AvalonAndOtherShinyContainers

Avalon-Magic, or how to provide container interop

Some code is in development in the avalon-sandbox module that should allow better integration between avalon and other container developments like PicoContainer. At the moment it has a single class Avalon2PicoAdapter, that can be used in a container to host arbitrary pico components. Somewhat like so:

```
Object comp = null;
if( isAnAvalonComponent(profile) )
{
  // act 'normally' to create the instance
  comp = profile.getImplementationClass().newInstance();
}
else if( isAPicoComponent(profile) )
{
  comp = AvalonInvocationHandler.getProxy( profile.getImplementationClass() );
}
ContainerUtil.enableLogging( comp, profile.getLogger() ); // etc etc
```

Until this is added to all containers, you can do this yourself, manually:

```
MyComponent comp = (MyComponent)
         AvalonInvocationHandler.getProxy( MyComponentImpl.class );

// the container should properly set up your logger, configuration and
// all other dependencies for you
myAvalonContainer.add( comp );
```

(provided, of course, the container allows you to add components in a fashion like this, which is, unfortunately, not so trivial for current containers).

How to make your pico-component avalon-compatible (or vice versa) by hand

Start with interfaces as normal

```
public interface Engine {
 void runEngine();
}
public interface Persistor {
 void persist(String key, Object data);
}
```

Write your implementations as normal

```
public class PersistorImpl
{
 void persist( String key, Object data ) { ;/* ... */ }
}
public class EngineImpl implements Engine
{
 private Persistor m_p;
 private String m_pk;

 public EngineImpl( Persistor p, String pKey )
 {
   setPersistor( p );
   setPersistorKey( pk );
 }

 public void runEngine()
 {
   getPersistor().persist( getKey(), new Object() );
 }

 protected void setPersistor( Persistor p ) { m_p = p; }
 protected void setPersistorKey( String pk ) { m_pk = pk; }
 protected Persistor getPersistor() { return m_p; )
 protected String getKey() { return m_pk; )
 protected Object getPersistableObject() { return new Object(); )
}
```

Either add avalon lifecycle directly...

```
public class EngineImpl implements Engine,
   Servicable, Configurable // AVALON COMPONENT SUPPORT
{
 /* ...same implementation here... */

 //
 // AVALON COMPONENT SUPPORT
 //
 public EngineImpl() {}

 public void service( ServiceManager sm )
     throws ServiceException
 {
   Persistor p = (Persistor)sm.lookup( Persistor.class.getName() );
   setPersistor( p );
 }

 public void configure( Configuration c )
     throws ConfigurationException
 {
   String pk = c.getAttribute( "persistorKey" );
   setPersistorKey( pk );
 }
}
```

...or add that support in an extended class...

```
public class AvalonEngine extends EngineImpl,
   Servicable, Configurable // AVALON COMPONENT SUPPORT
{
 //
 // AVALON COMPONENT SUPPORT
 //

 public AvalonEngine()
 {
   super(); // one important change required in your
            // base 'EngineImpl' class
            // -- you need this default constructor!
            // The same applies to webwork and spring,
            // so as a convenience always include it
            // for portable components
 }

 public void service( ServiceManager sm )
     throws ServiceException
 {
   Persistor p = (Persistor)sm.lookup( Persistor.class.getName() );
   setPersistor( p );
 }

 public void configure( Configuration c )
     throws ConfigurationException
 {
   String pk = c.getAttribute( "persistorKey" );
   setPersistorKey( pk );
 }
}
```

Hot to add support for Loom, Plexus, Turbine, Cocoon, James, Keel, ...

The avalon semantics are used and supported by a growing number of other containers and frameworks. If you support them, it becomes easy to deploy your components in (among others):

- Loom – http://www.jcontainer.org/
- Plexus – http://plexus.codehaus.org/
- Turbine – http://jakarta.apache.org/turbine/ (work in progress)
- Cocoon – http://cocoon.apache.org/
- James – http://james.apache.org/
- Keel – http://www.keelframework.org/

How to add spring framework compatiblity

```
public class EngineImpl implements Engine
{
 /* ...same implementation here... */

 //
 // SPRING COMPONENT SUPPORT
 //

 // empty constructor needed again!
 public EngineImpl() {}

 // and these need to be public!
 public void setPersistor( Persistor p ) { m_p = p; }
 public void setPersistorKey( String pk ) { m_pk = pk; }
}
```

and all you need now is your ugly xml descriptor file.

How to add HiveMind compatibility

Just like Spring, HiveMind doesn't really require your component to follow any kind of pattern, beyond having a specific service interface, and a bean that implements the interface. HiveMind can configure the properties of your service beans, so you need at least mutator methods. IoC is achieved by having HiveMind set a property of one service to a reference to another service.

Service implementations do need an empty constructor. HiveMind also requires your components be multithread-safe and assumes they will be 'singletons' (though additional service models are being added beyond the singleton model).

```
public class EngineImpl implements Engine
{
 private Persistor m_persistor;

 public void setPersistor(Persistor p)
 {
   m_persistor = p;
 }

 //
 // HIVEMIND COMPONENT SUPPORT
 //


 // thread safety is mandatory!
 public synchronized void runEngine()
 {
   /* ...same implementation here... */
 }
}
```

and all you need now is your beautful, elegant, expressive (original author said 'ugly') xml descriptor files.

Note: Most often, methods don't need to be synchronized to be thread safe, they just need to avoid making use of instance variables.

HiveMind is "wide open" compared to Avalon. Any service can get a reference to any other service; any module can extend any other module. A lot of HiveMind is built **in** HiveMind; people often miss the IoC stuff because it's accomplished using a HiveMind service, rather than some kind of native configuration.

How to add WebWork2/XWork compatibility

Write your XXXAware interfaces

```
public interface PersistorAware
{
 public void setPersistor( Persistor p );
}
public interface PersistorKeyAware
{
 public void setPersistorKey( String pk );
}
```

Add support in a subclass...

```
public class WebworkEngine extends EngineImpl
   implements PersistorAware, PersistorKeyAware
{
 //
 // WEBWORK COMPONENT SUPPORT
 //

 // empty constructor needed again!
 public EngineImpl() {}

 // and these need to be public!
 public void setPersistor( Persistor p ) { super.setPersistor( p ); }
 public void setPersistorKey( String pk ) {
     super.setPersistorKey( pk ); }
}
```

...or in the base class

```
public class EngineImpl implements Engine,
   implements PersistorAware, PersistorKeyAware // WW2 SUPPORT
{
 /* ...same implementation here... */

 //
 // WEBWORK COMPONENT SUPPORT
 //

 // empty constructor needed again!
 public EngineImpl() {}

 // and these need to be public!
 public void setPersistor( Persistor p ) { m_p = p; }
 public void setPersistorKey( String pk ) { m_pk = pk; }
}
```

Additional limitations

To run your components in Avalon-Phoenix (and/or HiveMind and/or Loom), they need to be threadsafe, and they need to operate as pseudo-singletons (one-instance-per-classloader). To run them in Avalon-ECM and cocoon, you need to then mark them as threadsafe:

```
public class EngineImpl implements Engine
   Servicable, Configurable, // support avalon
   implements PersistorAware, PersistorKeyAware, // support webwork
   implements ThreadSafe // support avalon's venerable ECM (and cocoon)
{ /* ... */ }
```

This is a good idea because singletons are somewhat of an LCD in many systems. For example, reverting to old-turbine-style singleton factories is then an option:

```
public class EngineFactory
{
  private static Engine m_instance;

  public static Engine getInstance()
  {
    if( m_instance == null )
    {
      Persistor p = PersistorFactory.getInstance();
      String pk = System.getProperty("persistence.key");
      m_instance = new EngineImpl( p, pk );
    }
    return m_instance;
  }
}
```

if you ever need to go back to such a horrid way of life.

Bringing it all together

So you want a component that is pico-compatible, avalon-compatible, webwork-compatible, and spring-compatible, and you don't mind typing? Here's what you do:

step 1: Work Interfaces

```
public interface Engine {
 void runEngine();
}
public interface Persistor {
 void persist(String key, Object data);
}
```

step 2: Awareness Interfaces

```
public interface PersistorAware
{
 public void setPersistor( Persistor p );
}
public interface PersistorKeyAware
{
 public void setPersistorKey( String pk );
}
```

step 3: Implementation

```
/**
 * @avalon.component version="1.0" name="persistor" lifestyle="singleton"
 * @avalon.service type="Persistor" version="1.0"
 */
public class PersistorImpl
    implements ThreadSafe // support Avalon-Fortress and Avalon-ECM
{
 synchronized void persist( String key, Object data ) { /* ... */ }
}
/**
 * @avalon.component name="engine" lifestyle="singleton" version="1.0"
```

```
 * @avalon.service type="Engine" version="1.0"
 */
public class EngineImpl implements Engine
   Servicable, Configurable, // support avalon
   ThreadSafe, // support Avalon-Fortress / Avalon-ECM
   PersistorAware, PersistorKeyAware // support webwork
{
 private Persistor m_p;
 private String m_pk;

 /**
  * be sure to initialize all dependencies before calling
  * any work interface methods if you use this constructor!
  */
 public EngineImpl() {}
 public EngineImpl( Persistor p, String pKey )
 {
   setPersistor( p );
   setPersistorKey( pk );
 }

 //
 // Work Interface
 //

 public synchronized void runEngine()
 {
   getPersistor().persist( getKey(), new Object() );
 }

 //
 // Dependency setup / retrieval
 // (ie your average javabean getter and setter)
 //

 public synchronized void setPersistor( Persistor p ) { m_p = p; }
 public synchronized void setPersistorKey( String pk ) { m_pk = pk; }
 protected synchronized Persistor getPersistor() { return m_p; )
 protected synchronized String getKey() { return m_pk; )
 protected synchronized Object getPersistableObject() { return new Object(); )

 //
 // Avalon-framework support
 //

 /**
  * @avalon.dependency type="Persistor"
  */
 public void service( ServiceManager sm )
     throws ServiceException
 {
   Persistor p = (Persistor)sm.lookup( Persistor.class.getName() );
   setPersistor( p );
 }

 public void configure( Configuration c )
     throws ConfigurationException
 {
   String pk = c.getAttribute( "persistorKey" );
   setPersistorKey( pk );
 }
}
```

step 4: container-specific files

Before you can use this component, you need to write or generate some container-specific files.

4.a) Avalon-Merlin

Auto-generate the <Classname>.xinfo files using the avalon-meta plugin for maven:

```
<preGoal name="java:compile">
  <attainGoal name="avalon:meta"/>
</preGoal>
```

Then, create a BLOCK-INF/block.xml file:

```
<container name="interop-demo">
  <classloader>
    <classpath>
      <repository>
        <resource id="avalon-framework:avalon-framework-impl" version="4.1.5"/>
      </repository>
    </classpath>
  </classloader>

  <component name="my-persistor" class="PersistorImpl"/>
  <component name="my-engine" class="EngineImpl"/>
</container>
```

Then, create a <Classname>.xconfig file for each component that needs configuration, in this case EngineImpl.xconfig:

```
<configuration>
  <persistorKey>mykey</persistorKey>
</configuration>
```

Merlin supports many more advanced configuration options including configuration overrides, component repositories, etc etc, which we won't go into here.

==== 4.b) Avalon-Fortress ===

Auto-generate the META-INF/services/<Classname> files using the fortress-tools ant task:

```
<goal name="fortress:meta">
  <ant:taskdef name="collect-meta"
      classname="org.apache.avalon.fortress.tools.ComponentMetaInfoCollector">
    <ant:classpath>
      <ant:path refid="maven.dependency.classpath"/>
      <ant:pathelement path="${java.build.dir}"/>
    </ant:classpath>
  </ant:taskdef>

  <collect-meta destdir="${java.build.dir}">
    <fileset dir="${java.src.dir}"/>
  </collect-meta>

  <ant:copy todir="${java.build.dir}">
    <ant:fileset dir="${java.src.dir}">
      <ant:exclude name="**/*.java"/>
      <ant:exclude name="**/package.html"/>
    </ant:fileset>
  </ant:copy>
</goal>
<postGoal name="java:compile">
  <attainGoal name="fortress:meta"/>
</postGoal>
```

Then, create a application.xconf file containing the configuration for each component:

```
<persistor id="my-persistor"/>
<engine id="my-engine">
  <persistorKey>mykey</persistorKey>
</engine>
```

Fortress doesn't look for this xconf file itself; you have to pass it in through a FortressConfig object:

```
FortressConfig config = new FortressConfig();
config.setContainerClass( DefaultContainer.class );
config.setContextDirectory( getServletContext().getRealPath("/") );
config.setWorkDirectory( (File)getServletContext().getAttribute( "javax.servlet.context.tempdir" ) );
config.setContainerConfiguration( "resource://application.xconf" );

m_containerManager = new DefaultContainerManager( config.getContext() );
ContainerUtil.initialize( m_containerManager );
m_container = m_containerManager.getContainer();
```

4.c) PicoContainer

The sister project to PicoContainer called NanoContainer includes a DomRegistrationNanoContainer that can read an xml configuration file, though you have to feed it in manually:

```
InputSourceRegistrationNanoContainer nc = new DomRegistrationNanoContainer.Default();
nc.registerComponents( new InputSource( new FileReader("pico-config.xml") );
```

Here's a pico-config.xml file:

```
<components>
  <component type="Persistor" class="PersistorImpl"/>
  <component type="Engine" class="EngineImpl">
    <param type="Persistor">null</param><!-- null parameters are automatically resolved -->
    <param type="java.lang.String">mykey</param>
  </component>
</components>
```

4.d) XWork/WebWork2

Create a components.xml file that lives at the root of the classpath:

```
<components>
  <component>
    <scope>application</scope>
      <class>PersistorImpl</class>
  </component>
  <component>
    <scope>application</scope>
    <class>EngineImpl</class>
    <enabler>PersistorAware</enabler>
    <enabler>PersistorKeyAware</enabler>
  </component>
</components>
```

XWork supports various other configuration settings (setup of interceptors, validation, etc) which we will not go into here.

4.e) HiveMind

Create a META-INF/hivemodule.xml file:

```
<?xml version="1.0"?>

<module id="interop.demo" version="1.0.0">

  <service id="my-persistor" interface="Persistor">
     <create-instance class="PersistorImpl"/>
  </service>
  <service id="my-engine" interface="Engine">
     <invoke-factory service-id="hivemind.BuilderFactory">
       <construct class="EngineImpl">
         <set-service property="persistor" service-id="my-persistor"/>
       </construct>
     </invoke-factory>
  </service>
</module>
```

HiveMind supports many more configuration settings (extension points, interceptor stacks, etc etc) which we won't go into here.

4.f) Spring

Spring, like Pico, doesn't look for an xml file; you have to pass one in. Lets call it applicationContext.xml:

```
<beans>
  <bean id="my-persistor" class="PersistorImpl"/>

  <bean id="my-engine" class="EngineImpl">
    <property name="persistor"><ref bean="my-persistor"/></property>
    <property name="persistorKey"><value>mykey</value></property>
  </bean>
</beans>
```

Exactly how you use this xml file depends on the application context (standalone, servlet, j2ee).