AvalonYouMakeTheDecision

You Make The Decision

Comparing avalon to other framework technologies

The goal of this essay is to provide a basic comparison between the avalon server framework and J2EE. Since the JBoss project provides by far the best J2EE implementation available (and the only real open source alternative to boot), we'll compare using avalon to using JBoss. We look at typical use cases and code examples to provide a first-person view of working with these two technologies.

This essay was inspired by the "You make the Decision" document by theVelocity project.

This essay is likely to be very biased, as it is written by an avalon developer. Add grains of salt where appropriate.

```
TODO: refactor contents to read more easily, get a JBoss/EJB expert to provide meaningful EJB examples.
```

Contents

Click, click, go! (Catering to the impatient)

The first step in getting started with a project is often to install it. How do you do it?

JBoss

With JBoss, you can download a 30MB zipfile containing kitchen sink and blender, which you can unzip and run, providing you with an EJB server, a Servlet Engine, a database, with no further setup neccessary:

```
wget http://unc.dl.sourceforge.net/sourceforge/jboss/jboss-3.2.1.zip
unzip jboss-3.2.1.zip
mv jboss-3.2.1 /usr/local/jboss-3.2.1
ln -s /usr/local/jboss-3.2.1 /usr/local/jboss
/usr/local/jboss/bin/run
```

Avalon

Avalon does not provide a similar "kitchen sink"-style distribution. You are usually required to think about the way you want to organise your application and how it fits in with your existing infrastructure. Avalon does provide a micro-kernel which can be run directly from the commandline. The EOB project packages up Avalon-Phoenix, an EJB-style container, the Jetty Servlet engine, and hypersonicSQL. This is the closest avalon gets to providing a J2EE-style setup:

```
# install Avalon-Phoenix
 wget http://dist.apache.easynet.nl/avalon/phoenix/phoenix-latest-bin.zip
 unzip phoenix-latest-bin.zip
 mv phoenix-4.0.4-bin /usr/local/phoenix-4.0.4
 ln -s /usr/local/phoenix-4.0.4 /usr/local/phoenix

 # get and install an EJB-style container with servlet support and
 # ["HypersonicSQL"]
 wget ttp://unc.dl.sourceforge.net/sourceforge/eob/eob-with-http-and-db.sar
 mv eob-with-http-and-db.sar /usr/local/phoenix/apps

 /usr/local/phoenix/bin/run
```

However, you can use many parts of avalon, and components built with the avalon-framework, without ever installing phoenix. In that case, your installation may look more like the following snippet of a Maven project.xml file:

```
 <dependency>
    <id>avalon-logkit</id>
    <groupId>logkit</id>
    <version>1.2</version>
</dependency>
<dependency>
    <id>avalon-framework</id>
    <groupId>framework</id>
    <version>4.1.4</version>
</dependency>
<dependency>
    <id>excalibur-logger</id>
    <version>1.0.1</version>
</dependency>
<dependency>
    <id>excalibur-pool</id>
    <version>1.2</version>
</dependency>
<dependency>
    <id>excalibur-i18n</id>
    <version>1.0</version>
</dependency>
<dependency>
    <id>excalibur-logger</id>
    <version>1.0.1</version>
</dependency>
<dependency>
    <id>excalibur-component</id>
    <version>complete-1.1</version>
</dependency>
```

coupled with a custom "project:deploy" goal which results in the building and installation of your application in whatever way makes sense.

Conclusion

JBoss is easier to install as a complete platform than what avalon has to offer. It provides a very convenient "fat" distribution.

OTOH, it is easier to custom-built your own distribution (which does not have to be so fat at all) with avalon, because avalon is componentized into smaller bits. The EOB project is one example of such a custom-built distribution, which is about as easy to install as JBoss.

I want drag and drop install! (Catering to the lazy)

Once you have your framework set up, what is next? How do you make it work for you?

JBoss

JBoss has hot-deploy capability for "enterprise application archives". In the ideal case, your deployment consists of:

```
cp my-application.ear /usr/local/jboss/deploy
```

you can also use a multitude of other options for deployment, including using the JMX management interface. I would guess a similar facility is exposed through JNDI but I'm not sure.

Avalon

The Avalon-Phoenix microkernel installed above has a similar capability:

```
cp my-application.sar /usr/local/phoenix/apps
```

you can also use a multitude of other options for deployment, including using the JMX management interface. However, depending on how you are using the various parts of avalon your process might look quite differently. For example, Apache Cocoon uses Avalon-ECM, which is embedded at buildtime rather than installed seperately. Like happens with cocoon:

```
cp cocoon.war /usr/local/tomcat/webapps
```

This provides you with a level of flexibility not normally present with J2EE-based applications (where the presence of an existing J2EE container is normally required); it also requires you to think about what makes most sense for you and figure out how to accomplish that.

Conclusion

The hot-deploy functionality available in Avalon-Phoenix is on par with that of JBoss. In addition, you have available totally different deployment models where you can reuse your components inside a very different environment. Ever tried to host and use an EJB inside a servlet application, doing away with the whole EJB container? With J2EE, that's near impossible. With avalon, it happens on a regular basis.

Combine the cool stuff (Servlets vs Avalon-framework components)

Servlets

Servlets provide a very nice way to provide web functionality to an application. A realistic example of an application encoded as a set of servlets isApache Axis. The main entrypoint into the application is the AxisServlet class. This servlet follows a setup roughly like:

```
public class MyServlet extends ["HttpPServlet"]
{
   public void init()
   {
       ServletContext context = getServletConfig().getServletContext();
       // configure servlet behaviour based on context
   }

   public void destroy()
   {
       if( engine != null )
           /* TODO: destroy the engine and remove from ctx */ ;
   }

   protected void service( HttpServletRequest req,
         HttpServletResponse resp )
         throws ServletException, IOException
   {
       ServletContext ctx = getServletConfig().getServletContext();
       MyAppEngine engine = getEngine( context );
       MyAppHandler handler = engine.getHandler( req, res, ctx );
       handler.handle( req, res, ctx );
   }

   public static MyAppEngine getEngine( ServletContext ctx )
   {
       Object o = context.getAttribute( MyAppEngine.class.getName() );
       if ( o instanceof MyAppEngine )
           return (MyAppEngine)o;
       else
       {
           Map config = null;
           /* TODO: create config based on ctx */
           MyAppEngine engine = MyAppEngineFactory.createEngine(
               config );

           context.setAttribute( MyAppEngine.class.getName(),
               engine );
       }
   }
}
```

Surely, this a setup that is difficult to beat, isn't it?

Avalon-Framework components

Well, we did anyway. Guess what? In the avalon world, you code your servlet all but exactly the same way. The thing is, with avalon, we go a few steps further. The tricky part in the above setup is that you still have to hand-code your MyAppEngine class, which, judging from AxisServer and AxisEngine and all their support classes, can be quite a lot of work in your average application.

With avalon, it is a lot easier: a truly "generic" engine is already available for you. In fact, you can choose between several of them (http://avalon.apache.org/framework/reference-containers.html) to pick the one which exactly serves your needs.

Here's an example of a MyAppEngine implemented using Avalon-Fortress (note this is untested code that probably contains typos and misses some try /catch statements and the like):

```
 public class MyAppEngine extends DefaultContainer
 {
    public MyAppHandler getHandler( HttpServletRequest req,
          HttpServletResponse resp, ServletContext ctx )
    {
        try
        {
            MyAppHandlerManager m = getServiceManager().lookup(
                    MyAppHandlerManager.ROLE );
            return m.getHandler( req, resp, ctx );
        }
        catch( ServiceException se )
        {
            return getErrorHandler();
        }
    }

    public MyAppHandler getErrorHandler()
    {
        return null; /* TODO: implement fail-safe mechanism in case
                        of a fortress configuration problem */
    }
 }
 public class DefaultMyAppHandlerManager extends AbstractLogEnabled
        implements MyAppHandlerManager, Configurable, Initializable
{
    protected Configuration m_configuration;
    protected HashMap m_handlers = new HashMap();

    /**
     * Expect configuration of the form:
     * <handler-manager>
     *  <handlers>
     *   <handler regex="/\*checkout\*/.*"
     *      className="com.my.handlers.CheckoutHandler">
     *    <!-- handler-specific configuration goes here -->
     *   </handler>
     *   <handler regex="/\*commit\*/.*"
     *      className="com.my.handlers.CommitHandler"/>
     *  </handlers>
     * </handler-manager>
     */
    public void configure( Configuration config )
            throws ConfigurationException
    {
        m_configuration = config;

        Configuration[] conf =
                m_configuration.getChild("handlers").getChildren();
        Configuration[] elements = config.getChildren();
        for ( int i = 0; i < elements.length; i++ )
        {
            Configuration element = elements[i];
            String regex = element.getAttribute( "regex" );
            String className = element.getAttribute( "class" );
            m_handlers.put( regex,
                new HandlerEntry( regex, className, element );
        }
    }

    public void initialize()
```

```
        throws InitializationException
{
    Iterator it = m_handlers.values().iterator();
    while( it.hasNext() )
    {
        HandlerEntry entry = it.next();
        try
        {
            entry.m_clazz = Class.forName( entry.m_className );
            if( !(MyAppHandler.isAssignableFrom( entry.m_clazz ) )
            {
                getLogger().error(
                    "instance of handler with regex " +
                    regex + " and className " + className +
                    "does not implement MyAppHandler!", cnfe );
                 it.remove();
            }
        }
        catch( ClassNotFoundException cnfe )
        {
            getLogger().error(
               "can't create instance of handler with regex " +
               regex + " and className " + className, cnfe );
            it.remove();
        }
        catch( InstantiationException ie )
        {
            getLogger().error(
               "can't create instance of handler with regex " +
               regex + " and className " + className, ie );
            it.remove();
        }
    }
}

public MyAppHandler getHandler( req, resp, ctx )
{
    return getHandler( req.getContextPath() );
}
public MyAppHandler getHandler( String path )
{
    Iterator it = m_handlers.values().iterator();
    while( it.hasNext() )
    {
        HandlerEntry entry = it.next();
        if( path.matches( entry.m_regex ) )
        {
            MyAppHandler h = entry.m_clazz.newInstance();

            ContainerUtil.enableLogging( h, getLogger() );
            ContainerUtil.configure( h, entry.m_configuration );
            ContainerUtil.initialize( h );

            return h;
        }
    }
    return null;
}

class HandlerEntry
{
    String m_regex;
    String m_className;
    Class m_clazz;
    Configuration m_configuration;

    HandlerEntry( String regex, String className, Configuration configuration )
    {
        m_regex = regex; m_className = className; m_clazz = null;
        m_configuration = configuration;
```

```
        }
    }
```

Conclusion

Rather than replacing the servlet API with something incompatible, meaning all the existing work that has gone into that API and the containers to support it would be void, Avalon provides some great tools, like Avalon-Fortress, which you can seamlessly embed into your servlet application, handling all of your "system-level concerns".

I'll have that neither shaken nor stirred please (EJBs vs Avalon-Framework components)

EJB Sucks

I'll be frank: I think the EJB API is one very big mess. Things like Entity Beans simply suck conceptually and implementation-wise. Most java developers have known this for a long time; try searching google for "EJB sucks" for an impression.

EJB is bloatware

Basically, the EJB API is way too big. It messes up concerns in many, many places, and from this, chaos usually ensues in EJB-based applications at some point, because everything depends on everything. The Avalon-Framework API is very compact: we need only one page in addition to the javadocs to describe it completely.

```
TODO: code example of a typical real-life EJB and a typical real-life avalon component
```

EJB requires too much work

With EJB, you have to take many, many steps before your class can qualify as an EJB, like writing Home and Remote interfaces to your EJB.

```
TODO: code example of an EJB and all its auxillary classes
```

With Avalon-AMTAGS, in order to qualify your components as a component you have to do next to nothing:

```
  /** @avalon.service type="MyService" */
  class MyClass implements MyService { /* ... */ }
```

Avalon properly seperates concerns

When you decide you need to address some system-level concern (like logging), you usually need to add only a single method:

```
 class MyClass implements MyService, LogEnabled
 {
    void enableLogging( Logger logger ) { /* ... */ }
 }

When you need to address a concern which is not system-level, Avalon does usuallly not provide it (by design!),
and you write (or reuse) a component which addresses it for you:

{{{
    class MyClass implements MyService, Servicable
  {
   MyHelperService m_helper = null;

    void service( ServiceManager sm ) throws ServiceException
    {
        helper = sm.lookup( MyHelperService.ROLE );
    }
  }
```

This is hands-down the closest you can get at the moment to cleancut AspectOrientedProgramming without requiring an extension of the java language.

```
TODO: point out some examples of typical bad mixture of concerns in EJB
```

predicatibility beats RAD on big projects (JNDI vs IoC)

Unlike EJB, JNDI does not suck. It is a directory API, and a good one. What sucks about JNDI is that it is unpredictable when you use it to store and retrieve all parts of your application. Any component can look up and use any other component. In practice, this often leads to a complex runtime dependency web between classes. There's no way to prevent circular references, there's no clean way to identify strong and weak couplings.

```
TODO: code example of JNDI use
```

The Inversion of Control design pattern, which exists in avalon at every level, avoids all of these potential issues. Rather than having a central directory containing all possible producers and consumers of components, in avalon there is a very strict, well-defined and predictable hierarchy of components. Your components do not check whether a resource exists, they need to know next to no semantics for looking up those resources. Instead, they just declare that they need something, and the avalon container figures out everything else.

```
TODO: code example of ServiceManager use
```

Conclusion

When you ask "does avalon provide feature X?", the answer is often "no". That is by design. We componentize avalon up into small parts which individually don't do a lot, and it is very easy to choose and use only exactly those things you need. We avoid the Big Ball of Mud design pattern which is common to most of the J2EE appservers, in fact to the whole J2EE idea.

This means a simple "feature comparsion" between, say, Avalon-Phoenix and JBoss will irrevocably lead to landslide victory for JBoss. Avalon-Phoenix does not come with a preconfigured database, it does not come with an embedded webserver, it does not come with a persistence mechanism. Instead, it comes with a cleaner architecture, predictable application behaviour, security and ease of maintainance. Like any part of avalon.

To use any of it we require you to think more first. We want to discourage you from desiring the kitchen sink, when you are not even sure your application requires a kitchen.

When you download avalon-framework.jar and you have written your first avalon component using it, there is nothing to "doubleclick" to make it run. You need to figure out first what kind of application you are building (command-line tool, GUI application, servlet application, server application, combination of those), and then you need to choose what kind of avalon-framework-supporting container you need to build that application. You will often need to lay a little bit of plumbing yourself in order to do things in the way which best fits those needs.

Once you have that figured out and you have configured your container of choice properly, you are still not ready to just start dropping components into it. You need to think about how to decompose the application you are building into components. Because Avalon places additional design-time requirements on your components (namely that they are passive and don't venture outside the black box you set up), this means you will need to think more at first.

Once you have your base platform in place like you want it to, you have followed all advice on system decomposition avalon provides, and you have successfully determined what components you need and what they will do, avalon gets out of the way. Implementing your components is a simple matter of defining a work interface you completely design yourself

```
   interface SomeWorkInterface
{
    public String ROLE = SomeWorkInterface.class.getName();

    Result doThis();
}
```

and providing an implementation that uses the Avalon-Framework lifecycle to get at all the stuff it needs from its environment. When you write the implementation of your work interface, what "stuff" it needs will quickly become apparent and you add lifecycle interfaces to get at that stuff.