framework to manage the class and ensure it lives in a shared classloader space.
bundemo: bycontract	contract-command	by-contract	contract	The value shared between the calculator and consumer services is described by an interface from the API bundle. As the consumer and command only interact with the interface (being ignorant of the concrete implementation in the calculator), there are no classloader issues.
bundemo: byabscontract	abstract-contract-command	by-abstract-contract	abstract-contract	The value shared between the calculator and consumer services is described by an abstract base class from the API bundle. As the consumer and command only interact with the abstract base class (being ignorant of the concrete implementation in the calculator), there are no classloader issues.

1. Basic Overview

Service Overview

Before diving into bundles, it is worth a brief discussion on the structure and architecture of an OSGi service.

Services consist of

- **An API Bundle.** The API, written as Java interfaces, defines the contract of the service and should, to the extent possible, reference only those concrete classes that are loaded by the root classloader. These classes being in the java.* packages. These exceptions to the java.* rule can be made:
    - Extra interfaces can be declared by the API and used as input parameters and return values from API methods.
    - Because of their complexity and relative permanence, generated JAXB classes can be exported from an API bundle for use by its consumers.
- **At least one implementation bundle.** As the intent of loosely coupled services is to allow a variety of implementations to be deployed into the container, it is common for there to be more than one concrete implementation of a service. However, that is not a requirement. A single implementation can suffice. It should include a `blueprint.xml` associating the implementation class(es) with interface(s), providing any other wiring of beans, services, and metadata necessary, and registering with the container.

At its most basic, a service is not an especially complex construct. It is an API bundle + an Implementation bundle.

Bundles 101

Bundles are jar archives where the `META-INF/MANIFEST.MF` defines the properties of the bundle. The best resource for learning about the structure and headers in the manifest definition is in section 3.6 of the OSGi Core Specification. Additionally, the tools we use to generate our manifests - the Apache Felix Maven Bundle Plugin and the underlying BND tool - have concise and valuable documentation.

There are a few simple rules about the definition and use of bundles that will make developing with them much less stressful:

- **Embedding is an implementation detail**. Using the `Embed-Dependency` instruction provided by the `maven-bundle-plugin` will insert the specified jar(s) into the target archive and add them to the `Bundle-ClassPath`. These jars and their contained packages/classes are *not* for public consumption; they are for the internal implementation of this service implementation only.
- **Bundles should never be embedded**. Bundles expose service implementations; they do not provide arbitrary classes to be used by other bundles.
- **Bundles should expose service implementations**. This is the corollary to the previous rule. Bundles should not be created when arbitrary concrete classes are being extracted to a library. In that case, a library/jar is the appropriate module packaging type.
- **Bundles should generally only export service packages**. If there are packages internal to a bundle that comprise its implementation but not its public manifestation of the API, they should be excluded from export and kept as private packages.
- **Concrete objects that are not loaded by the root classloader should not be passed in or out of a bundle.** This is a general rule with some exceptions (JAXB generated classes being the most prominent example). Where complex objects need to be passed in or out of a service method, an interface should be defined in the API bundle.

2. Your first bundle

All the tutorials in this session will start with an implementation of a simple `Calculator` service. The API and all the service implementations and helper classes can be found here in the repository, in the `calculator` module.

The first thing to do is define the API in its own submodule.

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
    <modelVersion>4.0.0</modelVersion>

    <parent>
        <groupId>com.connexta</groupId>
        <artifactId>calculator</artifactId>
        <version>1.0-SNAPSHOT</version>
    </parent>

    <artifactId>calculator-api</artifactId>
    <name>Calculator Service :: API</name>
    <packaging>bundle</packaging>


    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.felix</groupId>
                <artifactId>maven-bundle-plugin</artifactId>
                <extensions>true</extensions>
                <configuration>
                    <instructions>
                        <Bundle-SymbolicName>${project.artifactId}</Bundle-SymbolicName>
                        <Bundle-Name>${project.name}</Bundle-Name>
                    </instructions>
                </configuration>
            </plugin>
        </plugins>
```

```
    </build>
</project>
```

Of special note are the `packaging` directive on line 15 and the `maven-bundle-plugin` configuration.

As this is the API of the service and has no implementation classes, the instructions for the bundle plugin are quite simple.

And here is the service's interface:

Calculator.java

```
package com.connexta.calculator;

public interface Calculator<T> {
    T adder(int x, int y);

    T multiplier(int x, int y);
}
```

And that is it. The entirety of the service's API.

Note that for the purposes of these tutorials, the `Calculator` is defined with a generic return value so different permutations can be demonstrated. This is not common and in a real-world setting would require a way for consumers to specify the return types they were able to work with. The simplest way to do that would be for service implementors to provide a capability test by LDAP filter that consumers could they use. The examples in this session do not do that; however, it is a simple matter to implement and test.

3. Two clean implementations

The implementations provided in the `rootclass` module and the `rootclass-withembed` module provide the simplest implementation of the `Calculator` service. The only difference between the two is that the latter embeds the `guava` library for its internal use. Both return `Long` objects as their results.

Pom files

pom.xml - rootclass module

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
    <modelVersion>4.0.0</modelVersion>

    <parent>
        <groupId>com.connexta</groupId>
        <artifactId>calculator</artifactId>
        <version>1.0-SNAPSHOT</version>
    </parent>

    <artifactId>rootclass</artifactId>
    <name>Calculator Service :: Returns class managed by root loader</name>
    <packaging>bundle</packaging>

    <dependencies>
        <dependency>
            <groupId>com.connexta</groupId>
            <artifactId>calculator-api</artifactId>
            <version>${project.version}</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.felix</groupId>
                <artifactId>maven-bundle-plugin</artifactId>
                <extensions>true</extensions>
```

```
                <configuration>
                    <instructions>
                        <Bundle-SymbolicName>${project.artifactId}</Bundle-SymbolicName>
                        <Bundle-Name>${project.name}</Bundle-Name>
                    </instructions>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

pom.xml - rootclass-withembed module

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
    <modelVersion>4.0.0</modelVersion>

    <parent>
        <groupId>com.connexta</groupId>
        <artifactId>calculator</artifactId>
        <version>1.0-SNAPSHOT</version>
    </parent>

    <artifactId>rootclass-withembed</artifactId>
    <name>Calculator Service :: Returns class managed by root loader/embeds library for
        implementation
    </name>
    <packaging>bundle</packaging>

    <dependencies>
        <dependency>
            <groupId>com.connexta</groupId>
            <artifactId>calculator-api</artifactId>
            <version>${project.version}</version>
        </dependency>

        <dependency>
            <groupId>com.google.guava</groupId>
            <artifactId>guava</artifactId>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.felix</groupId>
                <artifactId>maven-bundle-plugin</artifactId>
                <extensions>true</extensions>
                <configuration>
                    <instructions>
                        <Bundle-SymbolicName>${project.artifactId}</Bundle-SymbolicName>
                        <Bundle-Name>${project.name}</Bundle-Name>

                        <!--
                            This is an implementation detail which should have no impact on
                            consumers of this service; no classes from the embedded library are
                            used as inputs/outputs of the service implementations.

                            However, if consumers incorrectly embed this bundle they will become
```

```
                            tightly coupled to the implementation details and fail to load becaus
                            this dependency will be missing.

                            Don't do that. Don't ever embed a bundle.

                            Don't.
                        -->
                        <Embed-Dependency>guava</Embed-Dependency>
                    </instructions>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

A quick glance between the two files will show that the only substantive difference is the `Embed-Dependency` instruction in the latter pom. As the verbose comment indicates, this is an internal implementation detail and *should* have no effect on users of the service. Only if another service incorrectly embeds the `rootclass-withembed` bundle will an issue present itself.

Implementations

CalcImpl.java - rootclass module

```
package com.connexta.rootclass;

import com.connexta.calculator.Calculator;

/**
 * This is a well-formed service implementation.
 * <p>
 * Its inputs and outputs are loaded by the root classloader and guaranteed to be available to al
 * consumers. It does not require coupling between consumers and implementors.
 */
public class CalcImpl implements Calculator<Long> {
    @Override
    public Long adder(int x, int y) {
        return (long) x + y;
    }

    @Override
    public Long multiplier(int x, int y) {
        return (long) x * y;
    }
}
```

CalcImpl.java - rootclass-withembed module

```
package com.connexta.rootclasswithembed;

import com.connexta.calculator.Calculator;
import com.google.common.math.LongMath;

public class CalcImpl implements Calculator<Long> {
    @Override
    public Long adder(int x, int y) {
        return LongMath.checkedAdd(x, y);
    }
```

```
    @Override
    public Long multiplier(int x, int y) {
        return LongMath.checkedMultiply(x, y);
    }
}
```

The difference here is again small: the latter implementation uses the Guava `LongMath` class internally for overflow checking.

Blueprints

Finally, here are the blueprints for the two implementations, also extremely similar:

blueprint.xml - rootclass module

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">

    <service interface="com.connexta.calculator.Calculator">
        <service-properties>
            <entry key="id" value="rootclass"/>
        </service-properties>

        <bean class="com.connexta.rootclass.CalcImpl"/>
    </service>

</blueprint>
```

blueprint.xml - rootclass-withembed module

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">

    <service interface="com.connexta.calculator.Calculator">
        <service-properties>
            <entry key="id" value="rootclass-withembed"/>
        </service-properties>

        <bean class="com.connexta.rootclasswithembed.CalcImpl"/>
    </service>

</blueprint>
```

These implementations are interchangeable from the perspective of their callers, even accounting for the generic typing of the return types. A consumer of the `calculator` service could request either of these implementations by ldap filter and operate the same. If the use of the return value were simply for output and display by invoking its `toString()` method, a consumer could request a `Calculator` from the container and accept any one. For our contrived examples, adding a capability filter with a `service-property` would allow consumers to request `Calculators` that returned, for example, `Long`. In that case, the consumers could accept either of these implementations and interact with them safely.

4. Don't do that

Don't Embed Bundles

The first two guidelines provided in our **Bundles 101** discussion regard embedding. Let's quickly review them:

- **Embedding is an implementation detail**.
- **Bundles should never be embedded**.

So it follows that the instructions to the `maven-bundle-plugin` for this `embed-bundle` module are wrong.

pom.xml fragment

```
<plugin>
    <groupId>org.apache.felix</groupId>
    <artifactId>maven-bundle-plugin</artifactId>
    <extensions>true</extensions>
    <configuration>
        <instructions>
            <Bundle-SymbolicName>${project.artifactId}</Bundle-SymbolicName>
            <Bundle-Name>${project.name}</Bundle-Name>

            <Embed-Dependency>rootclass-withembed</Embed-Dependency>
        </instructions>
    </configuration>
</plugin>
```

In fact, because this bundle embeds a bundle that itself embeds another library causes the bundle to fail to load. Had this embedded the {{rootclass}} bundle instead, this bundle would load and the problem would not manifest at this time; however, it would still be a grave error. This creates a pernicious link between the consuming bundle and the one that is incorrectly embedded.

Were this bundle changed to embed the `rootclass` bundle instead so that it works today, future changes to the *internal implementation details* of the `rootclass` could break its consumers.

Don't Pass Embedded Objects Between Bundles

The `library-class` bundle incorrectly returns a concrete class from an embedded library.

CalcImpl.java

```
package com.connexta.libraryclass;

import com.connexta.calculator.Calculator;
import com.connexta.commons.Wrapper;

public class CalcImpl implements Calculator<Wrapper> {
    @Override
    public Wrapper adder(int x, int y) {
        return new Wrapper((long) x + y);
    }

    @Override
    public Wrapper multiplier(int x, int y) {
        return new Wrapper((long) x * y);
    }
}
```

This forces consuming bundles to also embed that library.

However, because each bundle uses its own isolated `ClassLoader`, this will not work; rather, a `ClassCastException` will be thrown that can appear confusing:

```
com.connexta.commons.Wrapper cannot be cast to com.connexta.commons.Wrapper
```

This is caused because the fully qualified definition of a class is Loader::Package::Class; in this case, the classes have been loaded by different `ClassLoaders`.

Quoting section 3.6.6 of the OSGi spec in its entirety:

> Bundles that collaborate require the same classloader for types used in the collaboration. If multiple bundles export packages with collaboration types then they will have to be placed in disjoint classspaces, making collaboration impossible. Collaboration is significantly improved when bundles are willing to import exported packages; these imports will allow a framework to substitute exports for imports.
>
> Though substitution is recommended to increase collaboration, it is not always possible. Importing exported packages can only work when those packages are pure API and not encumbered with implementation details. Import of exported packages should only be done when:

- The exported package does not use private packages. If an exported package uses private packages then it might not be substitutable and is therefore not clean API.
- There is at least one private package that references the exported package. If no such reference exist, there is no purpose in importing it.

In practice, importing exported packages can only be done with clean API-implementation separation.
OSGi services are carefully designed to be as standalone as possible. Many libraries intertwine API and implementation in the same package making it impossible to substitute the API packages.

Importing an exported package must use a version range according to its compatibility requirements, being either a consumer or a provider of that API. See Semantic Versioning on page 45 for more information.

Don't Embed Bundles, Redux

Taking a look at the `local-concrete` and `use-bundles-export1` bundles demonstrates another issue that combines the prior two issues. In this case, instead of a shared library the concrete wrapper class is defined alongside the service implementation that uses it. The consuming bundle embeds that where it should `Import-Package`. However, the export of concrete classes (other than service implementations) from bundles should be avoided wherever possible.

5. Do do this instead

The `contract` bundle demonstrates a better, safer way to pass concrete implementations between services. Its concrete class seen below...

Wrapper.java

```
package com.connexta.contract;

import com.connexta.calculator.GoodWrapper;

public class Wrapper implements GoodWrapper {
    private final long value;

    public Wrapper(long value) {
        this.value = value;
    }

    @Override
    public long getValue() {
        return value;
    }
}
```

...implements an interface declared in the `calculator-api` bundle itself. As the calculator's API is imported by all consumers of the service, they all have access to the `GoodWrapper` interface through their bundle classloaders. There is no cross-loader mismatch when attempting to reference an implementation of it.

The `GoodWrapper` interface also includes a `default` method which can be invoked by all callers.

An acceptable alternative is to put an abstract base class into the API bundle; this approach has been used with the `abstract-contract` bundle.

6. Demo commands

Command	Consumer Command Module	Consumer Service Module	Calculator Module	Description

bundemo: embedbundle	embed-bundle	NO INJECTION	NO INJECTION	This command bundle directly embeds the rootclass-withembed calculator bundle and uses the contained implementation directly. This bundle will not install correctly as it has an unresolved constraint.
bundemo: cast	class-cast	embed-library	library-class	Calculator and consumer embed a common library and attempt to share a concrete object from it. This results in a runtime exception as the class does not share a classloader space.
bundemo: export1	use-bundles-export1	NO INJECTION	local-concrete	Creates a lambda (could use an anonymous inner class for greater verbosity) to process the concrete value returned from the calculator. Which concrete value was defined in the calculator bundle and then incorrectly embedded into the command bundle.
bundemo: imported	shared-concrete	import-concrete	exported-concrete	This calculator bundle includes a concrete result class in a package it exports. The consumer bundle imports that package; the command bundle includes it as a private package import. This allows the framework to manage the class and ensure it lives in a shared classloader space.
bundemo: bycontract	contract-command	by-contract	contract	The value shared between the calculator and consumer services is described by an interface from the API bundle. As the consumer and command only interact with the interface (being ignorant of the concrete implementation in the calculator), there are no classloader issues.
bundemo: byabscontract	abstract-contract-command	by-abstract-contract	abstract-contract	The value shared between the calculator and consumer services is described by an abstract base class from the API bundle. As the consumer and command only interact with the abstract base class (being ignorant of the concrete implementation in the calculator), there are no classloader issues.