MaduraBundle

User Guide

Document Revision: Revision: 48

Document Revision: Revision: 48

Table of Contents

1. Change Log

Author	Date	Comment
rogerparkinson	2015-05-09	Updated description of scope and release notes for 4.1.0
rogerparkinson	2015-04-06	clarified the need to release unused bundles.
rogerparkinson	2015-04-03	Added comment about releasing a bundle.
rogerparkinson	2015-04-03	checkpoint: maven build works and all tests run, but yet to verify servlet behaviour and deletion.
rogerparkinson	2015-03-30	release notes
Roger Parkinson	2015-01-27	Problem with github tags, had to redo build.
Roger Parkinson	2015-01-23	added caveat about WEB-INF/bundles
Roger Parkinson	2014-12-19	Initial commit of restructured project(s)

2. References

[1] Apache Licence 2.0
[2] slf4j
[3] Spring Framework
[4] aopalliance
[5] OSGI
[6] VaadinSupport
[7] pizzaOrderBundle
[8] maduraBundle
[9] perspectivesManager

3. Introduction

These are the objectives:

- Define sub-elements of an application which are self-contained bundles, ie have their own Java classes and resources.
- Dynamically load these bundles into a running application.
- The sub-elements are unaware of other bundles, so no danger of naming conflicts.
- Service code and resources in the main application can be accessed transparently by the bundles.

Here's the kind of problem we're solving. Say we have a bunch of resources and code which relates to a specific set of products and we have application code which calls on the product code. We want to be able to change the products without dropping the server.

Using Madura Bundle we bundle the resources and code into a jar file. We can then arrange for that jar file to be loaded dynamically.

The application code, when it wants to access the product information and code just specifies what bundle it wants to use (of the several that might be active). After that the application code doesn't know or care that it is accessing a bundle. It looks like normal code and normal resources. The bundled resources and code are actually injected into the application classes using Spring, so apart from selecting the bundle, the application code knows nothing about the bundles.

Yes, you can do something like this with OSGi, but not quite all of it. I found that to implement OSGi in our existing software I would have to repackage all of the existing jar files and resources, including 3rd party ones. Spring [3] have done a lot of work making OSGi bundles out of 3rd party libraries, but we also have dozens of in-house libraries that would have to be migrated to OSGi before we could start using it. Not going to happen in any time frame I could set, so we cooked up this instead.

The key difference between this and OSGi is that it will let you access things on the classpath of the calling application. So all our in-house jar files need *no change whatsoever*. To be fair to OSGi it does offer a bunch of things that Madura Bundles doesn't, such as events and security. In an attempt to maintain some compatibility with OSGi the manifest details used by Madura Bundles is designed to be compatible with OSGi. Migrating from Madura Bundles to OSGi has not been tested.

A second key difference between this and OSGi is that the bundles can be loaded from Maven (again, dynamically).

There are two general ways to use Madura Bundles. You can implement a bundle listener, described in *7.4*. In this you write a listener that will be called whenever a new bundle arrives or is removed from the system. Your listener then locates the relevant beans in the bundle and puts them in some structure you define such as a list. Your application then scans this list for functions. You might use this in the following situations

- You have a list of validation operations that your application needs to call at a certain stage and you want to vary them dynamically. By deploying them in bundles with a bundle listener your application can register new validation operations as they are added (and remove them if they are deleted).
- You have various UI components you want to register in a container application. The components might be menu items, with the code to run if they are picked, forms to appear etc. These can be delivered to the application as bundles which, as they register themselves with the application, add their various components to the UI.

Rather than write a bundle listener you can, in simpler cases, just query the bundle manager for beans of a given type. All beans of that type in all the current bundles will be returned.

The second way to use Madura Bundles is dynamic proxying. In this case you can inject proxied beans from the bundles into your application. You application is unaware that what was injected was not the actual bean but a proxy. When it calls the bean the proxy maps to the currently selected bundle (there can be only one) transparently. Of course your application must have selected the current bundle before the call takes place.

This is useful where you have a section of an application which is likely to vary over time but you want sessions that were started to keep running the same code. For example if the application is order entry you might want existing orders to keep using the order entry system they started with and new orders to use the newly deployed system. So you would record the bundle name when you save

the order and when the order is fetched for further processing you can select the bundle it was saved with. New orders select the latest bundle.

But note that it only makes sense for only one bundle to be proxied at any one time (actually for any one thread). This restriction might make you consider the bundle listener approach. However see *7.5*

As well as these various way of *using* bundles there is more than one way of providing bundles:

- The simplest way is to put the bundle into WEB-INF/bundles (in a web application). These bundles are loaded at startup time and it is a way to include initial or default bundles inside a war file. Why would you do this rather than simply putting the bundle in with the other ordinary jar files? Because you might want to add more bundles and be able to select between this and the other bundles dynamically. For example you might want to supersede the initial bundle with a later version. There is one restriction with this approach. If you want to refer to a Spring resource as `classpath:myfile.xml` it will fail to find it. Placing the same file in an external directory (the next option) works just fine.
- Have the bundles in a directory and copy any new bundles into that directory. The directory will be scanned periodically and new bundles loaded.
- Pull the bundles from Maven. This is similar to the directory but instead of jar files you have place holder files which describe the Maven artifact. The actual jar files are published to Maven.

If you are the sort of person who likes to go straight to the examples then take a look at the unit tests in the source project[8] then, for something more like an application you should look at pizzaOrderBundle[7] which is a working example of a bundle that plugs into the perspective manager application[9]. PizzaOrderBundle has a self contained UI as well as internal data and logic and it uses the bundle mechanism to plug into the main application.

There are two distinct bundle managers and which one you want to use will determine how you build your bundles. In all cases, though, bundles are just jar files with some extra stuff.

4. Building a Bundle

We deploy one jar file per bundle. The jar file contains classes and resources and one or more Spring context files. It can also specify an optional class path in its manifest. The jar files specified in the classpath are loaded into the bundle with the classes and resources from the bundle jar file.

The minimum jar file contains a Spring [3] application context file. You can add classes and resources but you can also refer to classes and resources supplied by the main application. Typically there are some resources and classes in the bundle as well. The context normally defines a bean called bundleName which delivers the name of the current bundle. This is a small context file:

```
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="
http://www.springframework.org/schema/beans http://
www.springframework.org/schema/beans/spring-beans-2.0.xsd">

 <bean id="bundleName"
 class="nz.co.senanque.madura.bundle.StringWrapperImpl">
  <constructor-arg value="${bundle.name}"/>
 </bean>
 <bean id="TestBean" class="nz.co.senanque.madura.bundle.TestBeanImpl">
  <property name="content" ref="bundleName"/>
  <property name="resource" value="classpath:BundleResource.xml"/>
 </bean>
</beans>
```

The `StringWrapperImpl` class just wraps a String so that we can proxy it. We would use `java.lang.String` but it has no suitable interface. We must always define a bundleName bean so that the application can find out what bundle this is. The second bean is just an example of an ordinary bean you might define.

The framework uses the manifest to discover information about the jar file.

4.1. Building with Ant

You need to compile your java files as usual and pack them into a jar file, but you also need to include a copy of BundleRootImpl.class from the madura-bundle.jar. This is easier to do in maven, but can be done in ant in various ways.

```
<jar jarfile="${base}/bundles/bundle-2.0.jar">
    <manifest>
       <attribute name="Built-By" value="${user.name}"/>
       <attribute name="Bundle-Activator"
 value="nz.co.senanque.madura.bundle.BundleRootImpl"/>
       <attribute name="Bundle-Description" value="Test Bundle"/>
       <attribute name="Bundle-Name" value="bundle"/>
       <attribute name="Bundle-Version" value="2.0"/>
       <attribute name="Bundle-Context" value="bundle-spring.xml"/>
       <attribute name="Class-Path" value="${bundle.class.path}"/>
 </manifest>
 <fileset dir="./bundle-2.0" includes="**/*.xml,**/*.properties,**/
*.class"/>
</jar>
```

That is the ant script needed to make the jar file. The vital entries are:

Bundle-Activator The class that implements the `BundleRoot` interface. In practice it is always the one in the example, you don't really need to write another.

Bundle-Name The name of the bundle.

Bundle-Version	The version of the bundle.
Bundle-Context	The file(s) that make up the Spring context. You can specify comma lists and wildcards here, the usual Spring conventions apply.
Class-Path	This is optional and is the standard manifest class path setting used when jar files are run stand-alone. In this context the jar files referred to in the classpath will be added to the bundle.

The name+version of the bundle must be unique and they ought to match the jar file name.

All of the manifest entries are available to Spring as place holders for example you can use ${Built-By} in the Spring file if you want.

You can use the Class-Path setting to name external jar files needed by this bundle. For example you might have a Class-Path that looks like this:

```
Class-Path: lib2/ant-optional-1.5.1.jar  lib2/antlr-2.7.6.jar lib2/
aopalliance-1.0.jar lib2/asm-1.5.3.jar
```

This will look for a directory called ./lib2 containing the jar files named.

The lib2 directory, or equivalent must be a subdirectory of your bundles directory, ie a subdirectory of the directory the jar file ends up in. Also when loading bundles from `WEB-INF/bundles` you cannot use any dependencies, ie no lib2 directory. It will be ignored and no dependencies will be loaded. If you do rely on dependencies then consider pulling bundles from Maven, which handles dependencies better.

That is all there is to the minimal jar file. Of course it would be more interesting if you include some .class files and some resources in there too as well as wiring them into the context, but we will look at a more elaborate example later.

4.2. Building with Maven

The equivalent in Maven looks like this:

```
<plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-jar-plugin</artifactId>
 <version>2.5</version>
 <configuration>
  <archive>
   <index>false</index>
   <addMavenDescriptor>true</addMavenDescriptor>
   <manifest>
    <addClasspath>false</addClasspath>
   </manifest>
   <manifestEntries>
    <Built-By>${user.name}</Built-By>
    <Bundle-Activator>nz.co.senanque.madura.bundle.BundleRootImpl</Bundle-
Activator>
    <Bundle-Description>Test Bundle Maven</Bundle-Description>
    <Bundle-Name>${project.artifactId}</Bundle-Name>
    <Bundle-Version>${project.version}</Bundle-Version>
    <Bundle-Context>bundle-spring.xml</Bundle-Context>
   </manifestEntries>
  </archive>
 </configuration>
</plugin>
```

This is the simplest Maven build. You might vary it by adding a classpath in the same way we showed with Ant. However if you are loading this bundle directly from Maven (as opposed to putting the jar file in a sweep directory) then the dependencies come through automatically. It is good practice, therefore, to ensure you scope most of your dependencies as `provided` where they can

be provided by the container application. This ensures you don't get an alarmingly large memory footprint.

You also need to include a copy of the BundleRootImpl.class file in your project somehow. It must be the *actual* BundleRootImpl. An extension of that class with nothing in it will not do. The important thing is that the code running in the BundleRootImpl must have been loaded by the bundle class loader not the application class loader.

You can easily do this by adding the following to your pom file:

```
<plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-dependency-plugin</artifactId>
 <executions>
  <execution>
   <id>unpack-dependencies</id>
   <phase>compile</phase>
   <goals>
    <goal>unpack-dependencies</goal>
   </goals>
   <configuration>
    <includes>**/BundleRootImpl.class</includes>
    <outputDirectory>${project.build.directory}/classes</outputDirectory>
    <overWriteReleases>true</overWriteReleases>
    <overWriteSnapshots>true</overWriteSnapshots>
   </configuration>
  </execution>
 </executions>
</plugin>
```

5. Bundle Manager

The main application is wired like this:

```
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:util="http://www.springframework.org/schema/util"
 xmlns:aop="http://www.springframework.org/schema/aop"
 xmlns:bundle="http://www.senanque.co.nz/madura/bundle"
 xsi:schemaLocation="
http://www.springframework.org/schema/beans http://
www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/util http://www.springframework.org/
schema/util/spring-util.xsd
http://www.senanque.co.nz/madura/bundle http://www.senanque.co.nz/madura/
bundle/MaduraBundle.xsd
http://www.springframework.org/schema/aop http://www.springframework.org/
schema/aop/spring-aop.xsd">

 <bundle:manager id="bundleManager" directory="./bundles" time="1000"/>
 <bundle:bean id="bundleName"
 interface="nz.co.senanque.madura.bundle.StringWrapper"/>
 <bundle:bean id="TestBean"
 interface="nz.co.senanque.madura.bundle.TestBean"/>

 <bean id="TestBean1" class="nz.co.senanque.madura.bundle.TestBeanImpl">
  <property name="content" ref="bundleName"/>
 </bean>
</beans>
```

This is using a Spring namespace to make the definitions simpler.

The bundleManager definition has a directory to scan for jar files and a time. The time is optional. If you leave it off the scan will happen only once, at startup, which might be all you need. If you add a time (in milliseconds) then the directory will be rescanned over and over looking for new bundles. If it does find a new bundle file it will lock the bundle collection briefly while it tidies things up. This is to ensure that bundles cannot be selected by the application while the bundles are being reworked. Bundles that are already selected are not affected by this.

The bundleName bean and the TestBean are defined using `<bundle:bean/>`. They are beans that must be defined in every bundle because the application expects to proxy them. You might like to think of them as a kind of interface definition of the bundle. The application knows those beans will always be there, and that they will support that interface, but the implementation may vary on a bundle by bundle basis. The application, of course, does not need to care about that.

Finally TestBean1 has the bean bundleName injected into it. Notice that there is nothing special at all about this. We can inject any bean defined by `<bundle:bean/>` into any of our application beans.

Naturally the beans we inject from the bundle must have interfaces and we cannot accept a concrete class as a bundled bean. This is so that the beans can be proxied correctly.

The bundles directory contains either the bundle jar files you saw building in *2*, or a reference to the Maven artifact. These reference files must have names ending in `.bundle` and their contents just specify the Maven artifact for example:

```
Bundle-Artifact=nz.co.senanque:madura-bundle-maven:0.0.1-SNAPSHOT
```

You would probably name a file like this `madura-bundle-maven-0.0.1-SNAPSHOT.bundle` or similar, but the name is not used. The value defined by the Bundle-Artifact entry is used to pull the jar file and its dependencies from Maven. It is added to the bundle classpath the same way as a jar file. To add bundles dynamically you just copy a bundle file into the bundles directory. You might also use

LATEST for the version and, when you publish a new version of the bundle to Maven, just `touch` the relevant bundle file.

If you want you can add extra values to the bundle file. These are added to the properties in the bundle so they can be used in the Spring configuration and elsewhere.

There are two system properties you can specify to tell Maven where to find its repositories, but they default to sensible values so you may not need them.

madura.maven.local.repo Defaults to `${user.home}/.m2/repository`

madura.maven.remote.repos Defaults to `central,default,http://repo1.maven.org/maven2/`. You can specify multiple remote libraries. Each entry needs a name, type, url (all comma separated) and the library entries are also comma separated.

6. Selecting the Bundle

Before the application code can use the beans in the bundle it needs to pick which bundle. It does this by calling the bundleManager bean.

The methods you need to know about on the bundleManager are:

setBundle(String bundleName)	Set the current bundle to the one named and pick the highest version number.
setBundle(String bundleName, String version)	Set the current bundle to the one named, using the name and version given.

You can set the bundle just by its name and then the latest version will be selected. Or you can specify a specific bundle and version. You can switch the bundle at any time and the bundle selected will be used for the current thread, so other threads can continue working with their own bundles. If you pick a version that is not there it will throw an exception.

When do you need to set the bundle?

- In a web application some time early on when processing a request. Requests are always assigned a thread for their processing.
- In a stand-alone application some time soon after the application starts.
- In a multi-threaded application you will need to manage the threads more aggressively, ensuring that each thread picks the correct bundle before it starts processing.

If no bundle is set you will probably see a null pointer exception because the beans you thought were injecting into your code will not be injected and you will find nulls.

In a collection of bundles how do you know which one you want?

If you are using dynamic proying then in the simplest case you have one kind of bundle and you want the latest version. The older versions are just lying around doing nothing, or maybe supporting some long running operation that started before they were superseded. So when you are starting a new process (for example a new customer order) you want to pick the latest bundle. Assuming your bundles are called `xyz-001.00,xyz-002.00` and so on then you can just say `setBundle("xyz")` and you will get `xyz-002.00` because that is the latest. If you say `setBundle("xyz-001.00")` you will get that bundle.

Sometimes you might want to make this more interesting. You can, if you want, add extra information into the manifest of the jar file and all of this information is available to you by querying the bundle manager.

For example say you wanted to have several bundles covering different date ranges. You could put a from-date and a to-date field in the manifest. Remember you can put arbitrarily named fields in there.

Here is what you do in Ant:

```
<jar jarfile="${base}/bundles/bundle-2.0.jar">
    <manifest>
       <attribute name="Built-By" value="${user.name}"/>
       <attribute name="Bundle-Activator"
 value="nz.co.senanque.madura.bundle.BundleRootImpl"/>
       <attribute name="Bundle-Description" value="Test Bundle"/>
       <attribute name="Bundle-Name" value="mybundle"/>
       <attribute name="Bundle-Version" value="2.0"/>
       <attribute name="Bundle-Context" value="bundle-spring.xml"/>
       <attribute name="from-date" value="01-Jan-2011"/>
       <attribute name="to-date" value="01-Jul-2011"/>
 </manifest>
</jar>
```

For Maven you would do this:

```
<plugin>
 <groupId>org.apache.maven.plugins</groupId>
```

```
 <artifactId>maven-jar-plugin</artifactId>
 <version>2.5</version>
 <configuration>
  <archive>
   <index>false</index>
   <addMavenDescriptor>true</addMavenDescriptor>
   <manifest>
    <addClasspath>false</addClasspath>
   </manifest>
   <manifestEntries>
    <Built-By>${user.name}</Built-By>
    <Bundle-Activator>nz.co.senanque.madura.bundle.BundleRootImpl</Bundle-
Activator>
    <Bundle-Description>Test Bundle Maven</Bundle-Description>
    <Bundle-Name>${project.artifactId}</Bundle-Name>
    <Bundle-Version>${project.version}</Bundle-Version>
    <Bundle-Context>bundle-spring.xml</Bundle-Context>
    <from-date>01-Jan-2011</from-date>
    <to-date>01-Jul-2011</to-date>
   </manifestEntries>
  </archive>
 </configuration>
</plugin>
```

If you are loading your bundles from Maven you can add the entries to your `.bundles` file like this:

```
Bundle-Artifact=nz.co.senanque:madura-bundle-maven:0.0.1-SNAPSHOT
from-date=01-Jan-2011
to-date=01-Jul-2011
```

Now you can examine the bundles from your application:

```
for (BundleRoot br:bm.getAvailableBundles())
{
 String fromDate = br.getProperties().getProperty("from-date");
 String toDate = br.getProperties().getProperty("to-date");
 String bundleName = br.getProperties().getProperty("bundle.name");
 if (some test to decide if this is the bundle you want)
 {
  bm.setBundle(bundleName);
  break;
 }
}
```

The `bundle.name` (note: case sensitive) property is automatically added by the bundle manager and is the bundle name, for example `madura-bundle-maven`

For proxying if you don't pick a bundle it will default to the latest version of that bundle registered with the bundle manager. That is probably the one you want. It is good practice to explicitly set the bundle though.

The bundle pick is stored in the ThreadLocal and in a multi-threaded application, such as a web server, you usually have a session consisting of multiple requests. Each request is run on a thread assigned by the application server, and you cannot assume it is the same thread. So each request must pick the bundle every time.

But once you have picked a bundle for a session (consisting of multiple requests) you probably want to stick with it. If a new bundle is deployed part way through a session it makes sense for that session to stick with the bundle they already had. So the application code should store the picked bundle name in some kind of session storage and pick the stored name on each request.

If you are not using dynamic proxying you can query for the beans and select the one you want like this:

```
MyBean useThisBean = null;
Map<MyBean,BundleRoot> map = bm.getBeansOfType(MyBean.class);
for (Entry<MyBean,BundleRoot> entry:map.entrySet())
{
 BundleRoot br = entry.getValue();
 String fromDate = br.getProperties().getProperty("from-date");
 String toDate = br.getProperties().getProperty("to-date");
 if (some test to decide if this is the bundle you want)
 {
  useThisBean = entry.getKey();
  break;
 }
}
```

The `BundleManager.getBeansOfType(Class)` method will search all the bundles available to this bundle manager for a bean with the class specified.

This loop assumes you have only one bean of type MyBean in each bundle so you can safely store the selection criteria in the bundle properties (held in the manifest). If you want to hold multiple beans of type MyBean in the same bundle then you cannot use the manifest properties because they will be the same for all beans of type MyBean in that bundle. The simplest solution is to write a criteria bean to hold the selection criteria. You inject the dates or whatever into your criteria bean and you also inject your functional bean (eg MyBean) into the criteria bean. Then your code goes looking for all the criteria beans, selects the one that applies and calls through it to the functional bean.

When you have finished with the bundle it is prudent to release it using the bundleManager method releaseBundle(). This will release the current bundle. It doesn't actually remove the bundle, but if you subsequently delete the bundle jar file then it will garbage collect the bundle if it is not in use by anyone else. Careful releasing of bundles ensures we can keep track of what bundle is in use and what bundle is not.

7. Advanced Topics

7.1. Alternate Bundle Location

For building demos it is sometimes useful to avoid specifying the directory. For example CloudFoundry demos are better off not requiring local disk space. This may seem to eliminate the whole point of using bundles but it does allow you to demo your bundle-dependent applications without reversing out that bundle dependency.

Assuming your demo is actually a JEE application all you need to do is add a `WEB-INF/bundles` directory to your war file and copy your bundles to there. Those bundles will be loaded during application startup. Use the BundleManagerWeb rather than the BundleManagerImpl as your bundle manager.

7.2. Alternate Timer configurations

You may want to use an external scheduler to trigger the scans. This is easy to configure in Spring:

```
<task:scheduler id="myScheduler" pool-size="10" />
<task:scheduled-tasks scheduler="myScheduler">
    <task:scheduled ref="bundleManager" method="scan"
        fixed-delay="10000" />
</task:scheduled-tasks>
```

There are many options for timers in Spring and this is a simple example that works well enough. If you have specialised requirements you should be able to find what you want using some variant of this.

7.3. Exported Beans

Sometimes you want to define beans in the application and inject them into your bundled beans. These are typically things that work with transactions. You start the transaction in your application and you want all the beans, including the bundled ones, to participate in that transaction rather that start their own.

This is quite simple. Just define your BundleManager like this:

```
<bundle:manager id="bundleManager" directory="./bundles" time="1000"
 export="sessionFactory,lockFactory"/>
```

In this case we have two beans we are exporting to the bundle. The bundle just refers to those beans as usual, no special namespace is needed there. If the bundle does not need those beans it is free to ignore them. All we are saying is that the bean definition, and the instantiated copy of the bean, is available at the bundle level.

If you are not using the name space it looks like this:

```
<bean id="bundleManager"
 class="nz.co.senanque.madura.bundle.BundleManagerImpl">
 <property name="directory" value="./target/bundles"/>
 <property name="inheritableBeans">
  <map>
   <entry key="parentBean" value-ref="parentBeanExample"/>
  </map>
 </property>
</bean>

<bean id="parentBeanExample" class="java.lang.String">
 <constructor-arg value="example of a parent bean"/>
```

```
</bean>
```

In this case the container application refers to the exported bean as `parentBeanExample` and the bundles refer to it as `parentBean`.

7.4. Listening for Bundles

You can optionally listen for bundles being added or deleted. This is useful if you need your application to do something interesting like register a bundle for use in certain circumstances. You can always just query for the bundles that are there but then you would have to poll for bundles which is dull.

To listen for bundles just implement the `nz.co.senanque.madura.bundle.BundleListener` interface and define your class as a bean in the application. The bundle manager will find it so there is no need to inject it anywhere.

7.5. Can I have Multiple Bundle Managers?

Yes, you can, but be careful. If you are not using dynamic proxying there seems little point in defining muliple bundle managers. Just have your bundle listener(s), you can have several, identify the different beans and register them in in your application in the different ways they fit. For example some beans will go into one list, other beans will go into another. It's your application, you decide this.

If you are using dynamic proxying you might want to use multiple bundle managers. There are some restrictions though.

First, you won't be able to use the namespace in your Spring context file. The namespace assumes just one bundle manager. This is not a major restriction, but it makes your Spring file a little more complex.

The second thing is that you need to select a bundle *for each bundle manager* so that the proxies are all set up. Possibly you can avoid this when you know that only one bundle will be used at a time, for example in this web request we know we are using a bundle of type A but not a bundle of type B. So we may be able to avoid selecting a bundle of type B. In that case we tell the bundle A bundle manager what bundle we want and leave the bundle B manager unselected. It is probably safer to set both, though, because someone will change your code later and not realise this, and the resulting problem might be obscure.

Finally each bundle manager should point at a different sweep directory so that the bundles are not being loaded more than once.

Can you have bundles within bundles, ie one bundle refers to another bundle? This should work, though we haven't tried it ourselves. The second bundle would, of course, need to have its own bundle manager defined in the first bundle with its own sweep directory etc. If you are using proxies then you'd need to ensure the first bundle picked the second bundle before it called it. Remember this has to happen for each thread.

If you are not using proxies you need to go find the bundle using `BundleManager.getBeansOfType(Class)` or you can use the `BundleListener` approach.

7.6. Scoped Beans

Spring allows you to define beans as session scoped like this:

```
<bean id="myBean" class="com.mycompany.MyBeanImpl" scope="session"/>
```

In a web application Spring will arrange for one of these beans to be created per session, and a proxy injected where necesary. This means that each different session sees its own bean, it does not share it with the other sessions, though the code calling the bean (through the proxy) is unaware of this. Spring manages mapping the right beans to each request based on its session identifier.

But when using bundles you have applications that may switch between several bundles and each of those bundles may define a copy of `myBean` that needs to be a session bean. But you do not want one bean per session. You want one bean per session *per bundle*.

To achieve this we have a special scope named 'bundle' and you use it like this:

```
<bean id="myBean" class="com.mycompany.MyBeanImpl" scope="bundle"/>
```

The scope is set up automatically by the bundle manager so there isn't anything else you have to do. Naturally this is only valid for beans defined inside bundles, not elsewhere. It will work just fine when there is no web session, falling back to an internal dummy session, though we can't think of a situation you would need that.

A. License

The code specific to MaduraBundle is licensed under the Apache License 2.0 [1].
The dependent products have compatible licenses specified in their pom files.

B. Release Notes

4.1.0

Added scope 'bundle', a custom Spring scope to manage session beans that are specific to a bundle.

Fixed a bug in the classloader which prevented scanning for classes in a package.

4.0.4

Reworked the bundle mapping because there were holes in the code. This means theat deleting a bundle from the directory now works. Also moved the BundleManager code that handles the servlet context into its own class.

Reworked the URL calculation. Previous versions were innacurate, often returning null on a request for a URL rather than a jar:file:... style URL.

4.0.3

Clean release.

4.0.2

Problem with github tags, had to redo build.

4.0.1

Allow for missing WEB-INF/bundles.

Publish the xsd file

4.0.0

Added Maven loading mechanism.

Restructured application to allow Maven testing, specifically added the madura-bundles parent project, madura-bundle-maven and the maven-bundle-test project.

3.9.3

Fixed a problem with variable case in bundle names. Bundle names are now case insensitives.

Changed the behaviour of bundle deletions so that the bundle remains in memory, but is flagged as shut down. Actually removing the bundle caused issues when a session remained bound to the deleted bundle.

Added loading bundles from WEB-INF/bundles, useful for demos.

Reworked the bundle map to make finding which bundle is required more explicit.

Store bundle classloader in a threadlocal so we can use it in class.forName().

Local classpath failed to find the jar files: fixed.

Upgraded Spring and slf4j versions.

3.9.2

Removed benign XSD errors reported by Eclipse.

3.9.1

Moved build to maven.

3.9

Added maven pom file

3.8

Built for Java 1.7

3.7

Found and fixed several inconsistencies in classpath handling. The std bundle manager now uses BundleClassLoader exclusively and ChildFirstURLClassLoader is now deprecated.

Improved documentation on how to handle session beans.

Added FixedUrlsBundleManager, a bundle manager which accepts a fixed list of bundles used to provide demos where switching the bundles is not critical.

3.6

Fixed problem with log out.

3.5

Fixed non display of source in Eclipse.

Improved the docs to describe handling multiple bundles.

Upgraded dependency on MaduraDocs to latest version.

Modified the ChildFirstURLClassLoader to try resources both with and without the leading slash in attempting to find one.

3.4

Added a way to propagate the session scope from the owner beanfactory to the bundle beanfactory, so you can define session scope beans in the bundle.

3.3

Just rearranging the dependencies.

3.2

Added default bundle, always the last bundle loaded.

Added getBeansOfType method to bundle manager. This fetches all beans of the given class type and returns a map containing the bean and the bundle name it was found inside.

Fixed an NPE that happens if we can't open the bundles directory.

3.1

Added child first classpath and made that the default. Not completely convinced this is working right, but if there is no conflict with the parent you definitely get what is in the bundle.

Added bundle listener interface and trivial sample implementation.

Added access to bundle properties.

Added export of application beans.

3.0

Added the timer.

Removed the need for explicit init.

Simplified the docs.

Reworked some of the sample code.

2.1

Tidying the build

2.0

Added classpath handling on the bundles. This is so you can specify a list of external jar files in the bundle jar and these will be loaded in the bundle's class loader.

1.0

Initial version