Products.PluggableAuthService Documentation

Release 2.2.dev0

Oct 23, 2019

Contents

CHAPTER 1

Purpose

The PluggableAuthService is designed to allow incorporation of *any* existing user folder (or related technology), and to make it simple to define project-specific extensions.

CHAPTER 2

Theory of Operation

The PluggableAuthService defines a framework for a set of plugins which it orchestrates to generate user objects from requests. These user objects implement the “traditional” BasicUser API, and provide additional functionality.

CHAPTER 3

Narrative documentation

Narrative documentation explaining how to use `Products.PluggableAuthService`.

3.1 Plugins

3.1.1 Plugin Types

PluggableAuthService defines the following plugin types:

- Authentication plugins identify the user based on data in the request.
  - Each PluggableAuthService must contain at least one authentication plugin.
  - The PluggableAuthService defines an ordered set of authentication plugins, and queries them in order for each request. The first plugin to recognize a user returns the user, or raises an exception (e.g., for password mismatches). If no plugin returns a user, the PluggableAuthService returns an "anonymous" user (which may still have "extended" information added later).
- Challenge plugins alter the response to force the user to (re)authenticate, e.g. by redirecting it to a login form, or by setting the protocol-specific headers which initiate the desired challenge.
- Decorator plugins add propertysheets to a user, based on request data or on other data sources.
  - These sources might include application data from the ZODB or from SQL, etc.
  - They might also pull in user data from LDAP, ActiveDirectory, passwd files, etc.
- Group plugins add groups to the list of groups to which the user belongs, using request data or previously-added decorations.
- Update plugins write updates back to the data store from which they came (ZODB, SQL, LDAP, etc.)
- Validation plugins impose business-specified policies on user properties (particularly on login ID and password).

Note: When using more than one plugin for authentication, only one challenge can be sent to the user - the one from the plugin at the top position of active **Challenge Plugins** configuration screen in the ZMI.

Nevertheless, you can instantiate the **Challenge Protocol Chooser Plugin**. Then you can assign, for instance, **Cookie Auth** for requests from the browser, and **HTTP Basic Auth** for requests via XML-RPC.

3.1.2 Plugin Registration

PluggableAuthService plugins are configured via the ZMI, or alternatively via an XML import / export mechanism. Each plugin is identified using a TALES path expression, which will be evaluated with an implied 'nocall' modifier; plugins are intended to be callables, with known argument signatures.

3.2 Request flow

1. The publisher asks the PluggableAuthService to validate the user's access to a given object:

```
groups.validate( request, auth, roles )
```

2. PluggableAuthService polls its authentication plugins in order, asking each in turn for a user:

```
for id, plugin in self.listAuthenticationPlugins():

    try:
        user = plugin( request, auth )

    except Unauthorized:
        self.dispatchChallenge( request )

    else:
        user.setAuthenticationSource( id )
        break

else:
    user = self.createAnonymousUser()
```

3. PluggableAuthService allows each of its decorator plugins to annotate the user:

```
for id, plugin in self.listDecoratorPlugins():

    known, schema, data = plugin( user )

    if known:
        sheet = UserPropertySheet( id, schema, **data )
        user.addPropertySheet( id, sheet )
```

4. PluggableAuthService allows each of its group plugins to assert groups for the user:

```
for id, plugin in self.listGroupPlugins():

    groups = plugin( user )
    user.addGroups( groups )
```

5. PluggableAuthService returns the annotated / group-ified user to the publisher.

3.3 Caching

The PluggableAuthService contains a caching mechanism based on the built-in Zope Caching framework with a mix-in class that defines caching methods for cacheable content and so-called Cache Managers that store cacheable data.

3.3.1 How does the site admin use it?

Given a plugin that is cache-enabled the steps to using the cache are easy. The site administrator needs to...

- Instantiate a "RAM Cache Manager" inside the PluggableAuthService
- Using the "Cache" tab in the cache-enabled plugin's ZMI view, associate the plugin with the RAM Cache Manager

At this point values will be cached inside the RAM Cache Manager. The effect can be viewed easily on the "Statistics" tab in the Cache Manager ZMI view, which shows which plugins have stored values, how many values are stored, an approximation of the memory consumption for the cached data, and how often the data has been retrieved from cache.

The Statistics view also provides an easy way to summarily invalidate cached values if needed. However, cache invalidation should be handled by the plugins itself if it is possible.

The PluggableAuthService itself is also cacheable this way. Caching PAS itself is the easiest way to achieve caching. In PAS, the _findUsers and _verifyUser methods, being a suitably central spot for caching, have been enhanced to use the RAM Cache Manager if so configured.

3.3.2 How does a plugin programmer use it?

Due to the pluggable (and thus infinitely variable) nature of the PluggableAuthService it is up to the plugin developer to decide what to cache and how to do so. Some of the built-in plugins provide a sample implementation by caching certain methods' return values and invalidating these records where necessary. In a nutshell, these are the steps needed to enable cacheability at the plugin level:

- Add the Cacheable mix-in class to the list of classes your plugin subclasses from
- Determine which method calls should have their return values cached, and which method calls affect the return value of the method that is being cached and thus should invalidate the cache
- In the cached method, add code to try and look up the return value in the cache first, and only perform the computation if the cache does not have the desired data. At the end of computing the return value, add it to the cache
- Add cache invalidation to those methods deemed to affect the cached method's return value.

A little illustration using code snippets:

```
from OFS.Cache import Cacheable
from Products.PluggableAuthService.plugins.BasePlugin import BasePlugin
from Products.PluggableAuthService.utils import createViewName

class MyPlugin(BasePlugin, Cacheable):

    def retrieveData(self, key):
        """ Get data for the given key """
        view_name = createViewName('retrieveData', key)
        keywords = {'key': key}
        cached_info = self.ZCacheable_get( view_name=view_name
                                         , keywords=keywords
```

(continues on next page)

(continued from previous page)

```
                                        , default=None
                                        )

        if cached_info is not None:
            return cached_info

        return_value = <perform return value calculation here>

        # Put the computed value into the cache
        self.ZCacheable_set( return_value
                           , view_name=view_name
                           , keywords=keywords
                           )

        return return_value

    def change_data(self, key, new_value):
        """ Change the value for the given key """
        <perform changes here>

        # Also, remove from the cache
        view_name = createViewName('retrieveData', key)
        self.ZCacheable_invalidate(view_name=view_name)
```

As you can see, due to the variable nature of plugins certain items, such as the relationships between the different accessor and mutator methods in use, cannot be computed or guessed, they have to be hardcoded. That's why inside change_data the view_name "retrieveData" is hardcoded as the caching key for information returned from the retrieveData method.

This example also shows how, due to the way the built-in Zope caching framework handles cached data, it is not possible to invalidate specific entries, such as the value for one specific view_name/key combination. All cached records for a specific view_name are invalidated at once.

It must be kept in mind that it is very hard if not impossible to reach a state where the cache is 100% synchronized with the live data in those situations where information is retrieved and manipulated in more than one plugin. Imagine a situation where one plugin's cached answer is dependent on another plugin that handles updating the underlying data. The retrieving plugin is not notified of updates happening in the "mutator plugin" (and thus does not invalidate cached data) unless the plugin developer forces some nasty cross-plugin dependencies.

The PluggableAuthService has two "sample implementations" for plugin caching. Both the DynamicGroupsPlugin and the ZODBUserManager are cache-enabled.

3.3.3 CAVEATS

The Caching mechanism should not be used to cache persistent objects. So if for some reason your PluggableAuthService emits user objects that are persistent (which is not the default) you should not enable caching at the PluggableAuthService-level.

CHAPTER 4

API documentation

API documentation for `Products.PluggableAuthService`.

4.1 User objects

interface `Products.PluggableAuthService.interfaces.authservice.`**IBasicUser**

Specify the interface called out in AccessControl.User.BasicUser as the "Public User object interface", except that '_getPassword' is *not* part of the contract!

getId()

Get the ID of the user.

o **The ID can be used, at least from Python, to get the user from** the user's UserDatabase

getUserName()

Return the name used by the user to log into the system.

o **Note that this may not be identical to the user's 'getId'** (to allow users to change their login names without changing their identity).

getRoles()

Return the roles assigned to a user "globally".

getRolesInContext(*object*)

Return the roles assigned to the user in context of 'object'.

o **Roles include both global roles (ones assigned to the user** directly inside the user folder) and local roles (assigned in context of the passed in object.

getDomains()

Return the list of domain restrictions for a user.

interface `Products.PluggableAuthService.interfaces.authservice.`**IPropertiedUser**

Extends: *Products.PluggableAuthService.interfaces.authservice.IBasicUser*

A user which has property sheets associated with it, i.e. a mapping from strings (property sheet ids) to objects implementing IPropertySheet

addPropertysheet(*id*, *data*)
Add a new property sheet to the user.

The property sheet has to be a map or an IPropertySheet instance.

listPropertysheets()
Return a sequence of property sheet ids

o for each id in the list getPropertysheet(id) returns a IPropertySheet

getPropertysheet(*id*)
Return a property sheet for the given id

o the returned object implements IPropertySheet and has the same id as the value passed to this method

o if there is no property sheet for the given id, raise a KeyError

An alternative way to get the property sheet is via item access, i.e. user.getPropertysheet(id) == user[id]

4.2 User property sheets

interface Products.PluggableAuthService.interfaces.propertysheets.**IPropertySheet**
Interface for queryable property sheets.

o Objects implementing this interface can play in read-only fashion in OFS.PropertySheets' framework.

getId()
Identify the sheet within a collection.

hasProperty(*id*)
Does the sheet have a property corresponding to 'id'?

getProperty(*id*, *default=None*)
Return the value of the property corresponding to 'id'.

o If no such property exists within the sheet, return 'default'.

getPropertyType(*id*)
Return the string identifying the type of property, 'id'.

o If no such property exists within the sheet, return None.

propertyInfo(*id*)
Return a mapping describing property, 'id'.

o Keys must include:

'id' – the unique identifier of the property.

'type' – the string identifying the property type.

'meta' – a mapping containing additional info about the property.

propertyMap()
Return a tuple of 'propertyInfo' mappings, one per property.

propertyIds()
Return a sequence of the IDs of the sheet's properties.

propertyValues()
Return a sequence of the values of the sheet's properties.

propertyItems()
Return a sequence of (id, value) tuples, one per property.

4.3 User Folder objects

interface Products.PluggableAuthService.interfaces.authservice.**IUserFolder**

Specify the interface called out in AccessControl.User.BasicUserFolder as the "Public UserFolder object interface":

o N.B: "enumeration" methods ('getUserNames', 'getUsers') are *not* part of the contract! See IEnumerableUserFolder.

getUser(*name*)
Return the named user object or None.

getUserById(*id*, *default=None*)
Return the user corresponding to the given id.

o If no such user can be found, return 'default'.

validate(*request*, *auth=''*, *roles=[]*)
Perform identification, authentication, and authorization.

o Return an IUser-conformant user object, or None if we can't identify / authorize the user.

o 'request' is the request object

o 'auth' is any credential information already extracted by the caller

o roles is the list of roles the caller

interface Products.PluggableAuthService.interfaces.authservice.**IPluggableAuthService**
Extends: *Products.PluggableAuthService.interfaces.authservice.IUserFolder*

The full, default contract for the pluggable authentication service.

searchUsers(***kw*)
Search for users. Returns a sequence of dicts, each dict representing a user matching the query, with the keys 'userid','id', 'login', 'title', and 'principal_type', possibly among others. 'principal_type' is always 'user'.

Possible keywords include the following:

o id: user id

o name: user name

o max_results: an int (or value castable to int) indicating the maximum number of results the method should return

o sort_by: the key in the user dictionary that should be used to sort the results

o login: user login

searchGroups(***kw*)
Search for groups. Returns a sequence of dicts, each dict representing a group matching the query, with the keys 'groupid','id', 'title', and 'principal_type', possibly among others. 'principal_type' is always 'group'.

Possible keywords include the following:

o id: user id

o name: user name

o max_results: an int (or value castable to int) indicating the maximum number of results the method should return

o sort_by: the key in the user dictionary that should be used to sort the results

searchPrincipals(*groups_first=False*, ***kw*)
Search for principals (users, groups, or both). Returns a sequence of dicts, each dict representing a principal (group or user) matching the query. groups will be represented with dictionaries as described in searchGroups, and users as described in searchUsers. Possible keywords include id, name, max_results, sort_by, and login.

updateCredentials(*request*, *response*, *login*, *new_password*)
Central updateCredentials method

This method is needed for cases where the credentials storage and the credentials extraction is handled by different plugins. Example case would be if the CookieAuthHelper is used as a Challenge and Extraction plugin only to take advantage of the login page feature but the credentials are not stored in the CookieAuthHelper cookie but somewhere else, like in a Session.

logout(*REQUEST*)
Publicly accessible method to log out a user. A wrapper around resetCredentials that may implement some policy (the default implementation redirects to HTTP_REFERER).

resetCredentials(*request*, *response*)
Reset credentials by informing all active resetCredentials plugins

updateLoginName(*user_id*, *login_name*)
Update login name of user.

updateOwnLoginName(*login_name*)
Update own login name of authenticated user.

updateAllLoginNames(*quit_on_first_error=True*)
Update login names of all users to their canonical value.

This should be done after changing the login_transform property of PAS.

You can set quit_on_first_error to False to report all errors before quitting with an error. This can be useful if you want to know how many problems there are, if any.

interface Products.PluggableAuthService.interfaces.authservice.**IMutableUserFolder**

Specify the interface called out in AccessControl.User.BasicUserFolder as the "Public UserFolder object interface":

o N.B: "enumeration" methods ('getUserNames', 'getUsers') are *not* part of the contract! See IEnumerableUserFolder.

userFolderAddUser(*name*, *password*, *roles*, *domains*, ***kw*)
Create a new user object.

userFolderEditUser(*name*, *password*, *roles*, *domains*, ***kw*)
Change user object attributes.

userFolderDelUsers(*names*)
Delete one or more user objects.

interface Products.PluggableAuthService.interfaces.authservice.**IEnumerableUserFolder**
Extends: *Products.PluggableAuthService.interfaces.authservice.IUserFolder*

Interface for user folders which can afford to enumerate their users.

getUserNames()
Return a list of usernames.

getUsers()
Return a list of user objects.

4.4 Plugins

interface Products.PluggableAuthService.interfaces.plugins.**IExtractionPlugin**
Extracts login name and credentials from a request.

extractCredentials(*request*)
request -> {...}

o Return a mapping of any derived credentials.

o Return an empty mapping to indicate that the plugin found no appropriate credentials.

interface Products.PluggableAuthService.interfaces.plugins.**ILoginPasswordExtractionPlugin**
Extends: *Products.PluggableAuthService.interfaces.plugins.IExtractionPlugin*

Common-case derivative.

extractCredentials(*request*)

request -> {'login': login, 'password': password, k1: v1, ... , kN: vN} | empty dict

o If credentials are found, the returned mapping will contain at least 'login' and 'password' keys, with the password in plaintext.

o Return an empty mapping to indicate that the plugin found no appropriate credentials.

interface Products.PluggableAuthService.interfaces.plugins.**ILoginPasswordHostExtractionPlug**
Extends: *Products.PluggableAuthService.interfaces.plugins.ILoginPasswordExtractionPlugin*

Common-case derivative.

extractCredentials(*request*)

request -> { 'login' [login] , 'password' : password , 'remote_host' : remote_host , 'remote_addr' : remote_addr , k1 : v1 , ... , kN : vN } | empty dict

o If credentials are found, the returned mapping will contain at least 'login', 'password', 'remote_host' and 'remote_addr' keys, with the password in plaintext.

o Return an empty mapping to indicate that the plugin found no appropriate credentials.

interface Products.PluggableAuthService.interfaces.plugins.**IAuthenticationPlugin**
Map credentials to a user ID.

authenticateCredentials(*credentials*)
credentials -> (userid, login)

o 'credentials' will be a mapping, as returned by IExtractionPlugin.

o Return a tuple consisting of user ID (which may be different from the login name) and login

o If the credentials cannot be authenticated, return None.

interface Products.PluggableAuthService.interfaces.plugins.**IChallengePlugin**
Initiate a challenge to the user to provide credentials.

Challenge plugins have an attribute 'protocol' representing the protocol the plugin operates under, defaulting to None.

Plugins operating under the same protocol will all be given an attempt to fire. The first plugin of a protocol group that successfully fires establishes the protocol of the overall challenge.

challenge(*request*, *response*)
Assert via the response that credentials will be gathered.

Takes a REQUEST object and a RESPONSE object.

Returns True if it fired, False otherwise.

Two common ways to initiate a challenge:

- Add a 'WWW-Authenticate' header to the response object.

  NOTE: add, since the HTTP spec specifically allows for more than one challenge in a given response.

- Cause the response object to redirect to another URL (a login form page, for instance)

interface Products.PluggableAuthService.interfaces.plugins.**ICredentialsUpdatePlugin**
Callback: user has changed her password.

This interface is not responsible for the actual password change, it is used after a successful password change event.

updateCredentials(*request*, *response*, *login*, *new_password*)
Scribble as appropriate.

interface Products.PluggableAuthService.interfaces.plugins.**ICredentialsResetPlugin**
Callback: user has logged out.

resetCredentials(*request*, *response*)
Scribble as appropriate.

interface Products.PluggableAuthService.interfaces.plugins.**IUserAdderPlugin**
Create a new user record in a User Manager

doAddUser(*login*, *password*)

Add a user record to a User Manager, with the given login and password. It is up to the implementation to determine if the login is used as user id as well.

o Return a Boolean indicating whether a user was added or not

interface Products.PluggableAuthService.interfaces.plugins.**IRoleAssignerPlugin**
Assign a role to an identified principal

doAssignRoleToPrincipal(*principal_id*, *role*)
Create a principal/role association in a Role Manager

o Return a Boolean indicating whether the role was assigned or not

doRemoveRoleFromPrincipal(*principal_id*, *role*)
Remove a principal/role association from a Role Manager

o Return a Boolean indicating whether the role was removed or not

interface Products.PluggableAuthService.interfaces.plugins.**IUserFactoryPlugin**
Create a new IPropertiedUser.

createUser(*user_id*, *name*)
Return a user, if possible.

o Return None to allow another plugin, or the default, to fire.

interface Products.PluggableAuthService.interfaces.plugins.**IAnonymousUserFactoryPlugin**
Create a new anonymous IPropertiedUser.

createAnonymousUser()
Return an anonymous user, if possible.

o Return None to allow another plugin, or the default, to fire.

interface Products.PluggableAuthService.interfaces.plugins.**IPropertiesPlugin**
Return a property set for a user.

getPropertiesForUser(*user*, *request=None*)
user -> empty dict

o User will implement IPropertiedUser.

o Plugin should return a dictionary or an object providing IPropertySheet.

o Plugin may scribble on the user, if needed (but must still return a mapping, even if empty).

o May assign properties based on values in the REQUEST object, if present

interface Products.PluggableAuthService.interfaces.plugins.**IGroupsPlugin**
Determine the groups to which a user belongs.

getGroupsForPrincipal(*principal*, *request=None*)
principal -> (group_1, ... group_N)

o Return a sequence of group names to which the principal (either a user or another group) belongs.

o May assign groups based on values in the REQUEST object, if present

interface Products.PluggableAuthService.interfaces.plugins.**IRolesPlugin**
Determine the (global) roles which a user has.

getRolesForPrincipal(*principal*, *request=None*)
principal -> (role_1, ... role_N)

o Return a sequence of role names which the principal has.

o May assign roles based on values in the REQUEST object, if present.

interface Products.PluggableAuthService.interfaces.plugins.**IUpdatePlugin**
Allow the user or the application to update the user's properties.

updateUserInfo(*user*, *set_id*, *set_info*)
Update backing store for 'set_id' using 'set_info'.

interface Products.PluggableAuthService.interfaces.plugins.**IValidationPlugin**
Specify allowable values for user properties.

o E.g., constrain minimum password length, allowed characters, etc.

o Operate on entire property sets, not individual properties.

validateUserInfo(*user*, *set_id*, *set_info*)
-> (error_info_1, ... error_info_N)

o Returned values are dictionaries, containing at least keys:

'id' – the ID of the property, or None if the error is not specific to one property.

'error' – the message string, suitable for display to the user.

interface Products.PluggableAuthService.interfaces.plugins.**IUserEnumerationPlugin**
Allow querying users by ID, and searching for users.

o ???: can these be done by a single plugin?

enumerateUsers(*id=None*, *login=None*, *exact_match=False*, *sort_by=None*, *max_results=None*, ***kw*)
-> (user_info_1, ... user_info_N)

o Return mappings for users matching the given criteria.

o 'id' or 'login', in combination with 'exact_match' true, will return at most one mapping per supplied ID ('id' and 'login' may be sequences).

o If 'exact_match' is False, then 'id' and / or login may be treated by the plugin as "contains" searches (more complicated searches may be supported by some plugins using other keyword arguments).

o If 'sort_by' is passed, the results will be sorted accordingly. known valid values are 'id' and 'login' (some plugins may support others).

o If 'max_results' is specified, it must be a positive integer, limiting the number of returned mappings. If unspecified, the plugin should return mappings for all users satisfying the criteria.

o Minimal keys in the returned mappings:

'id' – (required) the user ID, which may be different than the login name

'login' – (required) the login name

'pluginid' – (required) the plugin ID (as returned by getId())

'editurl' – (optional) the URL to a page for updating the mapping's user

o Plugin *must* ignore unknown criteria.

o Plugin may raise ValueError for invalid criteria.

o Insufficiently-specified criteria may have catastrophic scaling issues for some implementations.

updateUser(*user_id*, *login_name*)
Update the login name of the user with id user_id.

The plugin must return True (or any truth value) to indicate a successful update, also when no update was needed.

When updating a login name makes no sense for a plugin (most likely because it does not actually store login names) and it does not do anything, it must return None or False.

updateEveryLoginName(*quit_on_first_error=True*)
Update login names of all users to their canonical value.

This should be done after changing the login_transform property of PAS.

You can set quit_on_first_error to False to report all errors before quitting with an error. This can be useful if you want to know how many problems there are, if any.

interface Products.PluggableAuthService.interfaces.plugins.**IGroupEnumerationPlugin**
Allow querying groups by ID, and searching for groups.

o ???: can these be done by a single plugin?

enumerateGroups(*id=None*, *exact_match=False*, *sort_by=None*, *max_results=None*, ***kw*)
-> (group_info_1, ... group_info_N)

o Return mappings for groups matching the given criteria.

o 'id' in combination with 'exact_match' true, will return at most one mapping per supplied ID ('id' may be a sequence).

o **If 'exact_match' is False, then 'id' may be treated by** the plugin as "contains" searches (more complicated searches may be supported by some plugins using other keyword arguments).

o **If 'sort_by' is passed, the results will be sorted accordingly.** known valid values are 'id' (some plugins may support others).

o **If 'max_results' is specified, it must be a positive integer,** limiting the number of returned mappings. If unspecified, the plugin should return mappings for all groups satisfying the criteria.

o Minimal keys in the returned mappings:

'id' – (required) the group ID

'pluginid' – (required) the plugin ID (as returned by getId())

'properties_url' – (optional) the URL to a page for updating the group's properties.

'members_url' – (optional) the URL to a page for updating the principals who belong to the group.

o Plugin *must* ignore unknown criteria.

o Plugin may raise ValueError for invalid critera.

o **Insufficiently-specified criteria may have catastrophic** scaling issues for some implementations.

interface Products.PluggableAuthService.interfaces.plugins.**IRoleEnumerationPlugin**
Allow querying roles by ID, and searching for roles.

enumerateRoles(*id=None*, *exact_match=False*, *sort_by=None*, *max_results=None*, ***kw*)
-> (role_info_1, ... role_info_N)

o Return mappings for roles matching the given criteria.

o **'id' in combination with 'exact_match' true, will** return at most one mapping per supplied ID ('id' may be a sequence).

o **If 'exact_match' is False, then 'id' may be treated by** the plugin as "contains" searches (more complicated searches may be supported by some plugins using other keyword arguments).

o **If 'sort_by' is passed, the results will be sorted accordingly.** known valid values are 'id' (some plugins may support others).

o **If 'max_results' is specified, it must be a positive integer,** limiting the number of returned mappings. If unspecified, the plugin should return mappings for all roles satisfying the criteria.

o Minimal keys in the returned mappings:

'id' – (required) the role ID

'pluginid' – (required) the plugin ID (as returned by getId())

'properties_url' – (optional) the URL to a page for updating the role's properties.

'members_url' – (optional) the URL to a page for updating the principals to whom the role is assigned.

o Plugin *must* ignore unknown criteria.

o Plugin may raise ValueError for invalid critera.

o **Insufficiently-specified criteria may have catastrophic** scaling issues for some implementations.

interface Products.PluggableAuthService.interfaces.plugins.**IRequestTypeSniffer**
Given a request, detects the request type for later use by other plugins.

sniffRequestType(*request*)
Return a interface identifying what kind the request is.

interface Products.PluggableAuthService.interfaces.plugins.**IChallengeProtocolChooser**
Choose a proper set of protocols to be used for challenging the client given a request.

chooseProtocols(*request*)
-> (protocol_1, ... protocol_N) | None

o **If a set of protocols is returned, the first plugin with a** protocol that is in the set will define the protocol to be used for the current request.

o If None is returned, the 'first found protocol' wins.

o **Once the protocol is decided, all challenge plugins for that** protocol will be executed.

interface Products.PluggableAuthService.interfaces.plugins.**INotCompetentPlugin**
check whether this user folder is not competent to authenticate.

Never used for a top level user folder; primarily used to prevent shadowing of authentications by higher level user folders.

isNotCompetentToAuthenticate(*request*)
return true if this user folder should not authenticate *request*.

4.5 Events

interface Products.PluggableAuthService.interfaces.events.**IPASEvent**
An event related to a PAS principal.

principal
The subject of the event.

interface Products.PluggableAuthService.interfaces.events.**IPrincipalCreatedEvent**
Extends: *Products.PluggableAuthService.interfaces.events.IPASEvent*

A new principal has been created.

interface Products.PluggableAuthService.interfaces.events.**IUserLoggedInEvent**
Extends: *Products.PluggableAuthService.interfaces.events.IPASEvent*

A user logged in.

interface Products.PluggableAuthService.interfaces.events.**IUserLoggedOutEvent**
Extends: *Products.PluggableAuthService.interfaces.events.IPASEvent*

A user logged out.

interface Products.PluggableAuthService.interfaces.events.**IPrincipalDeletedEvent**
Extends: *Products.PluggableAuthService.interfaces.events.IPASEvent*

A user has been removed.

interface Products.PluggableAuthService.interfaces.events.**ICredentialsUpdatedEvent**
Extends: *Products.PluggableAuthService.interfaces.events.IPASEvent*

A principal has changed her password.

Sending this event will cause a PAS user folder to trigger its active credential update plugins.

password
The new password

interface Products.PluggableAuthService.interfaces.events.**IPropertiesUpdatedEvent**
Extends: *Products.PluggableAuthService.interfaces.events.IPASEvent*

A principals properties have been updated.

properties
List of modified property ids

interface Products.PluggableAuthService.interfaces.events.**IGroupDeletedEvent**
Extends: *Products.PluggableAuthService.interfaces.events.IPASEvent*

A group has been removed.

4.6 Requests

interface Products.PluggableAuthService.interfaces.request.**IRequest**
Base Request Interface

??? Add methods from BaseRequest?

interface Products.PluggableAuthService.interfaces.request.**IHTTPRequest**
Extends: *Products.PluggableAuthService.interfaces.request.IRequest*

HTTP Request

interface Products.PluggableAuthService.interfaces.request.**IBrowserRequest**
Extends: *Products.PluggableAuthService.interfaces.request.IHTTPRequest*

Browser Request

interface Products.PluggableAuthService.interfaces.request.**IWebDAVRequest**
Extends: *Products.PluggableAuthService.interfaces.request.IHTTPRequest*

WebDAV Request

interface Products.PluggableAuthService.interfaces.request.**IXMLRPCRequest**
Extends: *Products.PluggableAuthService.interfaces.request.IHTTPRequest*

XML-RPC Request

interface Products.PluggableAuthService.interfaces.request.**IFTPRequest**
Extends: *Products.PluggableAuthService.interfaces.request.IRequest*

FTP Request

CHAPTER 5

Changes

5.1 Change Log

5.1.1 2.2 (unreleased)

- Nothing changed yet.

5.1.2 2.1.1 (2019-10-23)

- Fix bug in `getRolesForPrincipal` for non PAS user.

5.1.3 2.1 (2019-08-29)

- Fix formatting in "Plugin Types" documentation.
- Fixed error assigning roles in `manage_roles` page in ZMI. See issues #43 and #51.

5.1.4 2.0 (2019-05-10)

- Drop unused `.utils.allTests` method.

5.1.5 2.0b6 (2019-04-17)

- fixed usage of deprecated `im_self` (#40)

5.1.6 2.0b5 (2019-04-13)

- fixed the "Configured PAS" factory (#39)
- styled "Configured PAS" add dialog for the Zope 4 ZMI (#38)
- prevent the ZMI add dialog showing in the Zope 4 ZMI (#37)
- added the indirect dependency `Products.Sessions` for the CSRF-support

5.1.7 2.0b4 (2019-04-04)

- simplified Travis CI test configuration
- added stricter linting configuration
- added `project_urls` to the setup so PyPI shows more relevant links
- added project badges to the README, which will show on the GitHub front page
- Fix ZMI Templates and add ZMI icons for Zope 4 (#36)

5.1.8 2.0b3 (2019-03-29)

- Fixed Dynamic Groups Plugin ZMI view (#33)
- Re-enabled XML-RPC support without requiring ZServer (#34)
- Specify supported Python versions using `python_requires` in setup.py
- Added support for Python 3.8
- Fix CSRF defense incompatibility with some session implementations

5.1.9 2.0b2 (2018-10-16)

- Add support for Python 3.7.
- Do not override a previously set response body in `HTTPBasicAuthHelper.challenge()` allowing to set the response body via an exception view in Zope >= 4.0b6.
- Add new event to be able to notify group creation.
- Refactoring to make it easier to override `updateCredentials`.

5.1.10 2.0b1 (2018-05-18)

- The dependency on `ZServer` is now optional. To use the features which require `ZServer` (WebDav, XML-RPC, FTP) use the setuptools extra *zserver* when installing the package.
- Do not fail when our base profiles are already registered. This may happen in tests if our `initialize` code is called twice.
- Add support for Python 3.
- Reformatted code for PEP-8 compliance.
- Require Zope 4.0b5 as minimum Zope version.

5.1.11 1.11.0 (2016-03-01)

- Add new event to be able to notify group deletion.
- Fix usage of os.path.split(). Could result in Errors during import on Windows.

5.1.12 1.10.0 (2013-02-19)

- Allow specifying a policy for transforming / normalizing login names for all plugins in a PAS:
  - Added `login_transform` string property to PAS.
  - Added `applyTransform` method to PAS, which looks for a method on PAS with the name specified in the `login_transform` property.
  - Added two possible transforms to PAS: `lower` and `upper`.
  - Changed the methods of PAS to call `applyTransform` wherever needed.
  - Added the existing `updateUser` method of ZODBUserManager to the IUserEnumerationPlugin interface.
  - Added a new `updateEveryLoginName` method to ZODBUserManager and the IUserEnumerationPlugin interface.
  - Added three methods to PAS and IPluggableAuthService: `updateLoginName`, `updateOwnLoginName`, `updateAllLoginNames`. These methods call `updateUser` or `updateEveryLoginName` on every IUserEnumerationPlugin. Since these are later additions to the plugin interface, we log a warning when a plugin does not have these methods (for example the `mutable_properties` plugin of PlonePAS) but will not fail. When no plugin is able to update a user, this will raise an exception: we do not want to quietly let this pass when for example a login name is already taken by another user.
  - Changing the `login_transform` property in the ZMI will call `PAS.updateAllLoginNames`, unless `login_transform` is the same or has become an empty string.
  - The new `login_transform` property is empty by default. In that case, the behavior of PAS is the same as previously. The various `applyTransform` calls will have a (presumably very small) performance impact.
- Launchpad #1079204: Added CSRF protection for the ZODBUserManager, ZODBGroupManager, ZODBRoleManger, and DynamicGroupsPlugin plugins.

5.1.13 1.9.0 (2012-08-30)

- Launchpad #649596: add a protocol for plugins which check whether a non-top-level PAS instance is "competent" to authenticate a given request; if not, the instance defers to higher-level instances. Thanks to Dieter Maurer for the patch.

5.1.14 1.8.0 (2012-05-08)

- Added export / import support for the ChallengeProtocolChooser plugin's label - protocols mapping.

5.1.15 1.7.8 (2012-05-08)

- In authenticateCredentials do NOT fall back to using the login as userid when there is no match, as that gives a high chance of seeming to log in successfully, but in reality failing. [maurits]

5.1.16 1.7.7 (2012-02-27)

- Explicitly encode/decode data for GS

5.1.17 1.7.6 (2011-10-31)

- Launchpad #795086: fixed creation of PropertiesUpdated event.

5.1.18 1.7.5 (2011-05-30)

- Launchpad #789858: don't allow conflicting login name in 'updateUser'.
- Set appropriate cache headers on CookieAuthHelper login redirects to prevent caching by proxy servers.

5.1.19 1.7.4 (2011-05-13)

- Added forward compatibility with DateTime 3.

5.1.20 1.7.3 (2011-02-10)

- In the ZODBRoleManager made it clearer that adding a removing a role does not have much effect if you do not do the same in the root of the site (at the bottom of the Security tab at manage_access). Fixes https://bugs.launchpad.net/zope-pas/+bug/672694
- Return the created user in _doAddUser, to match change in AccessControl 2.13.4.
- Fixed possible `binascii.Error` in `extractCredentials` of CookieAuthHelper. This is a corner case that might happen after a browser upgrade.

5.1.21 1.7.2 (2010-11-11)

- Allow for a query string in CookieAuthHelper's `login_path`.
- Trap "swallowable" exceptions from `IRoles` plugins. Thanks to Willi Langenburger for the patch. Fixes https://bugs.launchpad.net/zope-pas/+bug/615474 .
- Fixed possible TypeError in `extractCredentials` of CookieAuthHelper when the `__ac` cookie is not ours (but e.g. from plone.session, though even then only in a corner case).
- Fixed chameleon incompatibilities

5.1.22 1.7.1 (2010-07-01)

- Made `ZODBRoleManager.assignRoleToPrincipal` raise and log a more informative error when detecting a duplicate principal. https://bugs.launchpad.net/zope-pas/+bug/348795
- Updated `DynamicGroupsPlugin.enumerateGroups` to return an empty sequence for an unknown group ID, rather than raising KeyError. https://bugs.launchpad.net/zope-pas/+bug/585365
- Updated all code to raise new-style exceptions.
- Removed dependency on `zope.app.testing`.
- Cleaned out a number of old imports, because we now require Zope >= 2.12.
- Updated `setDefaultRoles` to use the `addPermission` API if available.

5.1.23 1.7.0 (2010-04-08)

- Allow CookieAuthHelper's `login_path` to be set to an absolute url for integration with external authentication mechanisms.
- Fixed xml templates directory path computation to allow reuse of `SimpleXMLExportImport` class outside `Products.PluggableAuthService`.

5.1.24 1.7.0b2 (2010-01-31)

- Modify ZODBGroupManager to update group title and description independently.

5.1.25 1.7.0b1 (2009-11-16)

- This release requires for Zope2 >= 2.12.
- Simplified buildout to just what is needed to run tests.
- Don't fail on users defined in multiple user sources on the ZODBGroupManager listing page.
- Fixed deprecation warnings for use of `Globals` under Zope 2.12.
- Fixed deprecation warnings for the `md5` and `sha` modules under Python >= 2.6.
- Added test for multiple auth header support in the HTTPBasicAuthHelper.
- Changed HTTPBasicAuthHelper to not rely on one obscure feature of the HTTPResponse.

5.1.26 1.6.2 (2009-11-16)

- Launchpad #420319: Fix misconfigured `startswith` match type filter in `Products.PluggableAuthService.plugins.DomainAuthHelper`.
- Fixed test setup for tests using page templates relying on the `DefaultTraversable` adapter.
- Fixed broken markup in templates.

5.1.27 1.6.1 (2008-11-20)

- Launchpad #273680: Avoid expensive / incorrect dive into `enumerateUsers` when trying to validate w/o either a real ID or login.
- Launchpad #300321: `Products.PluggableAuthService.pluginsZODBGroupManager.enumerateGroups` failed to find groups with unicode IDs.

5.1.28 1.6 (2008-08-05)

- Fixed another deprecation for `manage_afterAdd` occurring when used together with Five (this time for the `ZODBRoleManager` class).
- Ensure the `_findUser` cache is invalidated if the roles or groups for a principal change.
- Launchpad #15569586: docstring fix.
- Factored out `filter` logic into separate classes; added filters for `startswith` test and (if the IPy module is present) IP-range tests. See https://bugs.launchpad.net/zope-pas/+bug/173580 .
- Zope 2.12 compatibility - removed `Interface.Implements` import if `zope.interface` available.
- Ensure `ZODBRoleManagerExportImport` doesn't fail if it tries to add a role that already exists (idempotence is desirable in GS importers)
- Fixed tests so they run with Zope 2.11.
- Split up large permission tests into individual tests.
- Fixed deprecation warning occurring when used together with Five. (`manage_afterAdd` got undeprecated.)
- Added buildout.

5.1.29 1.5.3 (2008-02-06)

- ZODBUserManager plugin: allow unicode arguments to `enumerateUsers`. (https://bugs.launchpad.net/zope-pas/+bug/189627)
- plugins/ZODBRoleManager: added logging in case searchPrincipial() returning more than one result (which might happen in case of having duplicate id within difference user sources)

5.1.30 1.5.2 (2007-11-28)

- DomainAuthHelper plugin: fix glitch for plugins which have never configured any "default" policy: `authenticateCredentials` and `getRolesForPrincipal` would raise ValueError. (http://www.zope.org/Collectors/PAS/59)

5.1.31 1.5.1 (2007-09-11)

- PluggableAuthService._verifyUser: changed to use exact_match to the enumerator, otherwise a user with login `foobar` might get returned by _verifyUser for a query for `login='foo'` because the enumerator happened to return 'foobar' first in the results.
- Add a test for manage_zmi_logout and replace a call to isImplementedBy with providedBy. (http://www.zope.org/Collectors/PAS/58)

5.1.32 1.5 (2006-06-17)

- Add support for property plugins returning an IPropertySheet to PropertiedUser. Added addPropertysheet to the IPropertiedUser.
- Added a method to the IRoleAssignerPlugin to remove roles from a principal, and an implementation for it on the ZODBRoleManager. (http://www.zope.org/Collectors/PAS/57)
- Added events infrastructure. Enabled new IPrincipalCreatedEvent and ICredentialsUpdatedEvent events.
- Added support for registering plugin types via ZCML.
- Implemented authentication caching in _extractUserIds.
- Ported standard user folder tests from the AccessControl test suite.
- Passwords with ":" characters would break authentication (http://www.zope.org/Collectors/PAS/51)
- Corrected documented software dependencies
- Converted to publishable security sensitive methods to only accept POST requests to prevent XSS attacks. See http://www.zope.org/Products/Zope/Hotfix-2007-03-20/announcement and http://dev.plone.org/plone/ticket/6310
- Fixed issue in the user search filter where unrecognized keyword arguments were ignored resulting in duplicate search entries. (http://dev.plone.org/plone/ticket/6300)
- Made sure the Extensions.upgrade script does not commit full transactions but only sets (optimistic) savepoints. Removed bogus Zope 2.7 compatibility in the process. (http://www.zope.org/Collectors/PAS/55)
- Made the CookieAuthHelper only use the `__ac_name` field if `__ac_password` is also present. This fixes a login problem for CMF sites where the login name was remembered between sessions with an `__ac_name` cookie.
- Made the DomainAuthHelper return the remote address, even it the remote host is not available (http://www.zope.org/Collectors/PAS/49).
- Fixed bug in DelegatingMultiPlugin which attempted to validate the supplied password directly against the user password - updated to use AuthEncoding.pw_validate to handle encoding issues
- Fixed serious security hole in DelegatingMultiPlugin which allowed Authentication if the EmergencyUser login was passed in. Added password validation utilizing AuthEncoding.pw_validate
- Fixed a set of tests that tested values computed from dictionaries and could break since dictionaries are not guaranteed to have any sort order.
- Fixed test breakage induced by use of Z3 pagetemplates in Zope 2.10+.
- BasePlugin: The listInterfaces method only considered the old-style __implements__ machinery when determining interfaces provided by a plugin instance.
- ZODBUserManager: Already encrypted passwords were encrypted again in addUser and updateUserPassword. (http://www.zope.org/Collectors/Zope/1926)
- Made sure the emergency user via HTTP basic auth always wins, no matter how borken the plugin landscape.
- Cleaned up code in CookieAuthHelper which allowed the form to override login/password if a cookie had already been set.
- Removed some BBB code for Zope versions < 2.8, which is not needed since we require Zope > 2.8.5 nowadays.

5.1.33 1.4 (2006-08-28)

- Extended the DomainAuthHelper to function as its own extraction plugin, to allow for the case that another extractor is registered, but does not return any credentials. (http://www.zope.org/Collectors/PAS/46)
- Re-worded parts of the README so they don't point to specific or non-existing files (http://www.zope.org/Collectors/PAS/6 and http://www.zope.org/Collectors/PAS/47)

5.1.34 1.4-beta (2006-08-07)

- Created a "Configured PAS" entry in the ZMI add list, which allows creating a PAS using base and extension GenericSetup profiles registered for IPluggableAuthService. This entry should eventually replace the "stock" PAS entry (assuming that we make GenericSetup a "hard" dependency).
- Added an "empty" GenericSetup profile, which creates a PAS containing only a plugin registry and a setup tool.
- Repaired the "simple" GenericSetup profile to be useful, rather than catastrophic, to apply: it now creates and registers a set of ZODB-based user / group / role plugins, along with a basic auth helper.
- ZODBUserManager: Extend the "notional IZODBUserManager interface" with the left-out updateUser facility and a corresponding manage_updateUser method for ZMI use. Removed any responsibility for updating a user's login from the updateUserPassword and manage_updateUserPassword methods. This fixes the breakage described in the collector issue below, and makes the ZMI view for updating users work in a sane way. (http://www.zope.org/Collectors/PAS/42)
- CookieAuthHelper: If expireCookie was called and extractCredentials was hit in the same request, the CookieAuthHelper would throw an exception (http://www.zope.org/Collectors/PAS/43)
- Added a DEPENDENCIES.txt. (http://www.zope.org/Collectors/PAS/44)

5.1.35 1.3 (2006-06-09)

- No changes from version 1.3-beta

5.1.36 1.3-beta (2006-06-03)

- Modify CookieAuthHelper to prefer __ac form variables to the cookie when extracting credentials. (https://dev.plone.org/plone/ticket/5355)

5.1.37 1.2 (2006-05-14)

- Fix manage_zmi_logout which stopped working correctly as soon as the PluggableAuthService product code was installed by correcting the monkeypatch for it in __init__.py. (http://www.zope.org/Collectors/PAS/12)
- Add missing interface for IPropertiedUser and tests (http://www.zope.org/Collectors/PAS/16)
- Removed STX links from README.txt which do nothing but return 404s when clicked from the README on zope.org. (http://www.zope.org/Collectors/PAS/6)
- Fixing up inconsistent searching in the listAvailablePrincipals method of the ZODBRoleManager and ZODBGroupManager plugins. Now both constrain searches by ID. (http://www.zope.org/Collectors/PAS/11)
- Convert from using zLOG to using the Python logging module. (http://www.zope.org/Collectors/PAS/14)

5.1.38 1.2-beta (2006-02-25)

- Added suppport for exporting / importing a PAS and its content via the GenericSetup file export framework.
- Made ZODBRoleManager plugin check grants to the principal's groups, as well as those made to the principal directly.
- Added two new interfaces, IChallengeProtocolChooser and IRequestTypeSniffer. Those are used to select the 'authorization protocol' or 'challenger protocol' to be used for challenging according to the incoming request type.
- Repaired warings appearing in Zope 2.8.5 due to a couple typos in security declarations.
- Repaired DeprecationWarnings due to use of Zope2 interface verification.
- Repaired unit test breakage (unittest.TestCase instances have 'failUnless'/'failIf', rather than 'assertTrue'/'assertFalse').
- Fixed a couple more places where Zope 2-style `__implements__` were being used to standardize on using `classImplements`.
- Fixed fallback implementations of `providedBy` and `implementedBy` to always return a tuple.
- Make sure challenge doesn't break if existing instances of the PluginRegistry don't yet have `IChallengeProtocolChooser` as a registered interface. (Would be nice to have some sort of migration for the PluginRegistry between PAS releases)
- Don't assume that just because zope.interface can be imported that Five is present.

5.1.39 1.1b2 (2005-07-14)

- Repaired a missing 'nocall:' in the Interfaces activation form.

5.1.40 1.1b1 (2005-07-06)

- PAS-level id mangling is no more. All (optional) mangling is now done on a per-plugin basis.
- Interfaces used by PAS are now usable in both Zope 2.7 and 2.8 (Five compatible)

5.1.41 1.0.5 (2005-01-31)

- Simplified detection of the product directory using 'package_home'.
- Set a default value for the 'login' attribute of a PAS, to avoid UnboundLocalError.

5.1.42 1.0.4 (2005-01-27)

- Made 'Extensions' a package, to allow importing its scripts as modules.
- Declared new 'IPluggableAuthService' interface, describing additional PAS-specific API.
- Exposed PAS' 'resetCredentials' and 'updateCredentials' as public methods.
- Monkey-patch ZMI's logout to invoke PAS' 'resetCredentials', if present.
- CookieAuth plugin now encodes and decodes cookies in the same format as CookieCrumbler to provide compatibility between sites running PAS and CC.

- Add a publicly callable "logout" method on the PluggableAuthService instance that will call resetCredentials on all activated ICredentialsRest plugins, thus effecting a logout.
- Enabled the usage of the CookieAuthHelper login screen functionality without actually using the CookieAuthHelper to maintain the credentials store in its own auth cookie by ensuring that only active updateCredentials plugins are informed about a successful login so they can store the credentials.
- Added a _getPAS method to the BasePlugin base class to be used as the canonical way of getting at the PAS instance from within plugins.
- Group and user plugins can now specify their own title for a principal entry (PAS will not compute one if they do).
- PAS and/or plugins can now take advantage of caching using the Zope ZCacheable framework with RAM Cache Managers. See doc/caching.stx for the details.
- Make 'getUserById' pass the 'login' to '_findUser', so that the returned user object can answer 'getUserName' sanely.
- Harden 'logout' against missing HTTP_REFERRER.
- Avoid triggering "Emergency user cannot own" when adding a CookieAuthHelper plugin as that user.
- Detect and prevent recursive redirecting in the CookieAuthHelper if the login_form cannot be reached by the Anonymous User.
- Made logging when swallowing exceptions much less noisy (they *don't* necessarily require attention).
- Clarified interface of IAuthenticationPlugin, which should return None rather than raising an exception if asked to authenticate an unknown principal; adjusted ZODBUserManager accordingly.
- Don't log an error in zodb_user_plugin's authenticateCredentials if we don't have a record for a particular username, just return None.
- If an IAuthenticationPlugin returns None instead of a tuple from authenticateCredentials, don't log a tuple-unpack error in PAS itself.

5.1.43 1.0.3 (2004-10-16)

- Implemented support for issuing challenges via IChallengePlugins.
  - three challenge styles in particular:
    - HTTP Basic Auth
    - CookieCrumbler-like redirection
    - Inline authentication form
- Made unit tests pass when run with cAccessControl.
- plugins/ZODBRoleManager.py: don't claim authority for 'Authenticated' or 'Anonymous' roles, which are managed by PAS.
- plugins/ZODBRoleManager.py: don't freak out if a previously assigned principal goes away.
- plugins/ZODBGroupManager.py: don't freek out if a previously assigned principal goes away.
- plugins/ZODBUserManager.py: plugin now uses AuthEncoding for its password encryption so that we can more easily support migrating existing UserFolders. Since PAS has been out for a while, though, we still will authenticate against old credentials
- Repaired arrow images in two-list ZMI views.

- searchPrincipals will work for exact matches when a plugin supports both 'enumerateUsers' and 'enumerateGroups'.
- 'Authenticated' Role is now added dynamically by the PluggableAuthService, not by any role manager
- Added WARNING-level logs with tracebacks for all swallowed plugin exceptions, so that you notice that there is something wrong with the plugins.
- All authenticateCredentials() returned a single None when they could not authenticate, although all calls expected a tuple.
- The user id in extract user now calls _verifyUser to get the ID mangled by the enumeration plugin, instead of mangling it with the authentication ID, thereby allowing the authentication and enumeration plugins to be different plugins.

5.1.44 1.0.2 (2004-07-15)

- ZODBRoleManager and ZODBGroupManager needed the "two_lists" view, and associated images, which migrated to the PluginRegsitry product when they split; restored them.

5.1.45 1.0.1 (2004-05-18)

- CookieAuth plugin didn't successfully set cookies (first, because of a NameError, then, due to a glitch with long lines).
- Missing ZPL in most modules.

5.1.46 1.0 (2004-04-29)

- Initial release

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Index

H

I

L

P

R

S

U

V