OverviewAuthenticationAndAuthorization

This document is not up-to-date wrt Lenya 2.0.

This document is supposed to be a tutorial style introduction into Authentication and Authorization in Lenya. It is meant to help understand the existing document AccessControl for those who might not yet have a detailed background of Authentication and Authorization in general and how these topics are handled by HTTP applications, servlet containers, Cocoon and Lenya in particular.

Lenya Authentication and Authorization Walkthrough

The Cocoon Sitemap in Lenya

As Lenya is based upon Cocoon you will find the same configuration files you know from Cocoon in the Lenya directory tree as well. The most important configuration file in each Cocoon application is `sitemap.xmap` as the sitemap contains the ruleset that the Cocoon engine will use to find out what to send to the browser in reply to a HTTP request received.

Let's have a look at the part of the sitemap that handles the "Login as Author" part of a publication:

```
     <map:pipeline>

       <!-- ... -->

       <map:match type="usecase" pattern="login">

         <!-- This will just show the login form (== screen in Lenya terminology) -->
         <map:match type="step" pattern="showscreen">
           <map:generate type="serverpages" src="lenya/content/ac/login.xsp"/>
           <map:transform src="lenya/xslt/ac/login.xsl">
             <map:parameter name="publication_name" value="{page-envelope:publication-id}"/>
           </map:transform>
          <map:call resource="style-cms-page"/>
         </map:match>

         <!-- This will pass the username / password for authentication -->
         <map:match type="step" pattern="login">
           <map:act type="authenticator">
             <map:redirect-to uri="{request:requestURI}" session="true"/>
           </map:act>
           <map:redirect-to uri="{request:requestURI}?lenya.usecase=login&amp;lenya.step=showscreen&amp;
status=failed" session="true"/>
         </map:match>

       </map:match>

       <!-- ... -->

     </map:pipeline>
```

Note the <map:act type="authenticator"> ... </map:act>. The action is defined in the sitemap in the <map:actions> ... </map:actions> part:

```
   <map:actions>
    <!-- ... -->
    <map:action name="authenticator" src="org.apache.lenya.cms.cocoon.acting.DelegatingAuthenticatorAction"
logger="lenya.sitemap.action.authenticator"/>
    <!-- ... -->
```

As you can easily tell from the Java package name org.apache.lenya.cms.cocoon.acting this is where Cocoon hands over to Lenya code for the first time.

Authentication

Authentication is process of confirming a user is who it claims to be. This is usually acheved in logging-in process (supplying username and password and checking it against database) process. This is handled in special pipeline by `AuthenticatorAction`.

The DelegatingAuthenticatorAction supplied by Lenya

The class org.apache.lenya.cms.cocoon.acting.DelegatingAuthenticatorAction implements a Cocoon action which will handle the entire Lenya user authentication (read: login) process.

The code is really straightforward:

```
Map result = null;
if (getAccessController().authenticate(request)) {
    getLogger().debug("Authentication successful.");
    result = Collections.EMPTY_MAP;
}
else {
    getLogger().debug("Authentication failed.");
}
return result;
```

As the Delegating... in the classname suggests all this piece of code does is to find out what [AccessController] is configured for this case and delegate the job there, just reporting back the result. This is where the Cocoon / Avalon concept of **Inversion of Control** comes to play. The Cocoon configuration will be used internally to decide which class should actually perform the action thus determining against what the user should be authenticated (file [=default] | LDAP | others such as JAAS [yet to be implemented]).

Declaring the Authenticator in Lenya's cocoon.xconf configuration file

In WEB-INF/cocoon.xconf there is a line declaring that if someone is looking for a role called *org.apache.lenya.ac.Authenticator* this will be implemented by the class *org.apache.lenya.ac.impl.UserAuthenticator*:

```
<component role="org.apache.lenya.ac.Authenticator" class="org.apache.lenya.ac.impl.UserAuthenticator" logger="
lenya.ac.authenticator"/>
```

Internal Note: In WEB-INF/classes/org/apache/lenya you fill find a file called lenya.roles which contains roles to classes mapping as well. It seems though that this is overwritten by components that are declared in cocoon.xconf. So it might be best to erase the overlapping entries in lenya.roles. But don't erase the whole file as this will lead to the Avalon framework throwing an exception about the file not being there.

If you are looking to use anything else except files as your user database you might be temted now to replace the UserAuthenticator that comes with Lenya with a different one that will for example query an LDAP server. This could be done, but it would probably no give you the full mileage. You will also note that nothing in *org.apache.lenya.ac.impl.UserAuthenticator* (neither the package name nor the classname) names any mechanism. So read on, we haven't reached the right hook yet! (UserAuthenticator is still pretty agnostic to user databases!)

The UserAuthenticator is handed an AccreditableManager and a request and needs to answer: Yes - the user is allowed to make that request or No - the user is not allowed to make this request. This is achieved by calling the only public method of UserAuthenticator.

```
public boolean authenticate(AccreditableManager accreditableManager, Request request)
```

This is common method for all authenticators. Currently only the UserAuthenticator is implemented, and it delegates authentication to its specific method:

```
protected boolean authenticate(AccreditableManager accreditableManager, String username, String password,
Identity identity)
```

This method then retrieves User object from accreditableManager and calls to User.authenticate(password). If it passes, User object then stored in Identity - an object stored in session. Identity stores all things that could be used to identificate (and authorize) request, these are: remote host IP, user name, and World. They are called Accreditables and used by AccessController to authorize request.

Authorization

Authorization is process of determining whether current request is allowed. Authorization takes in account what url is requested, what usecase performed, etc. This process is performed by `AuthorizerAction`.

The DelegatingAuthorizerAction

This is another action used to check if requested resource, usecase or action is allowed during current request. `DelegatingAuthorizerAction` uses avalone stuff to get AccessControllerResolverSelector, to select the only implemented AccessControllerResolver, which can at last return `AccessController`, responsible to grant access for requested url (IIRC).

There are 1.5 AccessControllers described at http://lenya.apache.org/1_2_x/components/accesscontrol/accesscontrollers.html **FIXME: what is this mysterious half of an AccessController?**

The `DefaultAccessController` tries to authorize request with each of configured authorizers, calling `Authorizer.authorize(request)`.
Authorizers are described at http://lenya.apache.org/1_2_x/components/accesscontrol/authorizers.html and configured in ac.xconf

Accreditables, Identity and Authorizers

Accreditables are objects that can be used to gain access: `Users`, `Groups`, network (`IPRanges`). These are used by `AccessController` to assign roles to requests.
`Machines` are another type of accreditables, but they need not to be managed, and get accredited through `IPRanges`.

Every session contains an `Identity` object, which contains all accreditables associated with the request. These are: World (everybody owns this accreditable), remote host IP, and logged user.

When the `AccessController` authorizes a request, each selected authorizer takes `Accreditables` from the session, and for every item determines roles and assigns them to processing request. Each `Authorizer` has its own implementation of the `.authorize(request` method (e.g. using config files, policies or more hairy things).

Actually, authorizers do no check for items themselves, but for associated `Accreditables`, which are acquired via calls to `accreditable.getAccreditables()` for each item. For users returned accreditables are user itself and the groups the user belongs to. For hosts they are the networks the host belongs to.

Authorizers

There are several authorizers, some of them assigns roles to request and some of them checks assigned roles to allow or deny access.

The `PolicyAuthorizer` assigns roles to request using policy-manager and config directory. It checks if some roles assigned, then it grants access to requested url. **NOTE: it should be declared first in `ac.xconf`, otherwise other authorizers could not find roles to check.**

The `UsecaseAuthorizer` checks if request is allowed to access some usecase. What roles are allowed to perform usecases is configured in file, referenced in the authorizer configuration as parameter "configuration".

The `WorkFlowAuthorizer` checks if request is allowed to make workflow transition (an event). Allowed roles for each transition is assigned in workflow definition (/config/workflow/*.xml)

AccreditableManager

`AccreditableManager` is responsible to manage `Accreditables` 🙂.

`Roles` themselves are also handled by `AccreditableManager`, althoug `Roles` are not accreditables, and generaly sayng, need not to be managed.

The only function of `AccreditableManager` is to provide instances of `(User|Group|IPRange|Role)Manager}}s, associated with common configuration. There could be several {{AccreditableManagers` in system (egg. associated with different config directories), but their behaviour seems to be unpredictable '**FIXME**'.

These `*Manager}}s provides access to actual users/groups/etc database and is used to tretrieve/store/delete/create {{Accreditables`.

AC framework Configuration

All components 'deployment' is both in cocoon.xconf and lenya.roles. This is global components configuration.

The `PublicationAccessControllerResolver` is configured in file `$path_to_pub/config/ac/ac.xconf`
(location of file is hardcoded). So, it provides publication-specific configuration.

The `ConfigurableAccessControllerResolver` is configured in cocoon.xconf with structures similar to ac.xconf inside `<component-instance>` element.

All this stuff is described at http://lenya.apache.org/1_2_x/components/accesscontrol/
For all configuration directives assume them inside ac.xconf or `<component-instance>` for `ConfigurableAccessControllerResolver`.