Test of generating a larger text

PDF generation

Version: 1.0

Author: Tommy Svensson

Table of Contents

1 JPA in OSGi in a portable way part 2

I previously said that it was "quite difficult" to make JPA available as a service. I have however found a solution that I thought I share.

The main problem is that you want the JPA service clients to be able to provide the META-INF/persistence.xml file and the JPA usage code while the service should be able to provide a configured EntityManager. Since each bundle in OSGi has its own ClassLoader the classpath of both the service provider and client have to be merged somehow for the JPA code to be able to see both the JPA implementation code provided by the service and the persistence.xml and the persistent entities provided by the client. This is what escaped me at first, but it turned out that I was dumb! I have done class loaders before and I was blind to what was in front of my eyes :-).

1.1 The OSGi bundle is hiding something!

The OSGi Bundle class includes the following 3 interesting methods:

• Class loadClass(String name); *URL getResource(String name);* Enumeration<URL> getResources(String name);
They happen to map quite well to the 3 methods of ClassLoader that class loader implementations should override (even though they don't exactly name match):

• Class findClass(String name); *URL findResource(String name);* Enumeration<URL> findResources(String name);
A Bundle is not a class loader, but you can make a class loader that uses the bundle to load classes and resources. In this case we actually want to make a class loader that wraps 2 bundles:

```
public MergingClassLoader extends ClassLoader {

    private Bundle bundle1;
    private Bundle bundle2;

    public MergingClassLoader(Bundle bundle1, Bundle bundle2) {
        this.bundle1 = bundle1;
        this.bundle2 = bundle2;
    }

    public Class findClass(String name) throws ClassNotFoundException {
        Class found = null;
        try {
            found = this.bundle1.loadClass(name);
        } catch (ClassNotFoundException cnfe) {
            found = this.bundle2.loadClass(name);
        }
        return found;
    }

    public URL findResource(String name) {
        URL resource = this.bundle1.getResource(name);
        if (resource == null) {
            resource = this.bundle2.getResource(name);
        }
        return resource;
    }

    public Enumeration<URL> findResources(String name) {
        Enumeration<URL> resources = this.bundle1.getResources(name);
        if (resources == null) {
            resources = this.bundle2.getResources(name);
        }
        return resources;
    }
}
```

We then pass both the JPA service provider bundle and the JPA using bundle into this classloader.

1.2 Next questions ?

So the next question becomes, what do we need to do to merge these ? Well, we need to get the Bundle objects for both. For our JPA providing service it is not a problem, it can get the Bundle object representing its own bundle. But for clients wanting to use the JPA service it not as trivial. In Apache Aries they have gone with a service API like this:

```
    void registerContext(String unitName, Bundle client, HashMap<String,Object>
properties)
```

This provides the persistence unit name (from META-INF/persistence.xml), the Bundle object representing the bundle having the persistence.xml file and properties for creating the EnitiyManagerFactory (in addition to what is in persistence.xml). This is what you need from the client side and makes the job easier for the service implementation, but personally I prefer another more client friendly solution using the extender pattern of OSGi. This of course requires a bit more work in the JPA service provider bundle.

We need to listen to bundle events which is done by implementing org.osgi.framework.BundleListener:

```
    public void bundleChanged(BundleEvent event) {
        Bundle bundle = event.getBundle();

        if (bundle.getEntry("META-INF/persistence.xml") != null) {

            switch (event.getType()) {
                case BundleEvent.STARTED:
                    addPersistenceProviderForBundle(bundle);
                    break;

                case BundleEvent.STOPPED:
                    removePersistenceProviderForBundle(bundle);
                    break;
            }
        }
    }
```

Now we also have our client Bundle objects! If we want to support several clients with different persistence units then we need to create an EntityManagerFactory for each bundle & persistence unit. Thereby we need to resolve the persistence unit used by the client bundle. To do this we need to read the META-INF/persistence.xml file. There are many solutions for reading XML. I used my own :-): XMLObjectBinder (xob.sf.net). This was done many years ago pre annotations. I am considering updating it. To read the persistence.xml we need the following interfaces:

```
    public interface Persistence extends XMLObject {
        public String getVersion();
        public Iterator<PersistenceUnit> getPersistenceUnits();
    }

    public interface PersistenceUnit extends XMLObject {
        public static final String XOM_PersistenceUnit = "persistence-unit";

        public String getName();
    }
```

And then we load the persistence.xml with the following code:

```
    URL persistenceXML = bundle.getEntry("META-INF/persistence.xml");
    XMLObjectBinder binder = Factory.createXMLObjectBinder(Persistence.class);
    binder.getXMLParser().setValidating(false);
    XMLUnmarshaller unmarshaller = binder.createUnmarshaller();
    InputStream persistenceXMLStream = persistenceXML.openStream();
    Persistence persistence = (Persistence)unmarshaller.unmarshal(persistenceXMLStream);
    try {persistenceXMLStream.close();} catch (IOException ioe){}
```

We are only interested in the name of persistence units the other parts of the persistence.xml we don't care about, which is why the defined XML API ignores the other parts too.

Then we need to loop through the persistence units and create a EntityManagerFactory for each and save them in a Map keyed on persistence unit for when the client comes and asks for it. To correctly instantiate our persistence provider (OpenJPA in this case) we need the class loader that we made above:

```
    MultiBundleClassLoader multiBundleClassLoader = new
MultiBundleClassLoader(this.thisBundle, bundle);
    Thread.currentThread().setContextClassLoader(multiBundleClassLoader);
```

Note that we set the new classloader as context classloader on the thread. This is required for it to be used by OpenJPA when we create the provider instance:

```
    PersistenceProvider persistenceProvider = new PersistenceProviderImpl();
    EntityManagerFactory emf =
persistenceProvider.createEntityManagerFactory(persistenceUnitName, props);
```

Now we have an EntityManagerFactory and can save it until the client comes and asks for it. From here there are many options on how to provide the service. I wont go any further here, but there are a few things to think on:

- The client needs to have the classloader (or at least its own bundle classloader) as context classloader when using the EntityManagerFactory and EntityManager. If you want to be nice to the client you provide your own implementation of the EntityManagerFactory and EntityManager interfaces that sets a correct context classloader, forwards the call and then restores the context classloader, in which case the client doesn't have to bother.
- You have to decide who takes responsibility for EntityManagerFactory and EntityManager. Personally I choose to let the service be responsible for EntityManagerFactory and manage its life, creating and closing it, while the client is responsible for each created EntityManager.

Since OSGi allows services to come and go I also decided to not just provide the EntityManagerFactory right off. I created the following API:

```
    public interface APSJPAService {
        APSJPAEntityManagerProvider initialize(BundleContext bundleContext, String
persistenceUnitName, Map<String, String> props) throws APSResourceNotFoundException;

        public static interface APSJPAEntityManagerProvider {
            public boolean isValid();
            EntityManager createEntityManager();
            EntityManagerFactory getEntityManagerFactory();
        }
    }
```

This allowed me to handle redeployment of the service. Clients can then do a support method:

```
public EntityManager createEntityManager() {
    if (this.emp == null || !this.emp.isValid()) {
        this.emp = this.jpaService.initialize(. . .);
    }
    return this.emp.createEntityManager();
}
```

This will handle the JPA service having been restarted. The service marks the previous instance as invalid on shutdown. If you are using the standard OSGi ServiceTracker (which I really don't like!) the above example code will have to be slightly more complex.

1.3 An alternative tracker

I have however made my own service tracker that have a getWrappedService() method on it. This will return a java.lang.reflect.Proxy implementation of the service that uses the tracker to get a service instance, calls the service, releases the service instance, and returns whatever the service call returned. It also has a timeout and wait for that amount of time for a service to become available before failing. That makes it very safe to do the code in the example above. It is part of what I call Application Platform Services or APS for short. This is a set of useful OSGi services and administration WABs intended for simplicity of use (and thereby not perfect for all situations, but good enough for many). I will shortly be releasing the first version of this as open source. It is a "smörgåsbord" (yes, I'm Swedish) of services and utils of which you can pick whichever you want. It started out as a more advanced structured configuration service with a WAB providing config editing, but grew to be much more over time :-)

--Tommy