Authentication Framework (2.1 legacy document)

Table of contents

Table of contents

Warning:
This document was copied as is from the Cocoon 2.1 documentation, but has not yet been fully reviewed or moved to its new home.

1. Introduction

One central point in building a web application is authentication and authorization. The Cocoon authentication framework is a flexible module for authentication, authorization and user management. A user can be legitimated using any information available via any source, e.g. an existing database, LDAP or the file system. With this mechanism it is very easy to use an exisiting user management/authentication system within Cocoon.

The basic concept of the authentication framework is to protect documents generated by Cocoon. By document we refer to the result of a request to Cocoon, this can either be the result of a pipeline or of a reader defined in the sitemap.

A document is protected by a so called (authentication) handler. A document is associated to a defined handler to be protected. A user can only request this document if he is authenticated against this handler.

A handler can be used to protect several documents in the same way. If a user is authenticated he can access all these documents. It is possible to use different handlers, to product documents in different ways.

The use of the authentication framework and its components is described in the following chapters.

As you will see, the user management of the authentication framework is very flexible. You can design your application without taking into account, which backend is used for the user management. This can be the file-system, a SQL database, an XML database, a LDAP directory, just anything. Simply by developing the *authentication resource*, you can connect to any system. And another advantage is the flexible switching between user databases. You can for example use the file-system for the development process and switch than later on to a LDAP system on the production system. This can be done by simply changing the *authentication resource*. If you test this resource on your production system, you don't have to test your whole application again. (Although in general this might be a good idea...).

2. Sitemap-Components

The authentication Framework adds some actions to the sitemap: the *auth-protect* action, the *auth-login* action, the *auth-logout* action and the *auth-loggedIn* action. The *authentication-manager* gets the configuration for the authentication framework and the actions controle the pipelines. The *auth-login* and the *auth-logout* action control the authentication whereas the *auth-loggedIn* action controls the application flow.

Overview

3. Protecting Documents

One feature of the framework is the user authentication. A document can be accessible for everyone or it can be protected using this framework. The process of requesting a document can be described as follows:

1. The user request a document (original document).
2. The authentication framework checks if this document is protected. If no protection is specified, the response to the request is this original document.

3. If the document is protected, the framework checks, if the user is authenticated to view it.
4. If the user is authenticated, the response is the original document. If not the framework redirects to a special redirect-to document. This redirect-to document is freely configurable and can for example contain information about the unauthorized access and in addition a login form.
5. Using the login form an authentication resource can be called with the corresponding user information (e.g. user id and password). This authentication resource uses the framework for the authentication process.
6. In case of a successful authentication the framework can redirect to the original document (or to any configured start document).
7. If the authentication fails another document is invoked by the framework displaying information to the user.

This process is only one example for a use-case of the framework. It can be configured for any authentication scheme. All resources are freely configurable.

3.1. The Authentication handler

The basic object for authentication is the so called (authentication) handler. It controlles the access to the documents. Each document in the sitemap can be related to exactly one authentication handler. All documents belonging to the same handler are protected in the same way. If a user has access to the handler, the user has the same access rights for all documents of this handler.

Each authentication handler needs the following mandatory configuration:

- A unique name.
- The authentication resource: A Cocoon pipeline trying to authenticate a user. (We will see later on, that there are more possibilities than using a pipeline).
- The redirect-to document: This document is displayed when a not authorized user tries to access a protected document.

3.2. The Configuration of a Handler

So let's have a look at the configuration. A handler can be configured in the sitemap. It is a so-called component configuration for the authentication manager. This configuration takes place in the *map:pipelines* section of a sitemap:

<map:sitemap> ...component definitions... <map:pipelines> <map:component-configurations> <authentication-manager> <handlers> <handler name="portalhandler"> <redirect-to uri="cocoon:/sunspotdemoportal"/> <authentication uri="cocoon:raw:/sunrise-authuser"/> </handler> </handlers> </authentication-manager> </map:component-configurations> <map:pipeline> ... document pipelines following here:

Using a unique name for each handler (only alphabetical characters and digits are allowed for the handler name), the framework manages different handlers. So various parts of the sitemap can be protected in different ways.

A handler is inherited to a sub sitemap. Each sub sitemap can define its own handlers. These handlers are only available to the sub sitemap (and of course to its sub sitemaps). However, it is not possible to redefine (overwrite) a previously defined handler in a sub sitemap.

3.3. Protecting Documents

A document can be protected by associating it to a defined handler. This is done by using the *auth-protect* action and the handler parameter:

```
<map:match pattern="protectedresource"> <map:act type="auth-protect"> <map:parameter
name="handler" value="portalhandler"/> <map:generate src="source/resource.xml"/>
<map:serialize type="xml"/> </map:act> </map:match>
```

If this document is requested, the action checks if the user is authenticated against the defined handler. If not, the action automatically redirects to the *redirect-to* document configured in the handler. (In the example above this is the pipeline defined by *cocoon:/sunspotdemoportal*.

If the user is authenticated, the commands inside the *map:act* will be execute and the user gets the document itself.

So, the *auth-protect* action must be included in the pipeline of the document. It gets the handler information as a parameter. If the pipeline does not use the *auth-protect* action or the parameter *handler* is missing, the document is accessible by any user.

You will see learn later on how to efficiently protect several documents with a handler.

3.4. The redirect-to document

If the requested document is not accessible to the user, the authentication framework redirects to the configured *redirect-to* document. This document is a mandatory configuration of the authentication handler as we have seen above.

This *redirect-to* document is an unprotected pipeline in the sitemap. For tracking which document was originally requested by the user, the *redirect-to* pipeline gets the request parameter *resource* with that value. In addition all parameters specified inside the *redirect-to* tag of the handler configuration are passed to the pipeline as well.

For example, the *redirect-to* document can contain a form for the user authentication. This form should invoke the real authentication process that is described below. However, the document you show when an unauthorized access is made, can be controlled by you by defining this *redirect-to* document.

4. Authenticating a User

Usually, the *redirect-to* document of a handler contains a form for the user to authenticate. But of course, you are not limited to this. No matter how the *redirect-to* document looks like, the user has "somewhere" the abilitiy to authenticate, so in most cases the user has a form where he can enter his information (e.g. user name and password). You have to write a pipeline presenting this form to the user. When the form is submitted, the authentication process has to be started inside the authentication framework. As a submit of a form invokes a request to Cocoon, a pipeline in the sitemap is triggered. We refer to this pipeline with *login pipeline*.

4.1. The Login Process

The authentication process is started by invoking the *auth-login* action. So, the *login pipeline* has to contain this action:

```
<map:match pattern="login"> <map:act type="auth-login"> <map:parameter name="handler"
value="portalhandler"/> <map:parameter name="parameter_userid"
value="{request-param:name}"/> <map:parameter name="parameter_password"
value="{request-param:password}"/> <map:redirect-to uri="authentication-successful"/>
</map:act> <!-- authentication failed: --> <map:generate src="auth_failed.xml"/>
<map:transform src="tohtml.xsl"/> <map:serialize/> </map:match>
```

The *auth-login* action uses the handler parameter to call the *authentication resource* of this handler.

This *authentication resource* needs to know the information provided by the user, e.g. in the form. For each piece of information an own parameter is created. The name of this parameter has to start with "parameter_". So in the example above, the *authentication resource* gets two parameters: userid and password. As the values of these parameters were sent by a form they need to be passed on to the *authentication resource*. If you use "{request-param:...}" for the value of a parameter, the *auth-login* action will pass the actual value of that request parameter to the *authentication resource* (by using the input modules concept of Cocoon).

You might be wondering why we explicitly pass the request parameters on to the internal pipeline call. Note that the *authentication resource* of the portalhandler is defined by *cocoon:raw*. By using this, no request parameter of the original request is passed on to the internal pipeline by default and therefore we have to define them explicitly. If you use *cocoon:* then the parameters of the form are by default passed on to the *authentication resource* and we could omit the parameter definition from above. But we feel that it is safer to explicitly define them.

If the user is not already authenticated with this handler, the framework calls the *authentication resource* and passes it the parameters. If this authentication is successful, the action returns a map and the sitemap commands inside the *map:act* are executed. A session is created on the server (if not already done) as well.

If the authentication fails, the action does not deliver a map and therefore the commands inside the *map:act* are skipped. The error information delivered by the *authentication resource* is stored into the *temporary* context. So you can get the information using either the *session transformer* or the *session-context input module*.

As you can see from the example above, you are not limited in defining the information the user has to provide. This can be either one field, two or as many fields as you need.

4.2. The authentication resource

The last chapter described the authentication process but left out details about the authentication itself. This chapter closes this gap.

The authentication can be done by different components:

- A sitemap resource (pipeline).
- A distant resource, e.g. requested via HTTP.
- A java class.

The first two are actually similar as in both cases a URI is called. So we will talk about them in the next chapter. Authentication using a java class is the topic of the following chapter.

4.2.1. Using a URI as the authentication resource

Using this flexible approach nearly any kind of authentication is possible (e.g. database, LDAP). The *authentication resource* is another mandatory configuration of the authentication handler:

<autentication-manager> <handlers> <!-- Now follows the handlers configuration --> <handler name="portalhandler"> <!-- The login resource --> <redirect-to uri="cocoon:/sunspotdemoportal"/> <authentication uri="cocoon:raw:/sunrise-authuser"/> </handler> </handlers> </autentication-manager>

If the *authentication resource* is a sitemap resource or a remote resource, this resource is requested by the framework with the given parameters from the *auth-login* action (see previous chapter). In addition all parameters inside the *authentication* tag of the handler configuration are passed to the resource. The

response of this resource must contain valid XML conforming to the following scheme:

<authentication> <ID>Unique ID of the user in the system</ID> <role>rolename</role> <!-- optional --> <data> Any additional optional information can be supplied here. This will be stored in the session for later retrieval </data> </authentication>

The XML is very simply, only the root element *authentication* and the *ID* element with a valid unique ID for the user in this handler is required. Everything else is optional.

The framework checks the response of the authentication resource for the given scheme: the root node must be named *authentication* and one child called *ID* must be present. In this case the authentication is successfull and the framework creates an authentication session context and stores the XML inside.

The mandatory information inside this XML scheme, the *ID* tag, is an unique identification for the given user inside the web application or more precisly inside this handler. The *role* is optional and can for example be used for categorizing users and displaying different functionality inside the Cocoon portal engine).

As stated, the *role* element is optional, you can use your own categorization and exchange it with a *roles* element or a *group* element or leave it out, if you don't need it. In addition you can add any other element there as well and access the information later on.

Using the *data* node the *authentication resource* can pass any information of the user into the session. From there you can retrieve the information as long as the session is valid.

If the authentication is not successful, the resource must create an XML with the root node *authentication*, but of course without the *ID* tag. In addition a *data* node can be added containing more information about the unsuccessful attempt. This data node is then stored into the *temporay* context (see previous chapter).

It is advisable to make an internal pipeline for the *authentication resource*. An internal pipeline is not directly accessible by a user.

4.2.2. Using a Java class as the authentication resource

Using a class is an alternative for using a pipeline. You can define this class in the handler configuration as an attribute *authenticator* of the *authentication* element, e.g.:

<autentication-manager> <handlers> <!-- Now follows the handlers configuration --> <handler name="portalhandler"> <!-- The login resource --> <redirect-to uri="cocoon:/sunspotdemoportal"/> <authentication authenticator="mypkg.MyAuthenticator"/> </handler> </handlers> </autentication-manager>

This class must conform to the *Authenticator* interface. This interface provides a method that tries to authenticate a User and delivers XML that is stored in the session on success. So, the behaviour is similar to the pipeline.

4.3. Logging out

The logout process is triggered by the "auth-logout" action:

<map:act type="auth-logout"> <map:parameter name="handler" value="unique"/> </map:act>

This action logs the user out of the given handler and removes all information about this handler stored in the session.

5. User Management

In addition to the authentication the framework manages all kinds of information belonging to the user in XML format. For this reason the framework creates an own session context called *authentication*. All information is stored in this context.

The authentication information (the "authentication" scheme retrieved from the authentication resource) is stored in this context, so you can retrieve and change the information using the session transformer and the usual getxml, setxml etc. commands, so we suggest you to read the session context document.

The *authentication* context is only available to the *session transformer* if the pipeline, the transformer is running in, is associated to the (authentication) handler. Or putting it in other words: you have to use the *auth-project* action in that pipeline. Otherwise the *authentication* context is not available.

5.1. Getting information from the context

Each information from within the context is gettable using an XML tag:

```
<session:getxml context="authentication" path="/authentication/ID"/> <!-- Get the ID -->
<session:getxml context="authentication" path="/authentication/data/username"/>
```

The path expression is an absolute XPath-like expression where only concrete nodes and attributes are allowed. The session transformer replaced the tag with the value of the first node found in the context, this can either be text or XML.

5.2. Setting information in the context

Using another tag information can be stored into the context:

```
<session:setxml context="authentication" path="/authentication/data/usersername"> Mr. Sunshine </session:setxml>
```

Again the path is an absolute XPath-like expression where only concrete nodes and attributes are allowed. If the requested node exists, the framework changes the value of that node. If the node does not exists, the framework adds it to the context with the given value.

The tag is removed from the resource.

6. Application Management

A very useful feature for building and maintaining web applications is the application management. It allows to configure different applications and to manage the user data for these applications.

6.1. Configuring an Application

A "authentication" application is related to one authentication handler, so an application is part of the authentication handler configuration:

```
<autentication-manager> <handlers> <handler name="unique"> ....redirect-to/authentication configuration <applications> <!-- the applications for this handler --> <application name="unique"> <load uri="loadapp"/> <!-- optional --> <save uri="saveapp"/> <!-- optional --> </application> </applications> </handler> </handlers> </autentication-manager>
```

A configuration for an application consists of a unique name (only alphabetical characters and digits are allowed for the application name) and optional load and save resources. The application configuration can contain application specific configuration values for the various parts of the

application, e.g. information for a portal.

On a successful authentication the framework invokes for each application of the handler the load resource (if present). The content or result of the load resource is stored into the session context.

The user does not always visit all sides or all applications at once. So it is not necessary to load all applications in advance when not all information is needed. Each application can specify if the data is loaded on successful authentication or the first time needed:

....<application name="unique" loadondemand="true"/>...

The load resource gets several parameters: all values of the subnodes of the "authentication" node from the authentication context (e.g. ID, role etc.) and the parameter "application" with the unique name of the application. This unique name must not contain one of the characters '_', ':' or '/'.

In addition the load and save resource get all parameters specified inside the load / save tag of the handler configuration.

6.2. Configuring the resources

For managing the application the framework needs to know to which application a resource belongs. So in addition to the handler parameter the auth-protect action gets the application name as a second parameter:

<map:match pattern="protectedresource"> <map:action type="auth-protect"> <map:parameter name="handler" value="unique handler name"/> <map:parameter name="application" value="unique application name"/> <map:generate src="source/resource.xml"/> ... </map:action> </map:match>

With this mechanism each application resource can easily access its and only its information. If a resource has no "application" parameter it can not access information of any application.

6.3. Getting, setting and saving application information

Analogue to the access of the authentication data a resource can access its application data:

<session:getxml context="authentication" path="/application/username"/> <session:setxml context="authentication" path="/application/shoppingcart"><item1/><item2/></session:setxml>

The path underlies the same restrictions and rules as always, but it has to start with "/application/".

7. Module Management

In addition to the application management the framework offers a facility called module management. It enhances the application management by the possibility to configure components for the application. For example the Cocoon portal engine needs information about where the portal profile for the user is retrieved from, where the layout is stored etc. Now each portal needs this information. Assuming that a portal is an application each application needs this information. As only the portal engine itself knows what information it needs, the module management is a standarized way for configuring such components.

The module configuration is part of the application configuration:

<autentication-manager> <handlers> <handler name="unique"> ....redirect-to/authentication configuration <applications> <!-- the applications for this handler --> <application name="unique"> ... <configuration name="portal"> ...portal configuration </configuration>

</application> </applications> </handler> </handlers> </autentication-manager>

So whenever the portal engine is asked to build the portal it can easily retrieve its configuration from the current application by getting the module configuration named "portal".

8. User Administration

Using the framework it is possible to add new roles to the system and to add new users. For this purpose, there are several optional entries for the authentication handler which provide the needed functionality:

<autentication-manager> <handlers> <handler name="unique"> ...redirect-to/authentication configuration... <!-- Optional resource for loading user information --> <load-users uri="cocoon:raw://financeresource-sunrise-loaduser"/> <!-- Optional resource for loading roles information--> <load-roles uri="cocoon:raw://financeresource-sunrise-roles"/> <!-- Optional resource for creating a new user --> <new-user uri="cocoon:raw://financeresource-sunrise-newuser"/> <!-- Optional resource for creating a new role --> <new-role uri="cocoon:raw://financeresource-sunrise-newrole"/> <!-- Optional resource for changing user information --> <change-user uri="cocoon:raw://financeresource-sunrise-newuser"/> <!-- Optional resource for deleting a role --> <delete-role uri="cocoon:raw://financeresource-sunrise-delrole"/> <!-- Optional resource for deleting a user--> <delete-user uri="cocoon:raw://financeresource-sunrise-deluser"/> </handler> </handlers> </autentication-manager>

The entries are described in the following subchapters. All tags can have additional parameter definitions which are passed to the given resource, e.g:

<!-- Optional resource for deleting a user--> <delete-user uri="cocoon:raw://financeresource-sunrise-deluser"> <connection>database</connection> <url>db:usertable</url> </delete-user>

8.1. Getting Roles

The *load-roles* resource is invoked from the framework whenever it needs information about the available roles. This resource gets the parameter "type" with the value "roles" and should deliver an XML schema with the root node "roles" and for each role a subelement "role" with a text child of the rolename:

<roles> <role>admin</role> <role>guest</role> <role>user</role> </roles>

8.2. Getting Users

The *load-users* resource is called whenever information about the available users is needed. There are three different uses of this resource:

- Loading all users: The resource gets the parameter "type" with the value "users". It should then deliver all users in the system.
- Loading all users of one role. The resource gets the parameters "type" with the value "users" and "role" with the rolename.
- Load information of one user. The resource gets the parameters "type" with the value "user", "role" with the rolename and "ID" with the authentication ID of the user.

The XML format of the resource should look like the following:

<users> <user> <ID>authentication ID</ID> <role>rolename</role> <data> ... application

specific data ... </data> </user> <user> ... </user> ... </users>

8.3. Creating a new role

The *new-role* resource creates a new role in the system. It gets the parameters "type" with the value "role" and "role" with the new rolename.

8.4. Creating a new user

The *new-user* resource creates a new user with a role. It gets the parameters *"type"* with the value *"user", "role"* with the rolename and *"ID"* with the new ID for this user.

8.5. Changing information of a user

The *change-user* resources changes information of a user. It gets the parameters "type" with the value "user", "role" with the rolename and "ID" with the ID of the user. In addition all - application specific - information of this user is send as parameters.

8.6. Delete a user

The *delete-user* resource should delete a user. It gets the parameters "type" with the value "user", "role" with the rolename and "ID" with the ID of the user.

8.7. Delete a role

The *delete-role* resources deletes a role. It gets the parameters "type" with the value "role" and "role" with the rolename .

9. Configuration Summary

Here is a brief summary of the authentication handler configuration:

<autentication-manager> <handlers> <handler name="unique"> <!-- The redirect-to resource --> <redirect-to uri="cocoon:raw://loginpage"/> <!-- Authentication resource --> <authentication uri="cocoon:raw://authenticationresource"/> <load uri="cocoon:raw://authenticationsaveresource"> <!-- optional parameters --> </load> <!-- optional save resource --> <save uri="cocoon:raw://authenticationsaveresource"> <!-- optional parameters --> </save> <applications> <!-- the applications for this handler --> <application name="unique"> <!-- Loading/Saving --> <load uri="cocoon:raw://loadapp"> <!-- optional --> <!-- optional parameters --> </load> <save uri="cocoon:raw://saveapp"> <!-- optional --> <!-- optional parameters --> </save> <!-- module configurations: --> <configuration name="portal"> ...portal configuration </configuration> </application> </applications> </handler> </handlers> </autentication-manager>

10. Pipeline Patterns

As explained in the previous chapters, the framework uses the *auth-protect* action for authentication and protecting documents. This chapter shows some common used pipeline patterns for using this framework.

10.1. Single protected document

For protecting a document with an authentication handler only the *auth-protect* action with the

parameter configuration for the handler is required.

Pattern:

1. Pipeline matching
2. Using the *auth-protect* action for protecting

Example:

```
<map:match pattern="protected"> <map:act type="auth-protect"> <!-- protect the resource --> <map:parameter name="handler" value="myhandler"/> <map:generate src="resource.xml"/> <map:transform src="toHTML"/> <map:serialize/> </map:act> </map:match>
```

It is very important that the *auth-protect* action wrapps the real pipeline, as the pipeline is only invoked if the action grants access. The matching must be done before the action is checked as the action performs a redirect for this document.

10.2. Multiple protected documents

Often you want to protect a bunch of documents in the same way. One solution is to use the single protected document pattern for each document. With the multiple protected document pattern you only have to use the action once for all documents and not within each document pipeline.

The prerequisite for this is a common matching pattern for the documents:

1. Pipeline pattern matching
2. Using the *auth-protect* action for protection
3. Pipeline matching

Example:

```
<map:match pattern="protected-*"> <map:act type="auth-protect"> <!-- protect the resource --> <map:parameter name="handler" value="myhandler"/> <map:match pattern="protected-first"> <map:generate src="resource1.xml"/> <map:transform src="toHTML"/> <map:serialize/> </map:match> .... <map:match pattern="protected-second"> <map:generate src="resource2.xml"/> <map:transform src="toHTML"/> <map:serialize/> </map:match> </map:act> </map:match>
```

Very important - as explained with the single document pattern - is the leading match before the action is performed. The second match is required to check which pipeline to use.

10.3. Controlling the Application Flow

If you want to create documents which behave different wheather you are logged in or not, the *auth-loggedIn* action is the component to controll your application flow. This action checks if the user is authenticated for a given handler and calls all sitemap components inside the *act* tag.

```
<map:match pattern="startpage"> <map:act type="auth-loggedIn"> <!-- check authentication --> <map:parameter name="handler" value="myhandler"/> <map:redirect-to uri="loggedInStartPage"/> </map:act> <map:generate src="startpage.xml"/> <map:transform src="toHTML"/> <map:serialize/> </map:match>
```

In the example above, if the user is already logged he is redirected to the *loggedInStartPage* document. If he is not logged in for the given handler, the usual start page is generated.

The *auth-protect* action returns - if the user is logged in for the given handler - all values from the context to the sitemap, e.g. ID, role etc. These values can be used within the other components:

```
<map:match pattern"protected"> <map:act type="auth-protect"> <!-- protect the resource -->
```

<map:parameter name="handler" value="myhandler"/> <!-- Append the ID of the user to the file name --> <map:generate src="resource_{ID}.xml"/> <map:transform src="toHTML"/> <map:serialize/> </map:act> </map:match>

But the *auth-loggedIn* action does not give the included pipeline access to the authentication context belonging to the handler. If you want this, you have to nest the *auth-protect* action inside!

<map:match pattern"start"> <map:act type="auth-loggedIn"> <!-- check authentication --> <map:parameter name="handler" value="myhandler"/> <map:act type="auth-protect"> <!-- give access to the context --> <map:parameter name="handler" value="myhandler"/> <map:generate src="getinfofromcontext.xml"/> <map:transform type="session"/> <map:transform src="toHTML"/> <map:serialize/> </map:act> </map:act> </map:match>

10.4. Session Handling

If a user is authenticated the user has a session. However, care has to be taken that the session tracking works, which means that Cocoon can detect that a follow up request of the user belongs to the same session.

The easiest way is to use the *encodeURL* transformer as the last transformation step in your pipeline. For more information about session handling, have a look in the chapter about sessions.

1. Comments

add your comments