LoginActionPattern

In a number of occasions we've been using this simple pattern for handling login into (part of) a Cocoon web application. Possibly it can function as a pattern for your apps as well.

An important staring point for the design of this little thing has been the observation that access control and authentication is an orthogonal aspect to the resource-addressing. In other words: it should not influence the URI-request-space design.

Practically this means that people should be able to bookmark (i.e. also copy into emails) the resource URI regardless of the referenced resource being restricted or not. There should just be a interception on the request before the sitemap goes into selecting the actual pipeline to produce the resource.

A top-level Action in combination with a simple login-form is best suited to this purpose.

The login-form is created with jxtemplate: (file screen/login.xml)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<html xmlns:jx="http://apache.org/cocoon/templates/jx/1.0">
  <head>
    <title>MYPROJECT :: Login</title>
  </head>
  <body bgcolor="white">
    <center>
      <h3>Welcome, please login:</h3>
      <form method="post">
        <table border="0">
          <tr>
            <td colspan="2">
              <font color="#ff0000">${parameters.getParameter('login.error')}</font>
            </td>
          </tr>
          <tr>
            <td>Username:</td>
            <td><input type="text" name="login.usr"
                 value="${parameters.getParameter('login.usr')}"/></td>
          </tr>
          <tr>
            <td>Password:</td>
            <td><input type="password" name="login.pwd" /></td>
          </tr>
          <tr>
            <td><input type="submit" value="Login" /></td>
          </tr>
        </table>
      </form>
    </center>
  </body>
</html>
```

The implementation pattern for such a LoginAction looks like this:

```
package my.package;

import org.apache.avalon.framework.parameters.*;
import org.apache.avalon.framework.thread.*;
import org.apache.cocoon.acting.*;
import org.apache.cocoon.environment.*;

public class MyLoginAction extends AbstractAction implements  ThreadSafe
{
  public static final Map EMPTY_MAP = Collections.unmodifiableMap(new java.util.HashMap(0));
  public static final String SESSION_ATTR = MyLoginAction.class.getName() + "/user";
  public static final String LOGIN_RESULT_ATTR   = "login.result";
  public static final String LOGIN_USER_ATTR     = "login.usr";
  public static final String LOGIN_PWD_ATTR      = "login.pwd";
  public static final String LOGIN_ERROR_ATTR    = "login.error";
  public static final String LOGIN_URI_ATTR      = "login.uri";
  public static final String LOGIN_USER_DEFAULT  ="(your username)";
  public static final String RESULT_SUCCESS_CODE = "success";
  public static final String RESULT_FAILED_CODE  = "failed";
```

```
/**
 * This action will be called upon each request towards a URI that is
 * protected.
 * <p>
 * It will check the presence of ready credentials in the session...
 * <p>
 * If those are not found in the session
 * (meaning: user has not yet succesfully logged in), then they will be set
 * if the following conditions are met: <ol>
 *   <li>This request is using the POST method</li>
 *   <li>The present request parameters login.usr, login.pwd are valid
 * </li></ol>
 * <p>
 *
 * @return either null if the login was not performed because the credentials were already
 *   present in the session or else a hashmap with indicating attribute 'login.result'
 *   that will hold the value 'success' or 'failed' indicating if or not the login
 *   was succesful. Note that 'succes' guarantees that the current session holds the
 *   valid user-credential-object under the attribute "my.package.MyLoginAction/user".
 *   This map will further contain the following attributes:
 * <table>
 * <tr><td>login.usr</td><td>username as presented in the request or a default</td></tr>
 * <tr><td>login.error</td><td>an errormessage. (in case of failure)</td></tr>
 * <tr><td>login.uri</td><td>the resource URI that was requested.</td></tr>
 *  </table>
 */

public Map act(Redirector redirector, SourceResolver resolver,
               Map objectModel, String source, Parameters parameters) throws Exception
{

  Request req = ObjectModelHelper.getRequest(objectModel);
  Session sess = req.getSession();
  MyUserAccountClass userAccount = (MyUserAccountClass) sess.getAttribute(SESSION_ATTR);
  String usr = LOGIN_USER_DEFAULT;

  Map retMap = null;
  if (userAccount != null)
  {
    return null; //skip nested action stuff since authentication is already ok
  }
  else
  {
    retMap = new HashMap();

    if (req.getMethod().equals("POST"))
    {
      usr = req.getParameter(LOGIN_USER_ATTR);
      String pwd = req.getParameter(LOGIN_PWD_ATTR);

      userAccount = ... // whatever custom login you want to code

      try
      {
        userAccount.login();
        sess.setAttribute(SESSION_ATTR, userAccount);
        retMap.put(LOGIN_RESULT_ATTR, RESULT_SUCCESS_CODE);
        retMap.put(LOGIN_URI_ATTR, req.getRequestURI());

        return retMap;
      }
      catch (LoginException le)
      {
        retMap.put(LOGIN_ERROR_ATTR, "Invalid Login - Please try again.");
        userAccount = null;
      }
    }
  }

  if (userAccount == null)
  {
```

```
      retMap = new HashMap();
      retMap.put(LOGIN_USER_ATTR, usr);
      retMap.put(LOGIN_RESULT_ATTR, RESULT_FAILED_CODE);

    }

    return retMap;
  }
}
```

This action returns one of the following:

- in case that login was not needed (credentials were available in the session)
    - `null`
- in case that login was succesful (the request was a POST with login.usr and login.pwd)
    - a hashmap with
        - `login.result` set to 'succes'
        - `login.uri` holding the original URI that was requested
    - as as side effect the user credentials are put into the session
- in case that login was not succesful or credentials were not present
    - a hashmap with
        - `login.result` set to 'failed'
        - optionally `login.error` holding an error message that can be displayed (not present in initial GET)
        - `login.usr` holding the username (or a default) so that can be re-displayed

To close the loop we need to hook up action and login-form into the sitemap. The most uncommon thing here is that the <map:action> sits directly under the <map:pipeline> (thus not under the <map:match>)

```
<map:sitemap>

  ... somewhere in /sitemap/components/actions
  ...

      <map:action name="login" src="my.package.MyLoginAction" />

  ...

  ... and upfront in the /sitemap/pipelines of all resources
  ... that require login:

  <map:pipeline>
    <map:act type="login">
      <map:select type="parameter">
        <map:parameter name="parameter-selector-test" value="{login.result}"/>

        <map:when test="success">
          <map:redirect-to uri="{login.uri}"/>
        </map:when>
        <map:otherwise>
          <map:generate src="screen/login.xml" type="jx">
            <map:parameter name="login.usr" value="{login.usr}" />
            <map:parameter name="login.error" value="{login.error}" />
          </map:generate>
          <map:serialize />
        </map:otherwise>
      </map:select>
    </map:act>

    <!-- below are resources that are controlled by the login-action -->
    <map:match...

   ...

</map:sitemap>
```