MaduraBundle

User Guide

Document Revision: Revision: 48

Document Revision: Revision: 48

Table of Contents

1. Change Log

2. References

[1] Apache Licence 2.0
[2] Spring Framework
[3] OSGI
[4] Vaadin
[5] pizzaOrderBundle
[6] maduraBundle
[7] perspectivesManager
[8] Spring Properties

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 [2] 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 *8.5*. 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 *8.6*

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[6] then, for something more like an application you should look at pizzaOrderBundle[5] which is a working example of a bundle that plugs into the perspective manager application[7]. 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. Using Bundles

We deploy one jar file per bundle. The jar file contains classes and resources and one or more Spring contexts, either specified by annotations or XML. 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.

Let's assume you have a file called Config.java that is annotated with `@Configuration` along with several beans defined by `@Bean` or one of the `@Component` variants. This is all quite ordinary for Spring contexts, nothing new here at all.

The thing that is new is in the manifest file.

```
Manifest-Version: 1.0
Build-Jdk: 1.7.0_67
Built-By: roger
Bundle-Version: 4.2.0
Bundle-Name: my-bundle-name
Created-By: Apache Maven 3.2.1
Bundle-Description: Example Bundle
Bundle-Class: nz.co.senanque.madura.test.Config
Archiver-Version: Plexus Archiver
```

The crucial thing here is the Bundle-Class entry which specifies the Config class we mentioned earlier. To build your bundle using maven you need to specify these manifest entries 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>${user.name}</Built-By>
    <Bundle-Description>Test Bundle Maven</Bundle-Description>
    <Bundle-Name>${project.artifactId}</Bundle-Name>
    <Bundle-Version>${project.version}</Bundle-Version>
    <Bundle-Class>nz.co.senanque.madura.test.Config</Bundle-Class>
   </manifestEntries>
  </archive>
 </configuration>
</plugin>
```

Those manifest entries are copied into the Spring environment so if you have added a `PropertySourcesPlaceholderConfigurer` bean to your Config class you can inject those values into the other beans in your bundle.

The other thing to do in your Maven build is ensure there is a copy of the BundleRootImpl.class in the resulting jar file. We can't use the one in the application because we need it to be loaded by the bundle class loader, not the application class loader.

You don't have to worry about class loaders though. Just add this 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>
```

And you're done, you have your bundle. Now let's look at your main application, the thing that calls the bundle.

Just like in the bundle you define a Spring context in your application by adding a class that looks like this:

```
@Configuration
@EnableBundles
@ComponentScan(basePackages = {
  "nz.co.senanque.madura.bundle", "nz.co.senanque.madura.test"})
@PropertySource("classpath:config.properties")
public class SpringConfiguration {
...
```

These annotations, except for `@EnableBundles` are all documented in detail in the Spring docs but, briefly, the `@Configuration` declares this java class as a configuration class where beans are declared, `@ComponentScan` specifies which packages you want scanned for annotated classes to be made into beans. You must include `nz.co.senanque.madura.bundle`. The `@EnableBundles` adds some extra logic to the scan, and this is the extra bit that Madura Bundles needs.

That extra bit allows you to annotate your bundle interface classes like this:

```
@BundleInterface("TestBean")
public interface TestBean
{
...
}
```

With that annotation the scanner will locate the interface and generate a proxy bean ready for dynamic linking to the bundle. This is where the magic of dynamic proxies happens. In your main application you defined an annotated interface (actually you probably put it into a maven dependency) and in each of your bundles you defined an implementation of that interface. Once you select which bundle you want the application to use at any instance, the implementation in that bundle becomes active. Other than actually selecting the bundle the application does not need to know it is using a bundle at all. It just needs to know about the interface. You can have multiple annotated interfaces per bundle, of course, because you have multiple beans per bundle.

There is a little bit of configuration needed for the bundle manager and in the above example this is in the config.properties file that looks like this:

```
nz.co.senanque.madura.bundle.spring.BundleManagerFactory.directory=./
target/bundles
nz.co.senanque.madura.bundle.spring.BundleManagerFactory.type=impl
nz.co.senanque.madura.bundle.spring.BundleManagerFactory.time=-1
```

The values shown here are actually the defaults, except for the directory which defaults to a null string.

The directory holds your bundles and the time specifies (in milliseconds) how often to check for changes there (-1 means once only). The type parameter specifies the type of BundleManager to use. There are two different types: impl and web. The first is just the ordinary bundle manager and the web version allows you to put bundles into the WEB-INF/bundles directory in your war file. This is scanned just once at startup. You might wonder what possible use this is because if you have embedded a bundle in your war file you might as well have used a normal jar file dependency right? Yes, indeed. But it does come up sometimes when you want to put together a simple demo and not have to worry about deploying the bundles in a separate directory.

This configuration automatically arranges for an interface bean named bundleRoot to be created and this maps to the bundle root of the current bundle. That means you can inject it into your application code and examine it like this:

```
@Autowired BundleRoot bundleRoot;
...
String currentBundleName = bundleRoot.getName();
...
```

That will get you the name of the *current* bundle. If you switch bundles and call `bundleRoot.getName()` again you will find it returns a different name.

The last thing you have to know is how to select a bundle. To select a bundle you use the BundleManager bean (it was automatically defined in the above application configuration). Just auto wire it into a class in the usual way and call one of the following methods:

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.
setBundle(BundleVersion bv)	Set the current bundle to the one specified in the bv.

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 by name strings, or you can get the `BundleVersion` from the `bundleRoot` and pass that. 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 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.

`BundleManager` also has these two methods:

reserveBundle(BundleVersion bv)	Reserve the bundle to prevent it from being removed.
releaseBundle(BundleVersion bv)	Release the bundle allowing removal.

This allows an application to declare when it is using the specified bundle and version. Applications may be using more than one bundle, flicking between them as necessary. During that time the bundles being used must not be discarded by the bundle manager, and these two methods tell the bundle manager that they are in use.

The actual contents of your bundle can be anything you like, of course, but you also need to define one or more interfaces to be shared with the calling application. These are the interfaces you annotate with `@BundleInterface` and have the application scan. Naturally these interfaces must be in both application and bundle code for them to compile. But there is a trap here. The interface code can be compiled into the application but *not* in any bundle. The reason for this lies in the way the classloader works and we will explain that in a moment. The vital point to make here is that you will almost certainly want to put the interfaces into their own jar file or maven project. The application can have a maven compile dependency on that interface project but the bundles must have a maven *provided* dependency on it. That ensures the interface classes are not included in the bundle run time dependencies.

If you are using jar files rather than maven just do not include the interface jar file in the bundle dependencies.

There is more information on configuring dependencies in *5*

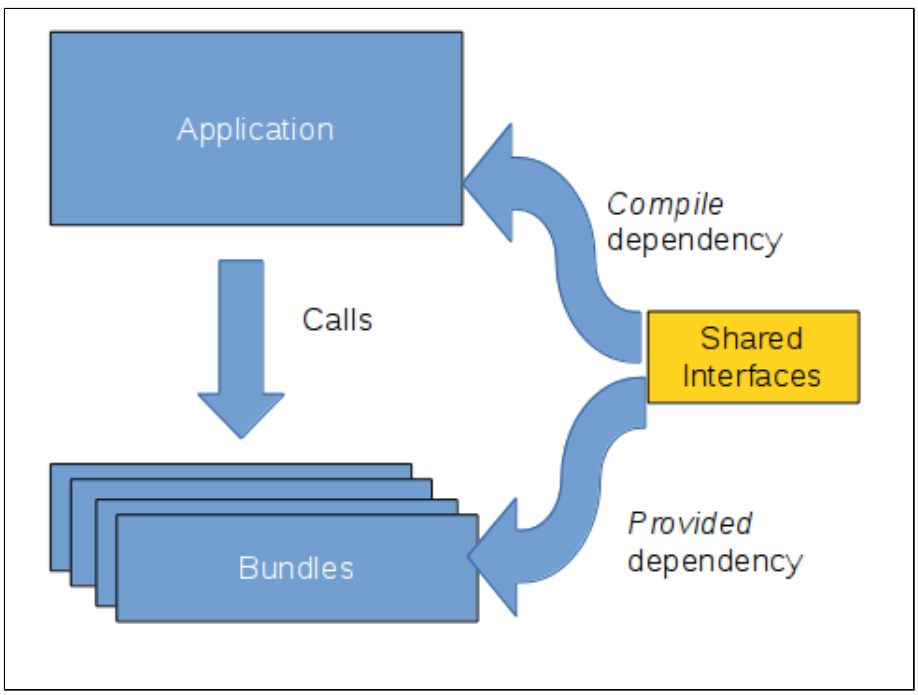

Figure (1) Interface Dependencies

Now, why is this important? The bundle classloader loads classes from the bundle in preference, and your application uses the normal class loader. If you have an interface `MyInterface` the application will load it from the application classloader, of course. The bundle will look for the same class in the bundle class loader. If it finds it there you will see strange errors suggesting that the implementation class of `MyInterface` does not conform to the `MyInterface` specified by the application. The two class definitions are not the same and the JVM complains. If, however, the bundle classloader does not find the `MyInterface` class it will then search the application classloader and it will find it there. In that case the two will match, eliminating the errors.

This only matters for interfaces shared across the aplication/bundle boundary. Other interfaces used by the bundle that the application does not see are no concern here. In practice this affects a very specific set of interfaces you have full controll over.

5. Managing Bundles

5.1. Selecting Bundles

The simplest way to make your bundles available to your application is to just copy their jar files into the directory specified by `nz.co.senanque.madura.bundle.spring.BundleManagerFactory.directory`. You can copy bundles into the directory while the application is running and if you have specified a value (in milliseconds) for `nz.co.senanque.madura.bundle.spring.BundleManagerFactory.time` then the directory will be scanned that often for new bundles. A common scenario is for the application code to specify only the bundle name, not the version when it selects that bundle. So when you copy a new version of the bundle into the directory the next select will find the latest version.

But sometimes you want to lock on to a specific bundle, for example if you have used the bundle to produce a quote for a customer and you want that quote to be stable for a time so you want to go back to the exact same bundle to redisplay it or recalculate it. In that case you can use the bundleRoot bean (again, just auto wire it into your application) to find out the current name and version of the bundle you are using, store that information in the quote and use it to select the bundle again when the quote is retrieved later.

5.2. Maven-based bundles

It is often useful to store the bundles in maven rather than a directory, but we still need a directory to tell the bundle manager what Maven artifacts we want to use. The bundles directory (defined by `nz.co.senanque.madura.bundle.spring.BundleManagerFactory.directory`) can have .jar files but it call also have .bundle files. They look like this:

```
Bundle-Artifact=nz.co.senanque:madura-testbundle:4.2.0
```

You would probably name a file like this `madura-testbundle:4.2.0.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.

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.repo** 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.

Your maven based bundles can have dependencies, of course, and these will be pulled from maven as well. To avoid having unnecessary duplicate libraries try and ensure all the dependencies are also dependencies of the main application, and make the bundle dependencies scope provided.

5.3. Selecting a Bundle based on Criteria

You can add other entries to the manifest file (or the .bundle file) and use them as bundle selection criteria. For example say your manifest file looks like this:

```
Manifest-Version: 1.0
Build-Jdk: 1.7.0_67
Built-By: roger
Bundle-Version: 4.2.0-SNAPSHOT
Bundle-Name: bundle-maven2
```

```
Created-By: Apache Maven 3.2.1
Bundle-Description: Test Bundle Maven2
Bundle-Class: nz.co.senanque.madura.bundle.Config
Archiver-Version: Plexus Archiver
from-date: 01-Jan-2011
to-date: 01-Jul-2011
```

We added two entries at the bottom: from-date and to-date. Now how do we use them?

```
for (BundleRoot br:bundleManager.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;
 }
}
```

This example loops through all available bundles and accesses the extra criteria in each to see if it is the bundle we want. If you are using Maven based bundles you can add the extra criteria to a .bundle file. The .bundle file entries will override the entries in the manifest file if they overlap.

6. XML Configuration

If you prefer XML rather than annotations to build your Spring contexts this is what the XML for a bundle might look like:

```
<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">

 <context:property-placeholder />

 <bean id="TestBean" class="nz.co.senanque.madura.bundle0.TestBeanImpl">
  <property name="content" ref="bundleName"/>
  <property name="resource" value="classpath:BundleResource.xml"/>
 </bean>
</beans>
```

You build this slightly differently:

```
<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>${user.name}</Built-By>
    <Bundle-Description>Test Bundle Maven</Bundle-Description>
    <Bundle-Name>${project.artifactId}</Bundle-Name>
    <Bundle-Version>${project.version}</Bundle-Version>
    <Bundle-Context>bundle-spring.xml</Bundle-Context>
   </manifestEntries>
  </archive>
 </configuration>
</plugin>
```

The only difference is that there is a `Bundle-Context` entry rather than a `Bundle-Class`. The bundle-spring.xml file should be in the top directory of the jar file.

To configure an application in XML use a context 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.madurasoftware.com/madura-bundle"
 xmlns:context="http://www.springframework.org/schema/context"
 xsi:schemaLocation="
http://www.springframework.org/schema/beans http://
www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/util http://www.springframework.org/
schema/util/spring-util.xsd
```

```
http://www.madurasoftware.com/madura-bundle http://www.madurasoftware.com/
madura-bundle.xsd
http://www.springframework.org/schema/context http://
www.springframework.org/schema/context/spring-context.xsd
http://www.springframework.org/schema/aop http://www.springframework.org/
schema/aop/spring-aop.xsd">

  <context:annotation-config/>
  <context:component-scan base-package="nz.co.senanque.madura.bundle"/>
  <bundle:component-scan base-package="nz.co.senanque.madura.test"/>
  <context:property-placeholder location="classpath:config.properties"/>
  <bean id="propertyConfigurer" class=

 "org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
   <property name="location" value="classpath:config.properties"/>
  </bean>

</beans>
```

This configures all the annotated classes, but we don't need the java class with `@Configuration`. The two `component-scan` entries are doing two different jobs. The first is Spring's usual scan for `@Component` beans. The second is scanning for interfaces to turn into proxy beans.

You can go a level deeper and avoid annotations altogether, but it is doubtful you want to read about it. Take a look at the tests in the madura-objects projects for examples.

7. Bundle Dependencies

Often your bundle code will have dependent code libraries it needs in order to run. There are several approaches to managing this.

The first way is to use maven based bundles and then just let maven handle the dependencies. This will work, but it is not quite optimal because there are probably duplicate libraries across multiple bundles and your application. These will increase the memory footprint unecessarily. The simple answer to this is to make sure as many bundle dependencies as possible are scoped `provided` in the bundles' pom files, and make sure these are also dependencies for the application.

If you are not using maven based bundles you can use a `Class-Path` entry in the bundle's manifest file and copy the relevant dependencies into a subdirectory under your bundles directory. This is harder to organise and you don't solve the memory footprint problem, but it does work. You only need to add entries for dependencies that are not also dependencies of the application.

You use the `Class-Path` entry 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 assumes the there is a directory called lib2, a subdirectory of your bundles directory, and that you copied the relevant jar files into it. This feature is of limited use though. For example it does not work with the `WEB-INF/bundles` directory. If your bundle has dependencies that are not in the main application consider using Maven based bundles.

8. Advanced Topics

8.1. JMX

The Bundle Manager is exposed to JMX for several operations, notably forcing it to scan for new bundles. Details of configuring a JMX in an application is beyond the scope of this document but you mostly need to just add the following to your Spring context:

```
<bean id="mbeanServer" class="java.lang.management.ManagementFactory"
 lazy-init="false" factory-method="getPlatformMBeanServer"/>
<context:mbean-export server="mbeanServer"/>
```

8.2. Alternate Timer configurations

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

```
@Component
public class MyScheduler {

 @Scheduled(fixedDelay=10000)
 @Autowired BundleManager bundleManager;

 public void runScheduler() {
  bundleManager.scan();
 }
}
```

Or in XML:

```
<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.

8.3. 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 annotate your beans like this:

```
@Component("exportBean")
@BundleExport
public class TestExportBean {
....
```

The scanner will identify the export beans and tell the bundle manager about them. You can auto wire the exported beans into your bundled classes as if they were defined locally.

You can get exactly the same effect using the `@Bean` annotation in a `@Configuration`.

```
@Bean
@BundleExport
public TestExportBean2 getTestExportBean2() {
....
```

That allows you to export a bean that uses a class from a library etc, that is not convenient to annotate directly.

If you are using XML then you also have two options for exporting:

```
<bean id="TestBean2" class="nz.co.senanque.madura.testbeans.TestBeanImpl"
 bundle:export="true">
...
</bean>
```

This does exactly the same as the annotations. The last way to export beans is to add them to the bundle manager definition like this:

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

This just injects a map of exported beans into the bundle manager. It does just a little more than the other methods because it allows you to change the name of the exported bean. There is actually only one bean, defined as `exportBeanExample` in the application context. The bundle contexts, however, refer to this bean as `exportBean`. In practice you don't normaly have any need for this extra name.

8.4. Configuring Properties

In several examples you have seen a file called `config.properties` being loaded by the Spring context eg:

```
<context:property-placeholder location="classpath:config.properties"/>
```

This is standard Spring. It loads the properties file in such a way that Spring can use it in various ways that are better described here [8]

By default these application level properties are not passed to the bundles. The bundle loads its own properties. But sometimes you want the application properties to be used in the bundle. To do this you need to add something to your bundle manager configuration:

```
<context:property-placeholder location="classpath:config.properties"/>
<util:properties id="ep" location="classpath:config.properties"/>
<bean id="bundleManager"
 class="nz.co.senanque.madura.bundle.BundleManagerImpl">
 ...
 <property name="exportedProperties" ref="ep"/>
</bean>
```

This injects the properties file into the bundle manager. Loading it with the `util:properties` tag is not the same as loading it with the `context:property-placeholder` tag, so you need both. The result is that all bundles will be able to use these properties in the usual ways.

That is the way to do it when configuring with XML. When using @Configuration the situation is different. Load the application properties like this:

```
@Configuration
@EnableBundles
@ComponentScan(basePackages = {
  "nz.co.senanque.madura.bundle", "nz.co.senanque.madura.test"})
@PropertySource("classpath:config.properties")
public class MyConfig {

 @Bean
 public static PropertySourcesPlaceholderConfigurer propertyConfigInDev()
 {
  PropertySourcesPlaceholderConfigurer ret = new
 PropertySourcesPlaceholderConfigurer();
  return ret;
 }
 ...
```

This is ordinary Spring, but there is a hidden difference from the XML. These properties *will* be passed to the bundles, which is probably what you want. Notice the static bean, you absolutely need this for the properties to work.

In both case the properties are passed to the bundle and they *override* any local properties of the same name the bundle might create. Sometimes you do not want this, you want locals to override application properties. In XML this is simple. In the bundle you just load your local properties like this:

```
<context:property-placeholder local-override="true"
 location="classpath:config.properties"/>
```

The official equivalent in @Configuration is to do this:

```
@Configuration
@EnableBundles
@ComponentScan(basePackages = {
  "nz.co.senanque.madura.bundle", "nz.co.senanque.madura.test"})
@PropertySource("classpath:config.properties")
public class MyConfig {

 @Bean
 public static PropertySourcesPlaceholderConfigurer propertyConfigInDev()
 {
  PropertySourcesPlaceholderConfigurer ret = new
 PropertySourcesPlaceholderConfigurer();
  ret.setLocalOverride(true);
  return ret;
 }
 ...
```

This, however, does not actually work, so we provide a different @PropertySource class which can be used like this:

```
@Configuration
@ComponentScan(basePackages = {
  "nz.co.senanque.madura.b2"})
```

```
@PropertySource(value="classpath:configb.properties", localOverride=true)
public class Config {
 ...
```

This PropertySource is `nz.co.senanque.propertysource.PropertySource` rather than the Spring one. And it does not require the static bean that Spring does. To use it you need to add a dependency to your project:

```
<dependency>
    <groupId>nz.co.senanque</groupId>
    <artifactId>madura-resource-loader</artifactId>
    <version>1.3.0</version>
</dependency>
```

Add this to your application and as scope=provided to your bundles.

8.5. 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. It will be called when the list of bundles change.

8.6. 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.

8.7. Scoped Beans

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

```
@Component("myBean")
@Scope("session")
public class MyBeanImpl {
....
```

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.

When using bundles you often want to export these beans forthe bundles to use. This is simple to do:

```
@Component("myBean")
@Scope(value="session", proxyMode = ScopedProxyMode.TARGET_CLASS)
@BundleExport
public class MyBeanImpl {
....
```

There are some extra qualifiers on the `@Scope` to ensure Spring wraps the bean in a proxy, making it suitable to export to the bundles. and after that the bundles can use it like a normal export bean. One thing to watch with any session beans is that Spring defers instantiation of session beans until they are asked for because it needs a web session to store them against and there isn't one at startup. So be careful you don't force the session bean to be created early, for example by injecting it into some bean that uses it in an initialisation method.

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:

```
@Component("myBean")
@Scope("bundle")
public class MyBeanImpl {
....
```

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.

By default the scope relies on Spring's session id which assumes you are using Spring's MVC that records the id in a `ThreadLocal`. If you are using something else, such as Vaadin[4], you need a different way to get a session id. Just implement `nz.co.senanque.madura.bundle.spring.SessionIdProvider`, in a `@Component` annotated class, and put it into a package scanned by the application. We have already built one for Vaadin. It is used in [7].

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

4.5.0

Travis build.

4.4.0

Added JMX functions.

Changed the way to configure the scan timer to simplify it.

Added handling of local and external properties.

4.3.0

Double loading of bundles on startup: fixed.

Java 1.8.

4.2.0

Added XML export attribute to bean definitions.

Extended BundleScope session id provider to allow configurable mechanism.

Reworked the testing structure for more comprehensive tests.

Fixed problem with loading environment variables and @PropertySource files.

Fixed problem with getting classes as resources from the class loader.

Implemented annotation-based configuration.

Added automatic bundleRoot bean (when using annotation-based configuration).

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