MaduraBundle

User Guide

Document Revision: Revision: 48

Document Revision: Revision: 48

Table of Contents

1.Change Log..... 5

2.References..... 6

3.Objectives..... 7

4.How to use it..... 9
4.1.The Jar File..... 9
4.2.The Application..... 10
4.3.Alternate Bundle Location..... 11
4.4.Alternate Timer configurations..... 12
4.5.Building Bundles..... 12
4.6.Exported Beans..... 14
4.7.Selecting the right bundle..... 15
4.8.Listening for Bundles..... 16
4.9.Can I have Multiple Bundle Managers?..... 16
4.10.Managing Session Beans inside Bundles..... 17

A.License..... 19

B.Eclipse Setup..... 20

C.Release Notes..... 21

1. Change Log

Author	Date	Comment
roger.parkinson	Mon Apr 28	[maven-release-plugin] prepare release madura-bundle-3.9.3

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. Objectives

These are the objectives:

- Define sub-elements of an application which are self-contained, ie have their own Java classes and resources.
- Dynamically load them 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.

Here's the problem we're solving. 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 and put it into a directory. Madura Bundle sweeps that directory every so often for new jar files and loads what it finds.

The application code, when it wants to access the product information and code 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.

There are two general ways to use Madura Bundles. You can implement a bundle listener, described in *4.8*. 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 newy 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 *4.9*

4. How to use it

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.

4.1. The Jar File

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. Specifically it sets the following:

```
<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.

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. The Application

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"
 childFirst="true"/>
 <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 jar files (ie new bundles). If it does find a new jar 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 `childFirst` flag ensures that the classes in the bundle are used in preference to the classes in the application. 'true' is actually the default so you don't need to specify it, but you can make it false to have the application classes override the ones in the bundle. You probably do not want to do this though.

This may not be working. It is better to have no classes in common between your app and your bundle so the override issue never comes up. You can, of course, have classes in common in separate bundles and they will not conflict.

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.

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, 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 pick the next version up if there is one.

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.

4.3. 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.

4.4. 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.

4.5. Building Bundles

As we have already seen the bundle is just a specialised jar file. You can build it with maven like this:

```
<plugin>
  <artifactId>maven-jar-plugin</artifactId>
  <version>2.3.1</version>
  <executions>
    <execution>
      <id>default-jar</id>
      <phase>package</phase>
      <goals>
        <goal>jar</goal>
      </goals>
      <configuration>
      <archive>
        <index>false</index>
        <addMavenDescriptor>false</addMavenDescriptor>
        <manifest>
          <addClasspath>false</addClasspath>
        </manifest>
        <manifestEntries>
          <Built-By>${user.name}</Built-By>
          <Bundle-Activator>nz.co.senanque.madura.bundle.BundleRootImpl</
Bundle-Activator>
          <Bundle-Description>user.details</Bundle-Description>
          <Bundle-Name>${project.artifactId}</Bundle-Name>
          <Bundle-Version>${project.version}</Bundle-Version>
          <Bundle-Context>user-spring.xml</Bundle-Context>
        </manifestEntries>
      </archive>
      </configuration>
    </execution>
  </executions>
</plugin>
```

The main thing to notice here are the manifest entries which are the same as the ones described in , except that the `Class-Path` is not yet covered. Adding that is simple enough but takes another step.

```
<plugin>
  <artifactId>maven-jar-plugin</artifactId>
  <version>2.3.1</version>
  <executions>
    <execution>
```

```
      <id>default-jar</id>
      <phase>package</phase>
      <goals>
        <goal>jar</goal>
      </goals>
      <configuration>
      <archive>
        <index>false</index>
        <addMavenDescriptor>false</addMavenDescriptor>
        <manifest>
          <addClasspath>false</addClasspath>
        </manifest>
        <manifestEntries>
          <Built-By>${user.name}</Built-By>
          <Bundle-Activator>nz.co.senanque.madura.bundle.BundleRootImpl</
Bundle-Activator>
          <Bundle-Description>user.details</Bundle-Description>
          <Bundle-Name>${project.artifactId}</Bundle-Name>
          <Bundle-Version>${project.version}</Bundle-Version>
          <Bundle-Context>user-spring.xml</Bundle-Context>
        </manifestEntries>
      </archive>
      </configuration>
    </execution>
  </executions>
</plugin>


<plugin>
  <artifactId>maven-jar-plugin</artifactId>
  <version>2.3.1</version>
  <executions>
    <execution>
      <id>default-jar</id>
      <phase>package</phase>
      <goals>
        <goal>jar</goal>
      </goals>
      <configuration>
      <archive>
        <index>false</index>
        <addMavenDescriptor>false</addMavenDescriptor>
        <manifest>
   <addClasspath>true</addClasspath>
   <classpathPrefix>lib</classpathPrefix>
        </manifest>
        <manifestEntries>
          <Built-By>${user.name}</Built-By>
          <Bundle-Activator>nz.co.senanque.madura.bundle.BundleRootImpl</
Bundle-Activator>
          <Bundle-Description>user.details</Bundle-Description>
          <Bundle-Name>${project.artifactId}</Bundle-Name>
          <Bundle-Version>${project.version}</Bundle-Version>
          <Bundle-Context>user-spring.xml</Bundle-Context>
        </manifestEntries>
      </archive>
      </configuration>
    </execution>
  </executions>
```

```
</plugin>
```

The difference here is in the `manifest` tag which has a `true` value for `addClassPath`, and a second entry `<classpathPrefix>lib</classpathPrefix>`. The second entry allows us to put the dependent jar files into the `lib` directory. You do not want to put them in the same directory as the bundle itself because the bundle manager scans that directory for jar file bundles and it would have to check all the non-bundles if they were there. It does not scan subdirectories so putting them into a subdirectory called `lib` (or whatever name you prefer) is just fine.

But how do we get the jar files into `lib`?

```
<plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-dependency-plugin</artifactId>
 <version>2.8</version>
 <executions>
  <!-- Unpack header files and libraries for build -->
  <execution>
            <id>copy-dependencies</id>
            <phase>package</phase>
            <goals>
              <goal>copy-dependencies</goal>
            </goals>
   <configuration>
    <includeScope>runtime</includeScope>
    <excludeTransitive>false</excludeTransitive>
    <outputDirectory>${project.build.directory}/lib</outputDirectory>
   </configuration>
  </execution>
 </executions>
</plugin>
```

This will copy all dependencies into the `lib`. It only copies `compile` dependencies (and their dependencies etc), it does not copy `test` and `provided` dependencies.

Once this is done you can copy the bundle jar file and the `lib` into the bundles directory and the dependencies will be loaded along with the bundle. It is okay to have just one `lib` directory shared by several bundles, though it may be hard to work out which dependencies are still being used as bundles are added and removed over time.

There is a limitation here in that your `lib` directory must be a subdirectory of your bundles directory. The bundle manager assumes this and will fail to load libraries from elsewhere. Also we have noticed some possible confusion arises when various Spring libraries are in both the parent and the bundle. You can get situations where versions mismatch and the bundle creates a class which it validated against the parent's version of a different Spring library and that throws an exception. In general try and keep the files in the `lib` directory to a minimum and use `provided` a lot when you specify bundle dependencies.

Also note that when you are loading bundles from the `WEB-INF/bundles` directory (*4.3*) you cannot use a `lib` directory, it will be ignored and the dependencies won't be loaded.

4.6. 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.

4.7. Selecting the right bundle

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.

```
<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>
```

Now you can examine the bundles like this:

```
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 with the version, in this case it would contain `mybundle-2.0`

For proxying if you don't pick a bundle it will default to the last bundle that registered with the bundle manager. That is probably the latest one and 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 stic 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.

4.8. 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.

4.9. 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.

4.10. Managing Session Beans inside Bundles

There's an interesting problem that can come up when you need to define a session bean inside a bundle. It relates to the way bundles are loaded and the way Spring handles session beans. As such it gets a little complex but the good news is this hardly ever comes up, so don't sweat too much over it.

Spring's session beans always assume there is a web session available but bundles are loaded *outside* the web session, and if we add the session scope to them Spring tells us that it doesn't have a session and dies.

But, actually it is not quite that simple. Spring defers the instantiation of the session beans until it has to. So it only dies if the bundle forces that bean to be instantiated up front. So the trick is to do two things:

- Make sure that every reference to the bean is scoped as session, that way Spring will know they are all references to the same bean.
- Make sure the bean is not instantiated until there is a web session available. That can be in the AppFactory.createApp(Blackboard) method which is not invoked until the bundle is bound by the application, and that doesn't normally hapen until there is a session.

So you can declare session beans anywhere. In practice there are three distinct xml files that define Spring beans in a normal web application.

- The bundle's own xml file.
- The applicationContext.xml file loaded by Spring's ContextLoaderListener.
- The session level application file loaded by one of the Vaadin application's init methods (assuming you are using the VaadinSupport approach).

You can define session beans in any of these like so:

```
<bean id="permissionManager"
 class="nz.co.senanque.vaadinsupport.permissionmanager.PermissionManagerImpl"
 scope="session"/>
<bean id="fieldFactory" class="nz.co.senanque.vaadinsupport.FieldFactory"
 scope="session"/>
<bean id="hints" class="nz.co.senanque.vaadinsupport.HintsImpl"
 scope="session"/>
```

These are three session beans defined in the session level application file. In this file every bean is effectively a session bean, but Spring doesn't know it and doesn't actually tie them to the http session in the same way. For these particular beans, though, we really want Spring to know they are session beans so we scope them.

The session level application file is never loaded until there is a web session so Spring will be fine with this.

Now, in our bundle context file we have this:

```
<bean id="permissionManager"
 class="nz.co.senanque.vaadinsupport.permissionmanager.PermissionManagerImpl"
 scope="session"/>
<bean id="fieldFactory" class="nz.co.senanque.vaadinsupport.FieldFactory"
 scope="session"/>
<bean id="hints" class="nz.co.senanque.vaadinsupport.HintsImpl"
 scope="session"/>
```

Same thing, right? Yes, but because these are explicitly Session scoped Spring will deliver the same bean. So multiple sessions talking to the same bundle will see different instances of the bean, ie their own instance. And they will reuse the instance defined in the session level application file.

At the bundle level these three beans are injected into this bean:

```
<bean id="maduraSessionManager"
 class="nz.co.senanque.vaadinsupport.application.MaduraSessionManager"
 scope="session"/>
```

This is another session scoped bean. The bean is instantiated when the following is executed:

```
m_maduraSessionManager =
 m_beanFactory.getBean("maduraSessionManager",MaduraSessionManager.class);
```

That code is from AppFactory.createApp(Blackboard) in the bundle. So Spring will defer instantiation until then, at which point it will instantiate and auto wire the dependent beans. But by then, of course, it has a web session underway and the session scope requirements are fulfilled, no problem.

Because these beans were defined as session scoped beans in the session level file Spring will find them and reuse them in the bundle, except for maduraSessionManager which it will create for this session.

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. Eclipse Setup

Eclipse will show some of the test XML configuration files to be in error because it fails to find the XSD file they depend on. This is easily fixed. Use the XML Catalog setting in Eclipse (Window->preferences->XML->XML Catalog) and create a schema location entry for key: http://www.senanque.co.nz/madura/bundle/MaduraBundle.xsd that points to MaduraBundle/src/main/resources/nz/co/senanque/madura/bundle/spring/MaduraBundle.xsd

C. Release Notes

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