HibernateAndTheAuthenticationFramework

Intro

The first chapter `1:Hibernate and the authentication framework` is the old version that uses a custom generator. You can find the code in `Hibernate_3.zip`

The second chapter `2:Single role user management` builds on the more up-to-date cocoon documentation. The authentication class now calls a java class directly. No need for an extra pipeline or a generator. So this solution is much more straightforward. The code is in `Contrib4.zip`. This example also explains how you can replace the flat file "single role user management" as used in e.g. the (old) portal example with a Hibernate solution.

1:Hibernate and the authentication framework

This is the third (and final?) chapter in my Hibernate trilogy.

I assume you had a quick look at part 1 and 2, and you are familiar with the Cocoon documentation about the authentication and session framework.

I wrote a generator for the Cocoon pipeline that gives you the option to use the Hibernate classes (see the flow examples) for user authentication with the Cocoon authentication framework. If you already walked through these Hibernate examples, be aware that I applied a "minor" modification to the user table: the login name is no longer the primary key. It will take some time to update the other examples.

The source is in the attachment. Change the package name to whatever you want, but make sure you use consistent names everywhere.

The goal is to create a pipeline that replaces the authentication pipeline in the example which uses a flat-file approach.

This is from the /samples :

```
 <map:pipeline internal-only="true">\\
   <!-- This is the authentication resource -->\\
       <map:match pattern="authenticate">\\
             <map:generate src="docs/userlist.xml"/>\\
          <map:transform src="stylesheets/authenticate.xsl">\\
                <map:parameter name="use-request-parameters" value="true"/>\\
        </map:transform>\\
          <map:serialize type="xml"/>\\
       </map:match>\\
 </map:pipeline>\\
```

BR

Replace it with:

```
<map:pipeline internal-only="true">\\
    <!-- This is the authentication resource -->\\
                <map:match pattern="authenticate">\\
                               <map:generate type="auth" src=""/>\\
                   <map:serialize  type="xml"/>\\
                </map:match>\\
</map:pipeline>\\
```

BR

The "auth" generator must be declared in your sitemap:

```
<map:generators default="file">\\
          <map:generator name="auth" src="nl.datagram.cocoon.generation.AuthGen"/>\\
</map:generators>\\
```

BR

And of course the compiled AuthGen class must be somewhere in your classpath (e.g. WEB-INF/classes)

Oops. Almost forgot: I am using two request variables (name and password) in my login resource (the two fields of the login page). So the part of the sitemap that performs auth service (the target of the login form) typically contains two lines:

{{
<map:parameter name="parameter_name" value="{request-param:name}"/>

<map:parameter name="parameter_password" value="{request-param:password}"/>

}}

These parameter names are hard coded in the generator (some room for improvement here).

A goodie of the generator: the user bean is stored in the session. So if your protected resource is a JXTemplate that contains something like:

{{
Welcome ${session.getAttribute("usr").firstName}
${session.getAttribute("usr").lastName}

}}
, the user will be welcomed with his own name.

2:Single role user management

Authentication

The authentication handler uses a class (HibernateAuthenticator.java) instead of an authentication pipeline. This is straightforward because it is described in the docs on the authentication framework http://cocoon.apache.org/2.1/developing/webapps/authentication.html See the java source and the sitemap. As a goodie, the user bean is also stored in the session. So, if you use the jx transformer or generator, you can use: "Welcome ${session.getAttribute ("usr").firstName} ${session.getAttribute("usr").lastName}"

Single role user management

The docs also describe some optional entries in the authentication handler for user management, e.g.:

```
<load-roles uri="cocoon:raw:/internal/roles"/>
```

The only implementation AFAIK, is the user management in the authentication framework as used by the (old) portal. To be more specific, the class:

```
org.apache.cocoon.webapps.authentication.generation.ConfigurationGenerator.java
```

The ConfigurationGenerator class of the framework mentioned above, invokes a cocoon pipeline and loads/saves the data from/to a flat xml file.

Although authentication allows for calling a java class directly instead of invoking a pipeline, the optional configuration syntax does not yet has this functionality.

So I added the following syntax:
{{{<load-roles
class="org.apache.cocoon.authdemo.ConfigurationManager"
method="loadRoles"/>
}}}

So instead of invoking a pipeline, the implementation should do a lookup of the supplied class (an Avalon component) and invoke the supplied method. This method has one parameter: the SourceParameter object with all the "type=.. , role=.. user=..." info supplied (see the authentication framework docs)

To implement this, I modified the class org.apache.cocoon.webapps.authentication.generation.ConfigurationGenerator.java
so it can differentiate between `<load-roles uri=..` and `<load-roles class=..`
It is straightforward plumbing. It has no dependencies on the instance of the implementation of the class that is invoked (the value of the class=...). So everybody can implement his/her own favorite implementation. The whole bunch of parameters is passed with the SourceParameter argument to the invoked instance.

Then I had to implement a ConfigurationManager. I defined an Avalon interface:

```
    public Document loadRoles(SourceParameters p);
    public Document loadUsers(SourceParameters p);
    public void newRole(SourceParameters p);
                etc...
```

and implemented it in a HibernateConfigurationManager. The Hibernate implementation was straightforward. And it probably should be easy to implement this as an OJBConfigurationManager because only some very basic O/R mapping features of Hibernate are used.

Things to look at to get things working:

- Copy the content of /dg/.. to the cocoon map and copy the content of WEB-INF to cocoon/WEB-INF
- Put the Hibernate jars in WEB-INF/lib
- Edit WEB-INF/classes/hibernate.properties. Enter you database login name etc.
- Have a look at WEB-INF/classes/nl/datagram/cocoon/authdemo/user.hbm.xml. This is where the Hibernate mapping is defined

- Create the user and role table (see script)
- Restart Tomcat and try /cocoon/dg/authentication-fw/ and login with admin/admin

Attachment: Hibernate_3.zip

Attachment: Contrib4.zip