Using a shared parent application context in a multi-war Spring application

Posted on June 11th, 2007 by Joris Kuipers in IOC Container, Spring, SpringSource.

Last month I gave a Core Spring training in Turkey. At the end of the course I discussed the architecture for an application that some of the participants were going to build after completing the course. This application would consist of an ear file with several war files inside, and the question came up if it was possible to define a single ApplicationContext that could be used as a shared parent to the WebApplicationContexts of all war files. This context would hold bean definitions for services, DAOs and other beans that were not specific to a single web module.

Actually, Spring makes it very easy to do this, but neither the course nor the reference manual explain in detail how to use this feature from your web application. I therefore wrote a short sample app that illustrates how this works, which I will discuss here in my first blog entry.

The ContextLoader and SingletonBeanFactoryLocator classes

In a typical Spring web application, you use a ContextLoaderListener (or, if you're using a Servlet 2.2 / 2.3 container, a ContextLoaderServlet) to bootstrap a WebApplicationContext. You configure the ContextLoader used by this class through context parameters in your web.xml. If you've used this, you're probably familiar with the `contextConfigLocation` parameter, that let's you specify which files make up the WebApplicationContext to construct.

As it turns out, there is another parameter that you can use to get the desired functionality in a declaritive fashion: **`parentContextKey`**. Using this, you instruct the ContextLoader to use another class called ContextSingletonBeanFactoryLocator to search for a bean named by the value of `parentContextKey`, defined in a configuration file whose name matches a certain pattern. By default, this pattern is 'classpath*:beanRefContext.xml', meaning all files called beanRefContext on the classpath. (For a plain SingletonBeanFactoryLocator it's 'classpath*:beanRefFactory.xml')
This bean must be an ApplicationContext itself, and this context will become the parent context for the WebApplicationContext created by the ContextLoader. However, if this context already exists then that one will be used and no new context will be created (hence the name **Singleton**BeanFactoryLocator).

Let's see what this means: first, we need a separate jar in our ear that holds the code for the services, DAOs, etc. Inside this jar we place a beanRefContext.xml file that holds a single bean-definition for an ApplicationContext. Typically, this will be a ClassPathXmlApplicationContext. Then that bean definition will refer to one or more 'regular' bean configuration files that contain the service beans and other stuff to be used by the code from your war files; something like this:

```
1  <!-- contents of beanRefContext.xml:
2         notice that the bean id is the value specified by the parentContextKey param -->
3  <bean id="ear.context"
   class="org.springframework.context.support.ClassPathXmlApplicationContext">
4      <constructor-arg>
5          <list>
6              <value>services-context.xml</value>
```

```
        </list>
    </constructor-arg>
</bean>
```

Finally, we need to make this jar available on the classpath of the war files using the `Class-Path` entry in the MANIFEST.MF file of each war.

Why would you want to use this?

A simpler solution is to skip the `parentContextKey` and to load the shared bean definition files from each war using the `contextConfigLocation` parameter. This way, each war will have its own instance of each shared bean. For simple stateless beans, such as typical services, this is actually a fine solution.

However, instantiating a new instance of each shared bean for each war can have several drawbacks: a common example is creating a Hibernate SessionFactory. This is often an expensive process, so to prevent your startup time getting out of hand the described solution will ensure that this is done only once. Another advantage of having a single instance of your SessionFactory is that it can safely act as a second level cache: you don't want multiple copies of that hanging around in a single application!
In general, if you have stateful beans that should really be used as a singleton (in the Spring sense, i.e. one instance per application as opposed to per JVM) you should define them in a single context accessed by the other contexts.

The sample

I've included the sample, both as an ear file and as source. In order to upload the ear, I had to give it a .zip extension: please rename the file to .ear before deploying it!. The source is actually an Eclipse workspace, so you can easily import and review it (it requires WTP and is configured for Spring IDE). All Spring jars needed are included. After you've deployed the app, go to the URLs /web1 and /web2 to see the output of a servlet in the first and the second war file. The `toString()` of the service will prove that the wars really use the same instance of the shared service.

More info

The best info on this feature is in Spring's excellent API documentation: have a look at the JavaDoc for the ContextLoader.loadParentContext method and the SingletonBeanFactoryLocator class. These docs contain further info on how to configure your web.xml and how to write the beanRefFactory.xml.

Similar Posts

- SpringSource Application Platform Deployment Options
- Spring: the foundation for Grails
- Spring Integration in 10 minutes
- Green Beans: Putting the Spring in Your Step (and Application)
- Spring Dynamic Language Support and a Groovy DSL

Share this Post

TwitThis

Search