repoze.who Documentation

Release 2.2

Agendaless Consulting

June 03, 2016

Contents

1 Sections **3**
- 1.1 `repoze.who` Narrative Documentation . . . 3
- 1.2 `repoze.who` Use Cases . . . 4
- 1.3 Using `repoze.who` Middleware . . . 6
- 1.4 Using the `repoze.who` Application Programming Interface (API) . . . 10
- 1.5 Configuring `repoze.who` . . . 15
- 1.6 About `repoze.who` Plugins . . . 20
- 1.7 Known Plugins for `repoze.who` . . . 26

2 Change History **29**
- 2.1 repoze.who Changelog . . . 29

3 Support and Development **39**

4 Indices and tables **41**

Python Module Index **43**

Author Chris McDonough / Tres Seaver

Version 2.2

Overview

repoze.who is an identification and authentication framework for arbitrary WSGI applications. It can be used as WSGI middleware, or as an API from within a WSGI application.
repoze.who is inspired by Zope 2's Pluggable Authentication Service (PAS) (but *repoze.who* is not dependent on Zope in any way; it is useful for any WSGI application). It provides no facility for authorization (ensuring whether a user can or cannot perform the operation implied by the request). This is considered to be the domain of the WSGI application.
It attempts to reuse implementations from `paste.auth` for some of its functionality.

CHAPTER 1

Sections

1.1 `repoze.who` Narrative Documentation

1.1.1 Using `repoze.who` as WSGI Middleware

`repoze.who` was originally developed for use as authentication middleware in a WSGI pipeline, for use by applications which only needed to obtain an "authenticated user" to enforce a given security policy.

See *Middleware Responsibilities* for a description of this use case.

1.1.2 Using `repoze.who` without WSGI Middleware

Some applications might want to use a configured set of *`repoze.who`* plugins to do identification and authentication for a request, outside the context of using *`repoze.who`* middleware. For example, a performance-sensitive application might wish to defer the effort of identifying and authenticating a user until the point at which authorization is required, knowing that some code paths will not need to do the work.

See *Using the repoze.who Application Programming Interface (API)* for a description of this use case.

1.1.3 Mixing Middleware and API Uses

Some applications might use the *`repoze.who`* middleware for most authentication purposes, but need to participate more directly in the mechanics of identification and authorization for some portions of the application. For example, consider a system which allows users to sign up online for membrship in a site: once the user completes registration, such an application might wish to log the user in transparently, and thus needs to interact with the configured *`repoze.who`* middleware to generate response headers, ensuring that the user's next request is properly authenticated.

See *Mixed Use of repoze.who Middleware and API* for a description of this use case.

1.1.4 Configuring `repoze.who`

Developers and integrators can configure *`repoze.who`* using either imperative Python code (see *Configuring repoze.who via Python Code*) or using an INI-style declarative configuration file (see *Configuring repoze.who via Config File*). In either case, the result of the configuration will be a `repoze.who.api:APIFactory` instance, complete with a request classifier, a challenge decider, and a set of plugins for each plugin interface.

1.2 repoze.who Use Cases

How should an application interact with *repoze.who*? There are three main scenarios:

1.2.1 Middleware-Only Use Cases

Examples of using the *repoze.who* middleware, without explicitly using its API.

Simple: Bug Tracker with REMOTE_USER

This application expects the REMOTE_USER variable to be set by the middleware for authenticated requests. It allows the middleware to handle challenging the user when needed.

In protected views, such as those which allow creating or following up to bug reports:

- Check environ['REMOTE_USER'] to get the authenticated user, and apply any application-specific policy (who is allowed to edit).
  - If the access check fails because the user is not yet authenticated, return an 401 Unauthorized response.
  - If the access check fails for authenticated users, return a 403 Forbidden response.

Note that the application here doesn't depend on *repoze.who* at all: it would work identically if run behind Apache's mod_auth. The Trac application works exactly this way.

The middleware can be configured to suit the policy required for the site, e.g.:

- challenge / identify using HTTP basic authentication
- authorize via an .htaccces-style file.

More complex: Wiki with repoze.who.identity

This application use the repoze.who.identity variable set in the WSGI environment by the middleware for authenticated requests. The application still allows the middleware to handle challenging the user when needed.

The only difference from the previous example is that protected views, such as those which allow adding or editing wiki pages, can use the extra metadata stored inside environ['repoze.who.identity'] (a mapping) to make authorization decisions: such metadata might include groups or roles mapped by the middleware onto the user.

1.2.2 API-Only Use Cases

Examples of using the *repoze.who* API without its middleware.

Simple: Wiki with its own login and logout views.

This application uses the *repoze.who* API to compute the authenticated user, as well as using its remember API to set headers for cookie-based authentication.

In each view:

- Call api.authenticate to get the authenticated user.
- Show a login link for non-authenticated requests.
- Show a logout link for authenticated requests.

- Don't show "protected" links for non-authenticated requests.

In protected views, such as those which allow adding or editing wiki pages:

- Call `api.authenticate` to get the authenticated user; check the metadata about the user (e.g., any appropriate roles or groups) to verify access.
  - If the access check fails because the user is not yet authenticated, redirect to the `login` view, with a `came_from` value of the current URL.
  - If the access check fails for authenticated users, return a 403 Forbidden response.

In the login view:

- For `GET` requests, show the login form.
- For `POST` requests, validate the login and password from the form. If successful, call `api.remember`, and append the returned headers to your response, which may also contain, e.g., a `Location` header for a redirect to the `came_from` URL. In this case, there will be no authenticator plugin which knows about the login / password at all.

In the logout view:

- Call `api.forget` and append the headers to your response, which may also contain, e.g., a `Location` header for a redirect to the `came_from` URL after logging out.

More complex: multiple applications with "single sign-on"

In this scenario, authentication is "federated" across multiple applications, which delegate to a central "login application." This application verifies credentials from the user, and then uses headers or other tokens to communicate the verified identity to the delegating application.

In the login application:

- The SSO login application works just like the login view described above: the difference is that the configured identifier plugins must emit headers from `remember` which can be recognized by their counterparts in the other apps.

In the non-login applications:

- Challenge plugins here must be configured to implement the specific SSO protocol, e.g. redirect to the login app with information in the query string (other protocols might differ).
- Identifer plugins must be able to "crack" / consume whatever tokens are returned by the SSO login app.
- Authenticators will normally be no-ops (e.g., the `auth_tkt` plugin used as an authenticator).

1.2.3 Hybrid Use Cases

Examples of using the *`repoze.who`* API in conjuntion with its middleware.

Most complex: integrate Trac and the wiki behind SSO

This example extends the previous one, but adds into the mix the requirement that one or more of the non-login applications (e.g., Trac) be used "off the shelf," without modifying them. Such applications can be plugged into the same SSO regime, with the addition of the :mod:`repoze.who` middleware as an adapter to bridge the gap (e.g., to turn the SSO tokens into the `REMOTE_USER` required by Trac).

In this scenario, the middleware would be configured identically to the API used in applications which do not need the middleware shim.

1.3 Using `repoze.who` Middleware

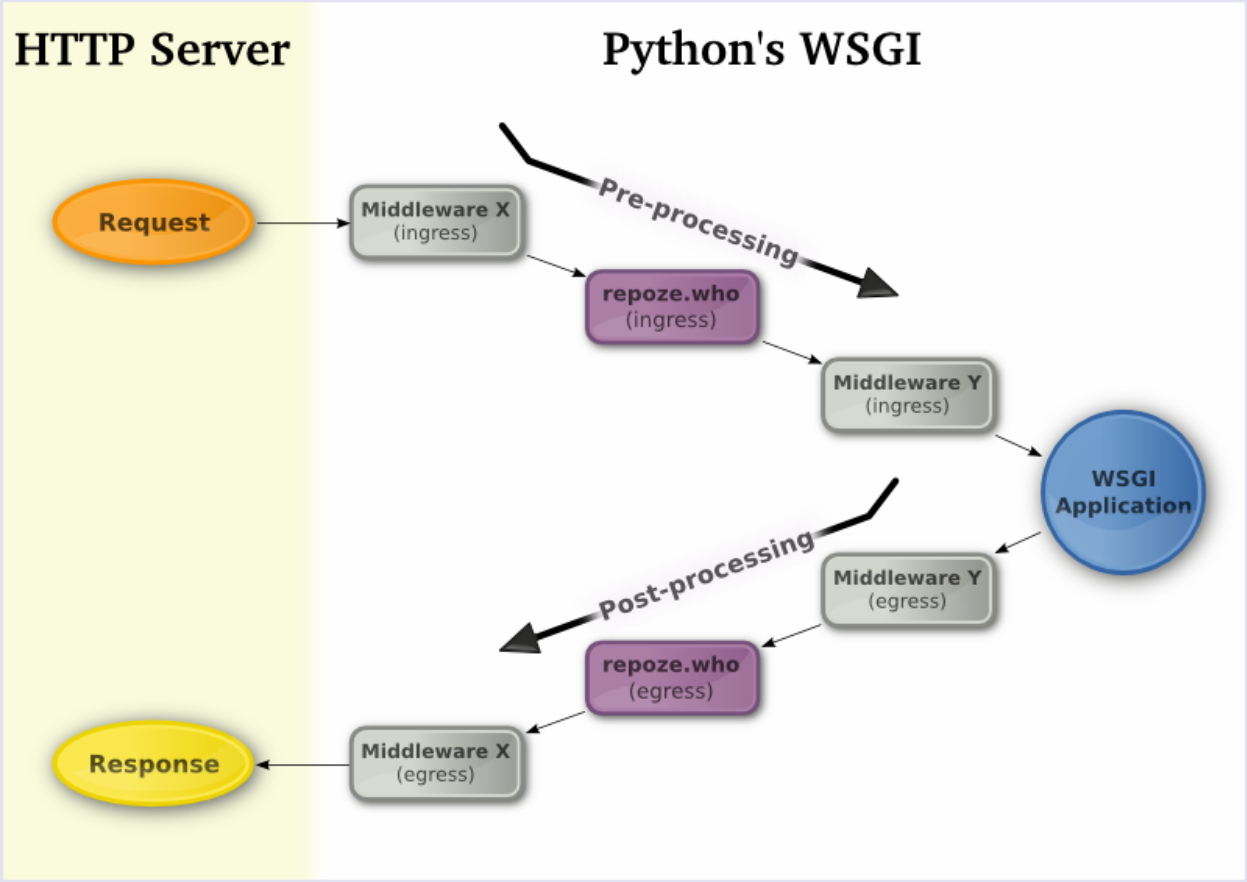
1.3.1 Middleware Responsibilities

repoze.who as middleware has one major function on ingress: it conditionally places identification and authentication information (including a `REMOTE_USER` value) into the WSGI environment and allows the request to continue to a downstream WSGI application.

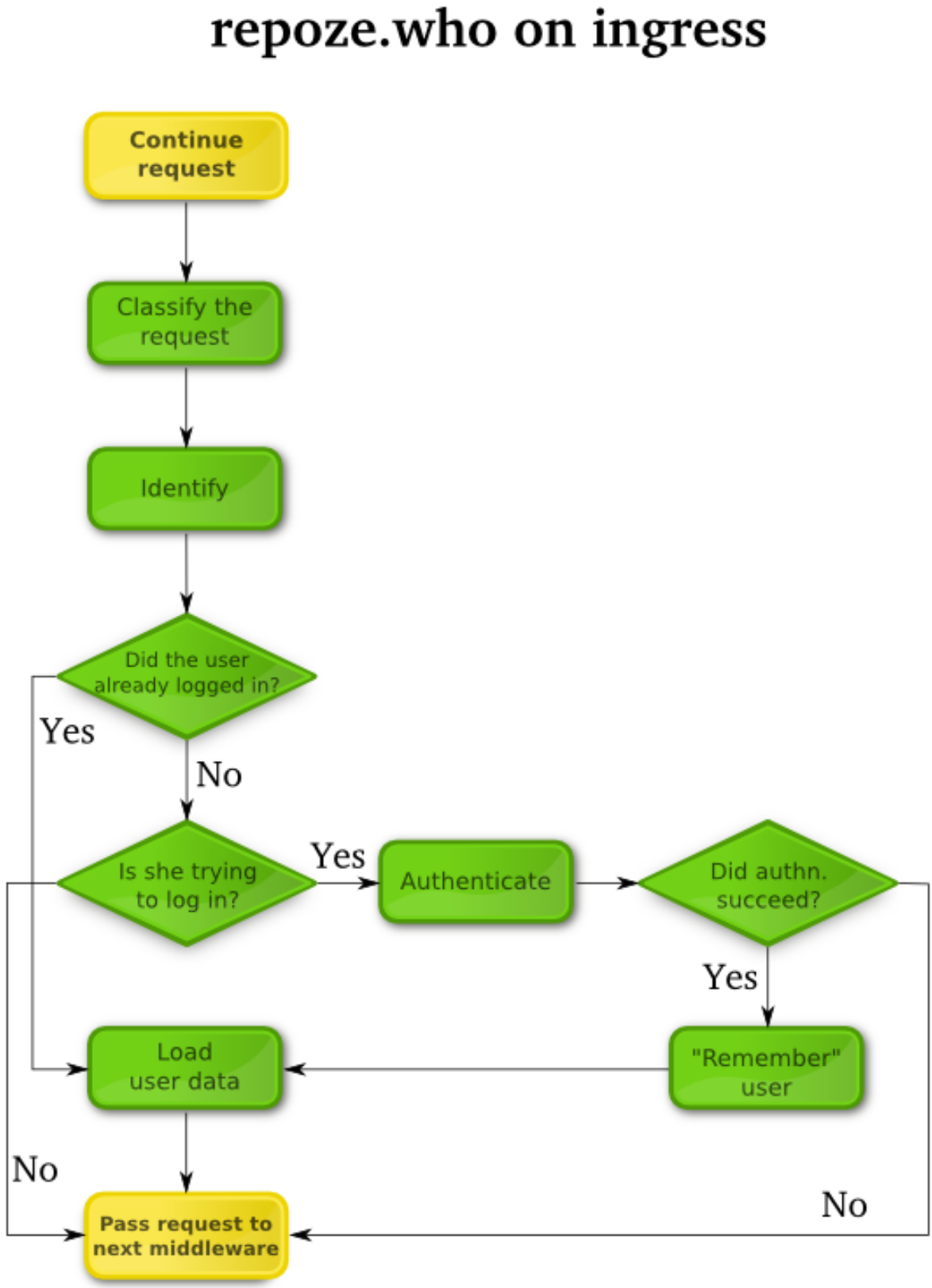
repoze.who as middleware has one major function on egress: it examines the headers set by the downstream application, the WSGI environment, or headers supplied by other plugins and conditionally challenges for credentials.

1.3.2 Lifecycle of a Request

repoze.who performs duties both on middleware "ingress" and on middleware "egress". The following graphic outlines where it sits in the context of the request and its response:



Request (Ingress) Stages



repoze.who performs the following operations in the following order during middleware ingress:

1. Environment Setup

   The middleware adds a number of keys to the WSGI environment:

   repoze.who.plugins A reference to the configured plugin set.

   repoze.who.logger A reference to the logger configured into the middleware.

   repoze.who.application A refererence to the "right-hand" application. The plugins consulted during request classification / identification / authentication may replace this application with another WSGI application, which will be used for the remainer of the current request.

2. Request Classification

   The middleware hands the WSGI environment to the configured `classifier` plugin, which is responsible for classifying the request into a single "type". This plugin must return a single string value classifying the request, e.g., "browser", "xml-rpc", "webdav", etc.

   This classification may serve to filter out plugins consulted later in the request. For instance, a plugin which issued a challenge as an HTML form would be inappropriate for use in requests from an XML-RPC or WebDAV client.

3. Identification

   Each plugin configured as an identifier for a particular class of request is called to extract identity data ("credentials") from the WSGI environment.

   For example, a basic auth identifier might use the `HTTP_AUTHORIZATION` header to find login and password information. Each configured identifier plugin is consulted in turn, and any non-None identities returned are collected into a list to be authenticated.

   Identifiers are also responsible for providing header information used to set and remove authentication information in the response during egress (to "remember" or "forget" the currently-authenticated user).

4. Authentication

   The middlware consults each plugin configured as an authenticators for a particular class of request, to compare credentials extracted by the identification plugins to a given policy, or set of valid credentials.

   For example, an htpasswd authenticator might look in a file for a user record matching any of the extracted credentials. If it finds one, and if the password listed in the record matches the password in the identity, the userid of the user would be returned (which would be the same as the login name). Successfully-authenticated ndenties are "weighted", with the highest weight identity governing the remainder of the request.

5. Metadata Assignment

   After identifying and authenticating a user, *repoze.who* consults plugins configured as metadata providers, which may augmented the authenticated identity with arbitrary metadata.

   For example, a metadata provider plugin might add the user's first, middle and last names to the identity. A more specialized metadata provider might augment the identity with a list of role or group names assigned to the user.

Response (Egress) Stages

repoze.who performs the following operations in the following order during middleware egress:

1. Challenge Decision

The middleare examines the WSGI environment and the status and headers returned by the downstream application to determine whether a challenge is required. Typically, only the status is used: if it starts with `401`, a challenge is required, and the challenge decider returns True.

This behavior can be replaced by configuring a different `challenge_decider` plugin for the middleware.

If a challenge is required, the challenge decider returns True; otherwise, it returns False.

2. Credentials reset, AKA "forgetting"

   If the challenge decider returns True, the middleware first delegates to the identifier plugin which provided the currently-authenticated identity to "forget" the identity, by adding response headers (e.g., to expire a cookie).

3. Challenge

   The plugin then consults each of the set of plugins configured as challengers for the current request classification: the first plugin which returns a non-None WSGI application will be used perform a challenge.

   Challenger plugins may use application-returned headers, the WSGI environment, and other items to determine what sort of operation should be performed to actuate the challenge.

4. Remember

   The identifier plugin that the "best" set of credentials came from (if any) will be consulted to "remember" these credentials if the challenge decider returns False.

1.4 Using the `repoze.who` Application Programming Interface (API)

1.4.1 Using `repoze.who` without Middleware

An application which does not use the *`repoze.who`* middleware needs to perform two separate tasks to use *`repoze.who`* machinery:

- At application startup, it must create an `repoze.who.api:APIFactory` instance, populating it with a request classifier, a challenge decider, and a set of plugins. It can do this process imperatively (see *Configuring repoze.who via Python Code*), or using a declarative configuration file (see *Configuring repoze.who via Config File*). For the latter case, there is a convenience function, `repoze.who.config.make_api_factory_with_config()`:

```
# myapp/run.py
from repoze.who.config import make_api_factory_with_config
who_api_factory = None
def startup(global_conf):
    global who_api_factory
    who_api_factory = make_api_factory_with_config(global_conf,
                                                   '/path/to/who.config')
```

- When it needs to use the API, it must call the `APIFactory`, passing the WSGI environment to it. The `APIFactory` returns an object implementing the `repoze.who.interfaces:IRepozeWhoAPI` interface.

```
# myapp/views.py
from myapp.run import who_api_factory
def my_view(context, request):
    who_api = who_api_factory(request.environ)
```

- Calling the `APIFactory` multiple times within the same request is allowed, and should be very cheap (the API object is cached in the request environment).

1.4.2 Mixed Use of `repoze.who` Middleware and API

An application which uses the *repoze.who* middleware may still need to interact directly with the `IRepozeWhoAPI` object for some purposes. In such cases, it should call `repoze.who.api:get_api()`, passing the WSGI environment.

```
from repoze.who.api import get_api
def my_view(context, request):
    who_api = get_api(request.environ)
```

Alternately, the application might configure the `APIFactory` at startup, as above, and then use it to find the API object, or create it if it was not already created for the current request (e.g. perhaps by the middleware):

```
def my_view(context, request):
    who_api = context.who_api_factory(request.environ)
```

1.4.3 Writing a Custom Login View

`repoze.who.api.API` provides a helper method to assist developers who want to control the details of the login view. The following BFG example illustrates how this API might be used:

```
1  def login_view(context, request):
2      message = ''
3
4      who_api = get_api(request.environ)
5      if 'form.login' in request.POST:
6          creds = {}
7          creds['login'] = request.POST['login']
8          creds['password'] = request.POST['password']
9          authenticated, headers = who_api.login(creds)
10         if authenticated:
11             return HTTPFound(location='/', headers=headers)
12
13         message = 'Invalid login.'
14     else:
15         # Forcefully forget any existing credentials.
16         _, headers = who_api.login({})
17
18     request.response_headerlist = headers
19     if 'REMOTE_USER' in request.environ:
20         del request.environ['REMOTE_USER']
21
22     return {'message': message}
```

This application is written as a "hybrid": the *repoze.who* middleware injects the API object into the WSGI enviornment on each request.

- In line 4, this application extracts the API object from the environ using `repoze.who.api:get_api()`.
- Lines 6 - 8 fabricate a set of credentials, based on the values the user entered in the form.
- In line 9, the application asks the API to authenticate those credentials, returning an identity and a set of respones headers.
- Lines 10 and 11 handle the case of successful authentication: in this case, the application redirects to the site root, setting the headers returned by the API object, which will "remember" the user across requests.
- Line 13 is reached on failed login. In this case, the headers returned in line 9 will be "forget" headers, clearing any existing cookies or other tokens.

- Lines 14 - 16 perform a "fake" login, in order to get the "forget" headers.
- Line 18 sets the "forget" headers to clear any authenticated user for subsequent requests.
- Lines 19 - 20 clear any authenticated user for the current request.
- Line 22 returns any message about a failed login to the rendering template.

1.4.4 Interfaces

interface `repoze.who.interfaces.`**`IAPIFactory`**

`__call__`(*environ*)
environ -> IRepozeWhoAPI

interface `repoze.who.interfaces.`**`IAPI`**
Facade for stateful invocation of underlying plugins.

`authenticate`()
-> {identity}

o Return an authenticated identity mapping, extracted from the request environment.

o If no identity can be authenticated, return None.

o Identity will include at least a 'repoze.who.userid' key, as well as any keys added by metadata plugins.

`forget`(*identity=None*)
-> [headers]

O Return a sequence of response headers which suffice to destroy any credentials used to establish an identity.

o If 'identity' is not passed, use the identity in the environment.

`challenge`(*status='403 Forbidden'*, *app_headers=()*)
-> wsgi application

o Return a WSGI application which represents a "challenge" (request for credentials) in response to the current request.

`logout`(*identifier_name=None*)
-> (headers)

o This is an API for browser-based application logout.

o If 'identifier_name' is passed, use it to look up the identifier; othewise, use the first configured identifier.

o Returned headers will be "forget" headers.

`login`(*credentials*, *identifier_name=None*)
-> (identity, headers)

o This is an API for browser-based application login forms.

o If 'identifier_name' is passed, use it to look up the identifier; othewise, use the first configured identifier.

o Attempt to authenticate 'credentials' as though the identifier had extracted them.

o On success, 'identity' will be authenticated mapping, and 'headers' will be "remember" headers.

o On failure, 'identity' will be None, and response_headers will be "forget" headers.

remember (*identity=None*)
-> [headers]

O Return a sequence of response headers which suffice to remember the given identity.

o If 'identity' is not passed, use the identity in the environment.

interface `repoze.who.interfaces.`**`IPlugin`**

interface `repoze.who.interfaces.`**`IRequestClassifier`**
Extends: *`repoze.who.interfaces.IPlugin`*

On ingress: classify a request.

__call__ (*environ*)
environ -> request classifier string

This interface is responsible for returning a string value representing a request classification.

o 'environ' is the WSGI environment.

interface `repoze.who.interfaces.`**`IChallengeDecider`**
Extends: *`repoze.who.interfaces.IPlugin`*

On egress: decide whether a challenge needs to be presented to the user.

__call__ (*environ*, *status*, *headers*)
args -> True | False

o 'environ' is the WSGI environment.

o 'status' is the HTTP status as returned by the downstream WSGI application.

o 'headers' are the headers returned by the downstream WSGI application.

This interface is responsible for returning True if a challenge needs to be presented to the user, False otherwise.

interface `repoze.who.interfaces.`**`IIdentifier`**
Extends: *`repoze.who.interfaces.IPlugin`*

On ingress: Extract credentials from the WSGI environment and turn them into an identity.

On egress (remember): Conditionally set information in the response headers allowing the remote system to remember this identity.

On egress (forget): Conditionally set information in the response headers allowing the remote system to forget this identity (during a challenge).

forget (*environ*, *identity*)
On egress (challenge required):

args -> [(header-name, header-value), ...] | None

Return a list of headers suitable for allowing the requesting system to forget the identification information (e.g. a Set-Cookie header with an expires date in the past). Return None if no headers need to be set. These headers will be included in the response provided by the challenge app.

identify (*environ*)
On ingress:

environ -> { k1 [v1] , ... , kN : vN } | None

o 'environ' is the WSGI environment.

o **If credentials are found, the returned identity mapping will** contain an arbitrary set of key/value pairs. If the identity is based on a login and password, the environment is recommended to contain at least 'login' and 'password' keys as this provides compatibility between the plugin and existing authenticator plugins. If the identity can be 'preauthenticated' (e.g. if the userid is embedded in the identity, such as when we're using ticket-based authentication), the plugin should set the userid in the special 'repoze.who.userid' key; no authenticators will be asked to authenticate the identity thereafer.

o **Return None to indicate that the plugin found no appropriate** credentials.

o **Only IIdentifier plugins which match one of the the current** request's classifications will be asked to perform identification.

o **An identifier plugin is permitted to add a key to the** environment named 'repoze.who.application', which should be an arbitrary WSGI application. If an identifier plugin does so, this application is used instead of the downstream application set up within the middleware. This feature is useful for identifier plugins which need to perform redirection to obtain credentials. If two identifier plugins add a 'repoze.who.application' WSGI application to the environment, the last one consulted will"win".

remember(*environ*, *identity*)

On egress (no challenge required):

args -> [(header-name, header-value), ...] | None

Return a list of headers suitable for allowing the requesting system to remember the identification information (e.g. a Set-Cookie header). Return None if no headers need to be set. These headers will be appended to any headers returned by the downstream application.

interface `repoze.who.interfaces.`**`IAuthenticator`**

Extends: *`repoze.who.interfaces.IPlugin`*

On ingress: validate the identity and return a user id or None.

authenticate(*environ*, *identity*)

identity -> 'userid' | None

o 'environ' is the WSGI environment.

o **'identity' will be a dictionary (with arbitrary keys and** values).

o **The IAuthenticator should return a single user id (optimally** a string) if the identity can be authenticated. If the identify cannot be authenticated, the IAuthenticator should return None.

Each instance of a registered IAuthenticator plugin that matches the request classifier will be called N times during a single request, where N is the number of identities found by any IIdentifierPlugin instances.

An authenticator must not raise an exception if it is provided an identity dictionary that it does not understand (e.g. if it presumes that 'login' and 'password' are keys in the dictionary, it should check for the existence of these keys before attempting to do anything; if they don't exist, it should return None).

An authenticator is permitted to add extra keys to the 'identity' dictionary (e.g., to save metadata from a database query, rather than requiring a separate query from an IMetadataProvider plugin).

interface `repoze.who.interfaces.`**`IChallenger`**

Extends: *`repoze.who.interfaces.IPlugin`*

On egress: Conditionally initiate a challenge to the user to provide credentials.

Only challenge plugins which match one of the the current response's classifications will be asked to perform a challenge.

challenge(*environ*, *status*, *app_headers*, *forget_headers*)

args -> WSGI application or None

o 'environ' is the WSGI environment.

- **o 'status' is the status written into start_response by the** downstream application.
- **o 'app_headers' is the headers list written into start_response by the** downstream application.
- **o 'forget_headers' is a list of headers which must be passed** back in the response in order to perform credentials reset (logout). These come from the 'forget' method of IIdentifier plugin used to do the request's identification.

Examine the values passed in and return a WSGI application (a callable which accepts environ and start_response as its two positional arguments, ala PEP 333) which causes a challenge to be performed. Return None to forego performing a challenge.

interface `repoze.who.interfaces.`**`IMetadataProvider`**

Extends: *repoze.who.interfaces.IPlugin*

On ingress: When an identity is authenticated, metadata providers may scribble on the identity dictionary arbitrarily. Return values from metadata providers are ignored.

`add_metadata`(*environ*, *identity*)

Add metadata to the identity (which is a dictionary). One value is always guaranteed to be in the dictionary when add_metadata is called: 'repoze.who.userid', representing the user id of the identity. Availability and composition of other keys will depend on the identifier plugin which created the identity.

1.5 Configuring `repoze.who`

1.5.1 Configuration Points

Classifiers

repoze.who "classifies" the request on middleware ingress. Request classification happens before identification and authentication. A request from a browser might be classified a different way than a request from an XML-RPC client. *repoze.who* uses request classifiers to decide which other components to consult during subsequent identification, authentication, and challenge steps. Plugins are free to advertise themselves as willing to participate in identification and authorization for a request based on this classification. The request classification system is pluggable. *repoze.who* provides a default classifier that you may use.

You may extend the classification system by making *repoze.who* aware of a different request classifier implementation.

Challenge Deciders

repoze.who uses a "challenge decider" to decide whether the response returned from a downstream application requires a challenge plugin to fire. When using the default challenge decider, only the status is used (if it starts with `401`, a challenge is required).

repoze.who also provides an alternate challenge decider, `repoze.who.classifiers.passthrough_challenge_decide` which avoids challenging `401` responses which have been "pre-challenged" by the application.

You may supply a different challenge decider as necessary.

Plugins

repoze.who has core functionality designed around the concept of plugins. Plugins are instances that are willing to perform one or more identification- and/or authentication-related duties. Each plugin can be configured arbitrarily.

repoze.who consults the set of configured plugins when it intercepts a WSGI request, and gives some subset of them a chance to influence what *repoze.who* does for the current request.

Note: As of *repoze.who* 1.0.7, the `repoze.who.plugins` package is a namespace package, intended to make it possible for people to ship eggs which are who plugins as, e.g. `repoze.who.plugins.mycoolplugin`.

1.5.2 Configuring `repoze.who` via Python Code

class `repoze.who.middleware.`**`PluggableAuthenticationMiddleware`**(*app*, *identifiers*, *challengers*, *authenticators*, *mdproviders*, *classifier*, *challenge_decider*[, *log_stream=None*[, *log_level=logging.INFO*[, *remote_user_key='REMOTE_USER'*]]])

The primary method of configuring the *repoze.who* middleware is to use straight Python code, meant to be consumed by frameworks which construct and compose middleware pipelines without using a configuration file.

In the middleware constructor: *app* is the "next" application in the WSGI pipeline. *identifiers* is a sequence of `IIdentifier` plugins, *challengers* is a sequence of `IChallenger` plugins, *mdproviders* is a sequence of `IMetadataProvider` plugins. Any of these can be specified as the empty sequence. *classifier* is a request classifier callable, *challenge_decider* is a challenge decision callable. *log_stream* is a stream object (an object with a `write` method) *or* a `logging.Logger` object, *log_level* is a numeric value that maps to the `logging` module's notion of log levels, *remote_user_key* is the key in which the `REMOTE_USER` (userid) value should be placed in the WSGI environment for consumption by downstream applications.

An example configuration which uses the default plugins follows:

```
from repoze.who.middleware import PluggableAuthenticationMiddleware
from repoze.who.interfaces import IIdentifier
from repoze.who.interfaces import IChallenger
from repoze.who.plugins.basicauth import BasicAuthPlugin
from repoze.who.plugins.auth_tkt import AuthTktCookiePlugin
from repoze.who.plugins.redirector import RedirectorPlugin
from repoze.who.plugins.htpasswd import HTPasswdPlugin

io = StringIO()
salt = 'aa'
for name, password in [ ('admin', 'admin'), ('chris', 'chris') ]:
    io.write('%s:%s\n' % (name, password))
io.seek(0)
def cleartext_check(password, hashed):
    return password == hashed
htpasswd = HTPasswdPlugin(io, cleartext_check)
basicauth = BasicAuthPlugin('repoze.who')
auth_tkt = AuthTktCookiePlugin('secret', 'auth_tkt', digest_algo="sha512")
redirector = RedirectorPlugin('/login.html')
redirector.classifications = {IChallenger:['browser'],} # only for browser
identifiers = [('auth_tkt', auth_tkt),
               ('basicauth', basicauth)]
authenticators = [('auth_tkt', auth_tkt),
                  ('htpasswd', htpasswd)]
```

```
challengers = [('redirector', redirector),
               ('basicauth', basicauth)]
mdproviders = []

from repoze.who.classifiers import default_request_classifier
from repoze.who.classifiers import default_challenge_decider
log_stream = None
import os
if os.environ.get('WHO_LOG'):
    log_stream = sys.stdout

middleware = PluggableAuthenticationMiddleware(
    app,
    identifiers,
    authenticators,
    challengers,
    mdproviders,
    default_request_classifier,
    default_challenge_decider,
    log_stream = log_stream,
    log_level = logging.DEBUG
    )
```

The above example configures the repoze.who middleware with:

- Two `IIdentifier` plugins (auth_tkt cookie, and a basic auth plugin). In this setup, when "identification" needs to be performed, the auth_tkt plugin will be checked first, then the basic auth plugin. The application is responsible for handling login via a form: this view would use the API (via **:method:'remember'**) to generate apprpriate response headers.
- Two `IAuthenticator` plugins: the auth_tkt plugin and an htpasswd plugin. The auth_tkt plugin performs both `IIdentifier` and `IAuthenticator` functions. The htpasswd plugin is configured with two valid username / password combinations: chris/chris, and admin/admin. When an username and password is found via any identifier, it will be checked against this authenticator.
- Two `IChallenger` plugins: the redirector plugin, then the basic auth plugin. The redirector auth will fire if the request is a `browser` request, otherwise the basic auth plugin will fire.

The rest of the middleware configuration is for values like logging and the classifier and decider implementations. These use the "stock" implementations.

Note: The `app` referred to in the example is the "downstream" WSGI application that who is wrapping.

1.5.3 Configuring `repoze.who` via Config File

`repoze.who` may be configured using a ConfigParser-style .INI file. The configuration file has five main types of sections: plugin sections, a general section, an identifiers section, an authenticators section, and a challengers section. Each "plugin" section defines a configuration for a particular plugin. The identifiers, authenticators, and challengers sections refer to these plugins to form a site configuration. The general section is general middleware configuration.

To configure *`repoze.who`* in Python, using an .INI file, call the *make_middleware_with_config* entry point, passing the right-hand application, the global configuration dictionary, and the path to the config file. The global configuration dictionary is a dictonary passed by PasteDeploy. The only key 'make_middleware_with_config' needs is 'here' pointing to the config file directory. For debugging people might find it useful to enable logging by adding the log_file argument, e.g. log_file="repoze_who.log"

```
from repoze.who.config import make_middleware_with_config
global_conf = {"here": "."}  # if this is not defined elsewhere
who = make_middleware_with_config(app, global_conf, 'who.ini')
```

repoze.who's configuration file can be pointed to within a PasteDeploy configuration file

```
[filter:who]
use = egg:repoze.who#config
config_file = %(here)s/who.ini
log_file = stdout
log_level = debug
```

Below is an example of a configuration file (what `config_file` might point at above) that might be used to configure the *repoze.who* middleware. A set of plugins are defined, and they are referred to by following non-plugin sections.

In the below configuration, five plugins are defined. The form, and basicauth plugins are nominated to act as challenger plugins. The form, cookie, and basicauth plugins are nominated to act as identification plugins. The htpasswd and sqlusers plugins are nominated to act as authenticator plugins.

```
[plugin:redirector]
# identificaion and challenge
use = repoze.who.plugins.redirector:make_plugin
login_url = /login.html

[plugin:auth_tkt]
# identification and authentication
use = repoze.who.plugins.auth_tkt:make_plugin
secret = s33kr1t
cookie_name = oatmeal
secure = False
include_ip = False
digest_algo = sha512

[plugin:basicauth]
# identification and challenge
use = repoze.who.plugins.basicauth:make_plugin
realm = 'sample'

[plugin:htpasswd]
# authentication
use = repoze.who.plugins.htpasswd:make_plugin
filename = %(here)s/passwd
check_fn = repoze.who.plugins.htpasswd:crypt_check

[plugin:sqlusers]
# authentication
use = repoze.who.plugins.sql:make_authenticator_plugin
# Note the double %%:  we have to escape it from the config parser in
# order to preserve it as a template for the psycopg2, whose 'paramstyle'
# is 'pyformat'.
query = SELECT userid, password FROM users where login = %%(login)s
conn_factory = repoze.who.plugins.sql:make_psycopg_conn_factory
compare_fn = repoze.who.plugins.sql:default_password_compare

[plugin:sqlproperties]
name = properties
use = repoze.who.plugins.sql:make_metadata_plugin
# Note the double %%:  we have to escape it from the config parser in
```

```
# order to preserve it as a template for the psycopg2, whose 'paramstyle'
# is 'pyformat'.
query = SELECT firstname, lastname FROM users where userid = %%(__userid)s
filter = my.package:filter_propmd
conn_factory = repoze.who.plugins.sql:make_psycopg_conn_factory

[general]
request_classifier = repoze.who.classifiers:default_request_classifier
challenge_decider = repoze.who.classifiers:default_challenge_decider
remote_user_key = REMOTE_USER

[identifiers]
# plugin_name;classifier_name:.. or just plugin_name (good for any)
plugins =
      auth_tkt
      basicauth

[authenticators]
# plugin_name;classifier_name.. or just plugin_name (good for any)
plugins =
      auth_tkt
      htpasswd
      sqlusers

[challengers]
# plugin_name;classifier_name:.. or just plugin_name (good for any)
plugins =
      redirector;browser
      basicauth

[mdproviders]
plugins =
      sqlproperties
```

The basicauth section configures a plugin that does identification and challenge for basic auth credentials. The redirector section configures a plugin that does challenges. The auth_tkt section configures a plugin that does identification for cookie auth credentials, as well as authenticating them. The htpasswd plugin obtains its user info from a file. The sqlusers plugin obtains its user info from a Postgres database.

The identifiers section provides an ordered list of plugins that are willing to provide identification capability. These will be consulted in the defined order. The tokens on each line of the `plugins=` key are in the form "plugin_name;requestclassifier_name:..." (or just "plugin_name" if the plugin can be consulted regardless of the classification of the request). The configuration above indicates that the system will look for credentials using the auth_tkt cookie identifier (unconditionally), then the basic auth plugin (unconditionally).

The authenticators section provides an ordered list of plugins that provide authenticator capability. These will be consulted in the defined order, so the system will look for users in the file, then in the sql database when attempting to validate credentials. No classification prefixes are given to restrict which of the two plugins are used, so both plugins are consulted regardless of the classification of the request. Each authenticator is called with each set of identities found by the identifier plugins. The first identity that can be authenticated is used to set `REMOTE_USER`.

The mdproviders section provides an ordered list of plugins that provide metadata provider capability. These will be consulted in the defined order. Each will have a chance (on ingress) to provide add metadata to the authenticated identity. Our example mdproviders section shows one plugin configured: "sqlproperties". The sqlproperties plugin will add information related to user properties (e.g. first name and last name) to the identity dictionary.

The challengers section provides an ordered list of plugins that provide challenger capability. These will be consulted in the defined order, so the system will consult the cookie auth plugin first, then the basic auth plugin. Each will have a

chance to initiate a challenge. The above configuration indicates that the redirector challenger will fire if it's a browser request, and the basic auth challenger will fire if it's not (fallback).

1.6 About `repoze.who` Plugins

1.6.1 Plugin Types

Identifier Plugins

You can register a plugin as willing to act as an "identifier". An identifier examines the WSGI environment and attempts to extract credentials from the environment. These credentials are used by authenticator plugins to perform authentication.

Authenticator Plugins

You may register a plugin as willing to act as an "authenticator". Authenticator plugins are responsible for resolving a set of credentials provided by an identifier plugin into a user id. Typically, authenticator plugins will perform a lookup into a database or some other persistent store, check the provided credentials against the stored data, and return a user id if the credentials can be validated.

The user id provided by an authenticator is eventually passed to downstream WSGI applications in the "REMOTE_USER' environment variable. Additionally, the "identity" of the user (as provided by the identifier from whence the identity came) is passed along to downstream application in the `repoze.who.identity` environment variable.

Metadata Provider Plugins

You may register a plugin as willing to act as a "metadata provider" (aka mdprovider). Metadata provider plugins are responsible for adding arbitrary information to the identity dictionary for consumption by downstream applications. For instance, a metadata provider plugin may add "group" information to the the identity.

Challenger Plugins

You may register a plugin as willing to act as a "challenger". Challenger plugins are responsible for initiating a challenge to the requesting user. Challenger plugins are invoked by *`repoze.who`* when it decides a challenge is necessary. A challenge might consist of displaying a form or presenting the user with a basic or digest authentication dialog.

1.6.2 Default Plugin Implementations

`repoze.who` ships with a variety of default plugins that do authentication, identification, challenge and metadata provision.

class `repoze.who.plugins.auth_tkt.`**`AuthTktCookiePlugin`**(*secret*[, *cookie_name='auth_tkt'*[, *secure=False*[, *include_ip=False*]]])

An *`AuthTktCookiePlugin`* is an `IIdentifier` and `IAuthenticator` plugin which remembers its identity state in a client-side cookie. This plugin uses the `paste.auth.auth_tkt`"auth ticket" protocol and is compatible with Apache's mod_auth_tkt. It should be instantiated passing a *secret*, which is used to encrypt

the cookie on the client side and decrypt the cookie on the server side. The cookie name used to store the cookie value can be specified using the *cookie_name* parameter. If *secure* is False, the cookie will be sent across any HTTP or HTTPS connection; if it is True, the cookie will be sent only across an HTTPS connection. If *include_ip* is True, the `REMOTE_ADDR` of the WSGI environment will be placed in the cookie.

Normally, using the plugin as an identifier requires also using it as an authenticator.

Note: Using the *include_ip* setting for public-facing applications may cause problems for some users. One study reports that as many as 3% of users change their IP addresses legitimately during a session.

Note: Plugin supports remembering user data in the cookie by saving user dict into `identity['userdata']` parameter of `remember` method. They are sent unencrypted and protected by checksum. Data will then be returned every time by `identify`. This dict must be compatible with `urllib.urlencode` function (`urllib.urlparse.urlencode` in python 3). Saving keys/values with unicode characters is supported only under python 3.

Note: Plugin supports multiple digest algorithms. It defaults to md5 to match the default for mod_auth_tkt and paste.auth.auth_tkt. However md5 is not recommended as there are viable attacks against the hash. Any algorithm from the hashlib library can be specified, currently only sha256 and sha512 are supported by mod_auth_tkt.

class `repoze.who.plugins.basicauth.`**`BasicAuthPlugin`**(*realm*)

A *`BasicAuthPlugin`* plugin is both an `IIdentifier` and `IChallenger` plugin that implements the Basic Access Authentication scheme described in **RFC 2617**. It looks for credentials within the `HTTP-Authorization` header sent by browsers. It challenges by sending an `WWW-Authenticate` header to the browser. The single argument *realm* indicates the basic auth realm that should be sent in the `WWW-Authenticate` header.

class `repoze.who.plugins.htpasswd.`**`HTPasswdPlugin`**(*filename*, *check*)

A *`HTPasswdPlugin`* is an `IAuthenticator` implementation which compares identity information against an Apache-style htpasswd file. The *filename* argument should be an absolute path to the htpasswd file' the *check* argument is a callable which takes two arguments: "password" and "hashed", where the "password" argument is the unencrypted password provided by the identifier plugin, and the hashed value is the value stored in the htpasswd file. If the hashed value of the password matches the hash, this callable should return True. A default implementation named `crypt_check` is available for use as a check function (on UNIX) as `repoze.who.plugins.htpasswd:crypt_check`; it assumes the values in the htpasswd file are encrypted with the UNIX `crypt` function.

class `repoze.who.plugins.redirector.`**`RedirectorPlugin`**(*login_url*, *came_from_param*, *reason_param*, *reason_header*)

A *`RedirectorPlugin`* is an `IChallenger` plugin. It redirects to a configured login URL at egress if a challenge is required . *login_url* is the URL that should be redirected to when a challenge is required. *came_from_param* is the name of an optional query string parameter: if configured, the plugin provides the current request URL in the redirected URL's query string, using the supplied parameter name. *reason_param* is the name of an optional query string parameter: if configured, and the application supplies a header matching *reason_header* (defaulting to `X-Authorization-Failure-Reason`), the plugin includes that reason in the query string of the redirected URL, using the supplied parameter name. *reason_header* is an optional parameter overriding the default response header name (`X-Authorization-Failure-Reason`) which the plugin checks to find the application-supplied reason for the challenge. *reason_header* cannot be set unless *reason_param* is also set.

class `repoze.who.plugins.sql.`**`SQLAuthenticatorPlugin`**(*query*, *conn_factory*, *compare_fn*)

A *`SQLAuthenticatorPlugin`* is an `IAuthenticator` implementation which compares login-password

identity information against data in an arbitrary SQL database. The *query* argument should be a SQL query that returns two columns in a single row considered to be the user id and the password respectively. The SQL query should contain Python-DBAPI style substitution values for `%(login)`, e.g. `SELECT user_id, password FROM users WHERE login = %(login)`. The *conn_factory* argument should be a callable that returns a DBAPI database connection. The *compare_fn* argument should be a callable that accepts two arguments: `cleartext` and `stored_password_hash`. It should compare the hashed version of cleartext and return True if it matches the stored password hash, otherwise it should return False. A comparison function named `default_password_compare` exists in the `repoze.who.plugins.sql` module demonstrating this. The *SQLAuthenticatorPlugin*'s `authenticate` method will return the user id of the user unchanged to *repoze.who*.

class `repoze.who.plugins.sql.`**`SQLMetadataProviderPlugin`**(*name*, *query*, *conn_factory*, *filter*)

A `SQLMetadataProviderPlugin` is an `IMetadataProvider` implementation which adds arbitrary metadata to the identity on ingress using data from an arbitrary SQL database. The *name* argument should be a string. It will be used as a key in the identity dictionary. The *query* argument should be a SQL query that returns arbitrary data from the database in a form that accepts Python-binding style DBAPI arguments. It should expect that a `__userid` value will exist in the dictionary that is bound. The SQL query should contain Python-DBAPI style substitution values for (at least) `%(__userid)`, e.g. `SELECT group FROM groups WHERE user_id = %(__userid)`. The *conn_factory* argument should be a callable that returns a DBAPI database connection. The *filter* argument should be a callable that accepts the result of the DBAPI `fetchall` based on the SQL query. It should massage the data into something that will be set in the environment under the *name* key.

1.6.3 Writing `repoze.who` Plugins

repoze.who can be extended arbitrarily through the creation of plugins. Plugins are of one of four types: identifier plugins, authenticator plugins, metadata provider plugins, and challenge plugins.

Writing An Identifier Plugin

An identifier plugin (aka an `IIdentifier` plugin) must do three things: extract credentials from the request and turn them into an "identity", "remember" credentials, and "forget" credentials.

Here's a simple cookie identification plugin that does these three things

```
class InsecureCookiePlugin(object):

    def __init__(self, cookie_name):
        self.cookie_name = cookie_name

    def identify(self, environ):
        from paste.request import get_cookies
        cookies = get_cookies(environ)
        cookie = cookies.get(self.cookie_name)

        if cookie is None:
            return None

        import binascii
        try:
            auth = cookie.value.decode('base64')
        except binascii.Error: # can't decode
            return None
```

```
        try:
            login, password = auth.split(':', 1)
            return {'login':login, 'password':password}
        except ValueError: # not enough values to unpack
            return None

    def remember(self, environ, identity):
        cookie_value = '%(login)s:%(password)s' % identity
        cookie_value = cookie_value.encode('base64').rstrip()
        from paste.request import get_cookies
        cookies = get_cookies(environ)
        existing = cookies.get(self.cookie_name)
        value = getattr(existing, 'value', None)
        if value != cookie_value:
            # return a Set-Cookie header
            set_cookie = '%s=%s; Path=/;' % (self.cookie_name, cookie_value)
            return [('Set-Cookie', set_cookie)]

    def forget(self, environ, identity):
        # return a expires Set-Cookie header
        expired = ('%s=""; Path=/; Expires=Sun, 10-May-1971 11:59:00 GMT' %
                   self.cookie_name)
        return [('Set-Cookie', expired)]

    def __repr__(self):
        return '<%s %s>' % (self.__class__.__name__, id(self))
```

.identify

The `identify` method of our InsecureCookiePlugin accepts a single argument "environ". This will be the WSGI environment dictionary. Our plugin attempts to grub through the cookies sent by the client, trying to find one that matches our cookie name. If it finds one that matches, it attempts to decode it and turn it into a login and a password, which it returns as values in a dictionary. This dictionary is thereafter known as an "identity". If it finds no credentials in cookies, it returns None (which is not considered an identity).

More generally, the `identify` method of an `IIdentifier` plugin is called once on WSGI request "ingress", and it is expected to grub arbitrarily through the WSGI environment looking for credential information. In our above plugin, the credential information is expected to be in a cookie but credential information could be in a cookie, a form field, basic/digest auth information, a header, a WSGI environment variable set by some upstream middleware or whatever else someone might use to stash authentication information. If the plugin finds credentials in the request, it's expected to return an "identity": this must be a dictionary. The dictionary is not required to have any particular keys or value composition, although it's wise if the identification plugin looks for both a login name and a password information to return at least {'login':login_name, 'password':password}, as some authenticator plugins may depend on presence of the names "login" and "password" (e.g. the htpasswd and sql `IAuthenticator` plugins). If an `IIdentifier` plugin finds no credentials, it is expected to return None.

.remember

If we've passed a REMOTE_USER to the WSGI application during ingress (as a result of providing an identity that could be authenticated), and the downstream application doesn't kick back with an unauthorized response, on egress we want the requesting client to "remember" the identity we provided if there's some way to do that and if he hasn't already, in order to ensure he will pass it back to us on subsequent requests without requiring another login. The remember method of an `IIdentifier` plugin is called for each non-unauthenticated response. It is

the responsibility of the `IIdentifier` plugin to conditionally return HTTP headers that will cause the client to remember the credentials implied by "identity".

Our InsecureCookiePlugin implements the "remember" method by returning headers which set a cookie if and only if one is not already set with the same name and value in the WSGI environment. These headers will be tacked on to the response headers provided by the downstream application during the response.

When you write a remember method, most of the work involved is determining *whether or not* you need to return headers. It's typical to see remember methods that compute an "old state" and a "new state" and compare the two against each other in order to determine if headers need to be returned. In our example InsecureCookiePlugin, the "old state" is `cookie_value` and the "new state" is `value`.

.forget

Eventually the WSGI application we're serving will issue a "401 Unauthorized" or another status signifying that the request could not be authorized. *`repoze.who`* intercepts this status and calls `IIdentifier` plugins asking them to "forget" the credentials implied by the identity. It is the "forget" method's job at this point to return HTTP headers that will effectively clear any credentials on the requesting client implied by the "identity" argument.

Our InsecureCookiePlugin implements the "forget" method by returning a header which resets the cookie that was set earlier by the remember method to one that expires in the past (on my birthday, in fact). This header will be tacked onto the response headers provided by the downstream application.

Writing an Authenticator Plugin

An authenticator plugin (aka an `IAuthenticator` plugin) must do only one thing (on "ingress"): accept an identity and check if the identity is "good". If the identity is good, it should return a "user id". This user id may or may not be the same as the "login" provided by the user. An `IAuthenticator` plugin will be called for each identity found during the identification phase (there may be multiple identities for a single request, as there may be multiple `IIdentifier` plugins active at any given time), so it may be called multiple times in the same request.

Here's a simple authenticator plugin that attempts to match an identity against ones defined in an "htpasswd" file that does just that:

```
class SimpleHTPasswdPlugin(object):

    def __init__(self, filename):
        self.filename = filename

    # IAuthenticatorPlugin
    def authenticate(self, environ, identity):
        try:
            login = identity['login']
            password = identity['password']
        except KeyError:
            return None

        f = open(self.filename, 'r')

        for line in f:
            try:
                username, hashed = line.rstrip().split(':', 1)
            except ValueError:
                continue
            if username == login:
```

```
                if crypt_check(password, hashed):
                    return username
        return None

def crypt_check(password, hashed):
    from crypt import crypt
    salt = hashed[:2]
    return hashed == crypt(password, salt)
```

An `IAuthenticator` plugin implements one "interface" method: "authentictate". The formal specification for the arguments and return values expected from these methods are available in the `interfaces.py` file in *repoze.who* as the `IAuthenticator` interface, but let's examine this method here less formally.

.authenticate

The `authenticate` method accepts two arguments: the WSGI environment and an identity. Our SimpleHTPasswdPlugin `authenticate` implementation grabs the login and password out of the identity and attempts to find the login in the htpasswd file. If it finds it, it compares the crypted version of the password provided by the user to the crypted version stored in the htpasswd file, and finally, if they match, it returns the login. If they do not match, it returns None.

Note: Our plugin's `authenticate` method does not assume that the keys `login` or `password` exist in the identity; although it requires them to do "real work" it returns None if they are not present instead of raising an exception. This is required by the `IAuthenticator` interface specification.

Writing a Challenger Plugin

A challenger plugin (aka an `IChallenger` plugin) must do only one thing on "egress": return a WSGI application which performs a "challenge". A WSGI application is a callable that accepts an "environ" and a "start_response" as its parameters; see "PEP 333" for further definition of what a WSGI application is. A challenge asks the user for credentials.

Here's an example of a simple challenger plugin:

```
from paste.httpheaders import WWW_AUTHENTICATE
from paste.httpexceptions import HTTPUnauthorized

class BasicAuthChallengerPlugin(object):

    def __init__(self, realm):
        self.realm = realm

    # IChallenger
    def challenge(self, environ, status, app_headers, forget_headers):
        head = WWW_AUTHENTICATE.tuples('Basic realm="%s"' % self.realm)
        if head[0] not in forget_headers:
            head = head + forget_headers
        return HTTPUnauthorized(headers=head)
```

Note that the plugin implements a single "interface" method: "challenge". The formal specification for the arguments and return values expected from this method is available in the "interfaces.py" file in *repoze.who* as the `IChallenger` interface. This method is called when *repoze.who* determines that the application has returned an "unauthorized" response (e.g. a 401). Only one challenger will be consulted during "egress" as necessary (the first one to return a non-None response).

.challenge

The challenge method takes environ (the WSGI environment), 'status' (the status as set by the downstream application), the "app_headers" (headers returned by the application), and the "forget_headers" (headers returned by all participating `IIdentifier` plugins whom were asked to "forget" this user).

Our BasicAuthChallengerPlugin takes advantage of the fact that the HTTPUnauthorized exception imported from paste.httpexceptions can be used as a WSGI application. It first makes sure that we don't repeat headers if an identification plugin has already set a "WWW-Authenticate" header like ours, then it returns an instance of HTTPUnauthorized, passing in merged headers. This will cause a basic authentication dialog to be presented to the user.

Writing a Metadata Provider Plugin

A metadata provider plugin (aka an `IMetadataProvider` plugin) must do only one thing (on "ingress"): "scribble" on the identity dictionary provided to it when it is called. An `IMetadataProvider` plugin will be called with the final "best" identity found during the authentication phase, or not at all if no "best" identity could be authenticated. Thus, each `IMetadataProvider` plugin will be called exactly zero or one times during a request.

Here's a simple metadata provider plugin that provides "property" information from a dictionary:

```
_DATA = {
    'chris': {'first_name':'Chris', 'last_name':'McDonough'} ,
    'whit': {'first_name':'Whit', 'last_name':'Morriss'}
    }

class SimpleMetadataProvider(object):

    def add_metadata(self, environ, identity):
        userid = identity.get('repoze.who.userid')
        info = _DATA.get(userid)
        if info is not None:
            identity.update(info)
```

.add_metadata

Arbitrarily add information to the identity dict based in other data in the environment or identity. Our plugin adds `first_name` and `last_name` values to the identity if the userid matches `chris` or `whit`.

1.7 Known Plugins for `repoze.who`

1.7.1 Plugins shipped with `repoze.who`

See *Default Plugin Implementations*.

1.7.2 Deprecated plugins

The `repoze.who.deprecatedplugins` distribution bundles the following plugin implementations which were shipped with *`repoze.who`* prior to version 2.0a3. These plugins are deprecated, and should only be used while migrating an existing deployment to replacement versions.

repoze.who.plugins.cookie.InsecureCookiePlugin An `IIdentifier` plugin which stores identification information in an insecure form (the base64 value of the username and password separated by a colon) in a client-side cookie. Please use the `AuthTktCookiePlugin` instead.

`repoze.who.plugins.form.FormPlugin`

An `IIdentifier` and `IChallenger` plugin, which intercepts form POSTs to gather identification at ingress and conditionally displays a login form at egress if challenge is required.

Applications should supply their own login form, and use `repoze.who.api.API` to authenticate and remember users. To replace the challenger role, please use *`repoze.who.plugins.redirector.RedirectorPlugin`*, configured with the URL of your application's login form.

`repoze.who.plugins.form.RedirectingFormPlugin`

An `IIdentifier` and `IChallenger` plugin, which intercepts form POSTs to gather identification at ingress and conditionally redirects a login form at egress if challenge is required.

Applications should supply their own login form, and use `repoze.who.api.API` to authenticate and remember users. To replace the challenger role, please use *`repoze.who.plugins.redirector.RedirectorPlugin`*, configured with the URL of your application's login form.

1.7.3 Third-party Plugins

repoze.who.plugins.zodb.ZODBPlugin This class implements the *`repoze.who.interfaces.IAuthenticator`* and *`repoze.who.interfaces.IMetadataProvider`* plugin interfaces using ZODB database lookups. See http://pypi.python.org/pypi/repoze.whoplugins.zodb/

repoze.who.plugins.ldap.LDAPAuthenticatorPlugin This class implements the *`repoze.who.interfaces.IAuthenticator`* plugin interface using the `python-ldap` library to query an LDAP database. See http://code.gustavonarea.net/repoze.who.plugins.ldap/

repoze.who.plugins.ldap.LDAPAttributesPlugin This class implements the *`repoze.who.interfaces.IMetadataProvider`* plugin interface using the `python-ldap` library to query an LDAP database. See http://code.gustavonarea.net/repoze.who.plugins.ldap/

repoze.who.plugins.friendlyform.FriendlyFormPlugin This class implements the *`repoze.who.interfaces.IIdentifier`* and *`repoze.who.interfaces.IChallenger`* plugin interfaces. It is similar to `repoze.who.plugins.form.RedirectingFormPlugin`, bt with with additional features:

- Users are not challenged on logout, unless the referrer URL is a private one (but that's up to the application).
- Developers may define post-login and/or post-logout pages.
- In the login URL, the amount of failed logins is available in the environ. It's also increased by one on every login try. This counter will allow developers not using a post-login page to handle logins that fail/succeed.

See http://code.gustavonarea.net/repoze.who-friendlyform/

repoze.who.plugins.openid.identifiers.OpenIdIdentificationPlugin() This class implements the *`repoze.who.interfaces.IIdentifier`*, *`repoze.who.interfaces.IAuthenticator`*, and *`repoze.who.interfaces.IChallenger`* plugin interfaces using OpenId. See http://quantumcore.org/docs/repoze.who.plugins.openid/

repoze.who.plugins.openid.classifiers.openid_challenge_decider() This function provides the *`repoze.who.interfaces.IChallengeDecider`* interface using OpenId. See http://quantumcore.org/docs/repoze.who.plugins.openid/

repoze.who.plugins.use_beaker.UseBeakerPlugin This packkage provids a *repoze.who.interfaces.IIdentifier* plugin using `beaker.session` cache. See http://pypi.python.org/pypi/repoze.who-use_beaker/

repoze.who.plugins.cas.main_plugin.CASChallengePlugin This class implements the *repoze.who.interfaces.IIdentifier* *repoze.who.interfaces.IAuthenticator*, and *repoze.who.interfaces.IChallenger* plugin interfaces using CAS. See http://pypi.python.org/pypi/repoze.who.plugins.cas

repoze.who.plugins.cas.challenge_decider.my_challenge_decider This function provides the *repoze.who.interfaces.IChallengeDecider* interface using CAS. See http://pypi.python.org/pypi/repoze.who.plugins.cas/

repoze.who.plugins.recaptcha.captcha.RecaptchaPlugin This class implements the *repoze.who.interfaces.IAuthenticator* plugin interface, using the recaptch API. See http://pypi.python.org/pypi/repoze.who.plugins.recaptcha/

repoze.who.plugins.sa.SQLAlchemyUserChecker User existence checker for *repoze.who.plugins.auth_tkt.AuthTktCookiePlugin*, based on the SQLAlchemy ORM. See http://pypi.python.org/pypi/repoze.who.plugins.sa/

repoze.who.plugins.sa.SQLAlchemyAuthenticatorPlugin This class implements the *repoze.who.interfaces.IAuthenticator* plugin interface, using the the SQLAlchemy ORM. See http://pypi.python.org/pypi/repoze.who.plugins.sa/

repoze.who.plugins.sa.SQLAlchemyUserMDPlugin This class implements the *repoze.who.interfaces.IMetadataProvider* plugin interface, using the the SQLAlchemy ORM. See http://pypi.python.org/pypi/repoze.who.plugins.sa/

repoze.who.plugins.formcookie.CookieRedirectingFormPlugin This class implements the *repoze.who.interfaces.IIdentifier* and *repoze.who.interfaces.IChallenger* plugin interfaces, similar to `repoze.who.plugins.form.RedirectingFormPlugin`. The plugin tracks the `came_from` URL via a cookie, rather than the query string. See http://pypi.python.org/pypi/repoze.who.plugins.formcookie/

CHAPTER 2

Change History

2.1 repoze.who Changelog

2.1.1 2.3.1 (unreleased)

- TBD

2.1.2 2.3 (2016-05-31)

- Add support for Python 3.4, Python 3.5, and PyPy3.
- Drop support for Python 2.6 and 3.2.
- `middleware`: avoid passing extracted `identity` to `remember` during egress (the app may have called `api.forget()`). See #21.
- `_auth_tkt` / `plugins.auth_tkt`: add support for any hash algorithm supported by the `hashlib` module in Python's standard library. Fixes #22 via #23.
- `plugins.auth_tkt`: Fix storage of "userdata" to save dict. Fixes #14 via #18.
- middleware: avoid UnboundLocalError when wrapped generater yields no items. See: http://bugs.repoze.org/issue184
- Make cookie expiration date RFC-2616 compliant (independent of locale, including 'GMT' zone). See #11.

2.1.3 2.2 (2013-05-17)

- Parse INI-file configuration using `SafeConfigParser`: allows escaping the `'%'` so that e.g. a query template using for a DB-API connection using `pyformat` preserves the template.
- Added support for Python 3.3, PyPy.

2.1.4 2.1 (2013-03-20)

- `_compat` module: tolerate missing `CONTENT_TYPE` key in the WSGI environment. Thanks to Dag Hoidal for the patch.
- `htpasswd` plugin: add a `sha1_check` checker function (the `crypt` module is not available on Windows). Thanks to Chandrashekar Jayaraman for the patch.

- Documentation typo fixes from Carlos de la Guardia and Atsushi Odagiri.

2.1.5 2.1b1 (2012-11-05)

- Ported to Py3k using the "compatible subset" mode. - Dropped support for Python < 2.6.x. - Dropped dependency on Paste (forking some code from it). - Added dependency on WebOb instead. Thanks to Atsushi Odagiri (aodag) for the initial effort.

2.1.6 2.0 (2011-09-28)

- `auth_tkt` plugin: strip any port number from the 'Domain' of generated cookies. http://bugs.repoze.org/issue66
- Further harden middleware, calling `close()` on the iterable even if raising an exception for a missing challenger. http://bugs.repoze.org/issue174

2.1.7 2.0b1 (2011-05-24)

- Enabled standard use of logging module's configuration mechanism. See http://docs.python.org/dev/howto/logging.html#configuring-logging-for-a-library Thanks to jgoldsmith for the patch: http://bugs.repoze.org/issue178
- `repoze.who.plugins.htpasswd`: defend against timing-based attacks.

2.1.8 2.0a4 (2011-02-02)

- Ensure that the middleware calls `close()` (if it exists) on the iterable returned from thw wrapped application, as required by PEP 333. http://bugs.repoze.org/issue174
- Make `make_api_factory_with_config` tolerant of invalid filenames / content for the config file: in such cases, the API factory will have *no* configured plugins or policies: it will only be useful for retrieving the API from an environment populated by middleware.
- Fix bug in `repoze.who.api` where the `remember()` or `forget()` methods could return a None if the identifier plugin returned a None.
- Fix `auth_tkt` plugin to not hand over tokens as strings to paste. See http://lists.repoze.org/pipermail/repoze-dev/2010-November/003680.html
- Fix `auth_tkt` plugin to add "secure" and "HttpOnly" to cookies when configured with `secure=True`: these attributes prevent the browser from sending cookies over insecure channels, which could be vulnerable to some XSS attacks.
- Avoid propagating unicode 'max_age' value into cookie headers. See https://bugs.launchpad.net/bugs/674123 .
- Added a single-file example BFG application demonstrating the use of the new 'login' and 'logout' methods of the API object.
- Add `login` and `logout` methods to the `repoze.who.api.API` object, as a convenience for application-driven login / logout code, which would otherwise need to use private methods of the API, and reach down into its plugins.

2.1.9 2.0a3 (2010-09-30)

- Deprecated the following plugins, moving their modules, tests, and docs to a new project, `repoze.who.deprecatedplugins`:
  - `repoze.who.plugins.cookie.InsecureCookiePlugin`
  - `repoze.who.plugins.form.FormPlugin`
  - `repoze.who.plugins.form.RedirectingFormPlugin`
- Made the `repoze.who.plugins.cookie.InsecureCookiePlugin` take a `charset` argument, and use to to encode / decode login and password. See http://bugs.repoze.org/issue155
- Updated `repoze.who.restrict` to return headers as a list, to keep `wsgiref` from complaining.
- Helped default request classifier cope with xml submissions with an explicit charset defined: http://bugs.repoze.org/issue145 (Lorenzo M. Catucci)
- Corrected the handling of type and subtype when matching an XML post to `xmlpost` in the default classifier, which, according to RFC 2045, must be matched case-insensitively: http://bugs.repoze.org/issue145 (Lorenzo M. Catucci)
- Added `repoze.who.config:make_api_factory_with_config`, a convenience method for applications which want to set up their own API Factory from a configuration file.
- Fixed example call to `repoze.who.config:make_middleware_with_config` (added missing `global_config` argument). See http://bugs.repoze.org/issue114

2.1.10 2.0a2 (2010-03-25)

Bugs Fixed

- Fixed failure to pass substution values in log message string formatting for `repoze.who.api:API.challenge`. Fix included adding tests for all logging done by the API object. See http://bugs.repoze.org/issue122

Backward Incompatibilities

- Adjusted logging level for some lower-level details from `info` to `debug`.

2.1.11 2.0a1 (2010-02-24)

Features

- Restored the ability to create the middleware using the old `classifier` argument. That argument is now a deprecated-but-will-work-forever alias for `request_classifier`.
- The `auth_tkt` plugin now implements the `IAuthenticator` interface, and should normally be used both as an `IIdentifier` and an `IAuthenticator`.
- Factored out the API of the middleware object to make it useful from within the application. Applications using `repoze.who` ` now fall into one of three catgeories:
  - "middleware-only" applications are configured with middleware, and use either `REMOTE_USER` or `repoze.who.identity` from the environment to determing the authenticated user.

  - "bare metal" applications use no `repoze.who` middleware at all: instead, they configure and an `APIFactory` object at startup, and use it to create an `API` object when needed on a per-request basis.
  - "hybrid" applications are configured with `repoze.who` middleware, but use a new library function to fetch the `API` object from the environ, e.g. to permit calling `remember` after a signup or successful login.

Bugs Fixed

- Fix http://bugs.repoze.org/issue102: when no challengers existed, logging would cause an exception.
- Remove `ez_setup.py` and dependency on it in setup.py (support distribute).

Backward Incompatibilities

- The middleware used to allow identifier plugins to "pre-authenticate" an identity. This feature is no longer supported: the `auth_tkt` plugin, which used to use the feature, is now configured to work as an authenticator plugin (as well as an identifier).
- The `repoze.who.middleware:PluggableAuthenticationMiddleware` class no longer has the following (non-API) methods (now made API methods of the `repoze.who.api:API` class):
  - `add_metadata`
  - `authenticate`
  - `challenge`
  - `identify`
- The following (non-API) functions moved from `repoze.who.middleware` to `repoze.who.api`:
  - `make_registries`
  - `match_classification`
  - `verify`

2.1.12 1.0.18 (2009-11-05)

- Issue #104: AuthTkt plugin was passing an invalid cookie value in headers from `forget`, and was not setting the `Max-Age` and `Expires` attributes of those cookies.

2.1.13 1.0.17 (2009-11-05)

- Fixed the `repoze.who.plugins.form.make_plugin` factory's `formcallable` argument handling, to allow passing in a dotted name (e.g., from a config file).

2.1.14 1.0.16 (2009-11-04)

- Exposed `formcallable` argument for `repoze.who.plugins.form.FormPlugin` to the callers of the `repoze.who.plugins.form.make_plugin` factory. Thanks to Roland Hedburg for the report.
- Fixed an issue that caused the following symptom when using the ini configuration parser:

```
TypeError: _makePlugin() got multiple values for keyword argument 'name'
```

  See http://bugs.repoze.org/issue92 for more details. Thanks to vaab for the bug report and initial fix.

2.1.15 1.0.15 (2009-06-25)

- If the form post value `max_age` exists while in the `identify` method is handling the `login_handler_path`, pass the max_age value in the returned identity dictionary as `max_age`. See the below bullet point for why.
- If the `identity` dict passed to the `auth_tkt remember` method contains a `max_age` key with a string (or integer) value, treat it as a cue to set the `Max-Age` and `Expires` headers in the returned cookies. The cookie `Max-Age` is set to the value and the `Expires` is computed from the current time.

2.1.16 1.0.14 (2009-06-17)

- Fix test breakage on Windows. See http://bugs.repoze.org/issue79 .
- Documented issue with using `include_ip` setting in the `auth_tkt` plugin. See http://bugs.repoze.org/issue81 .
- Added 'passthrough_challenge_decider', which avoids re-challenging 401 responses which have been "pre-challenged" by the application.
- One-hundred percent unit test coverage.
- Add `timeout` and `reissue_time` arguments to the auth_tkt identifier plugin, courtesty of Paul Johnston.
- Add a `userid_checker` argument to the auth_tkt identifier plugin, courtesty of Gustavo Narea.

  If `userid_checker` is provided, it must be a dotted Python name that resolves to a function which accepts a userid and returns a boolean True or False, indicating whether that user exists in a database. This is a workaround. Due to a design bug in repoze.who, the only way who can check for user existence is to use one or more IAuthenticator plugin `authenticate` methods. If an IAuthenticator's `authenticate` method returns true, it means that the user exists. However most IAuthenticator plugins expect *both* a username and a password, and will return False unconditionally if both aren't supplied. This means that an authenticator can't be used to check if the user "only" exists. The identity provided by an auth_tkt does not contain a password to check against. The actual design bug in repoze.who is this: when a user presents credentials from an auth_tkt, he is considered "preauthenticated". IAuthenticator.authenticate is just never called for a "preauthenticated" identity, which works fine, but it means that the user will be considered authenticated even if you deleted the user's record from whatever database you happen to be using. However, if you use a userid_checker, you can ensure that a user exists for the auth_tkt supplied userid. If the userid_checker returns False, the auth_tkt credentials are considered "no good".

2.1.17 1.0.13 (2009-04-24)

- Added a paragraph to `IAuthenticator` docstring, documenting that plugins are allowed to add keys to the `identity` dictionary (e.g., to save a second database query in an `IMetadataProvider` plugin).
- Patch supplied for issue #71 (http://bugs.repoze.org/issue71) whereby a downstream app can return a generator, relying on an upstream component to call start_response. We do this because the challenge decider needs the status and headers to decide what to do.

2.1.18 1.0.12 (2009-04-19)

- auth_tkt plugin tried to append REMOTE_USER_TOKENS data to existing tokens data returned by auth_tkt.parse_tkt; this was incorrect; just overwrite.
- Extended auth_tkt plugin factory to allow passing secret in a separate file from the main config file. See http://bugs.repoze.org/issue40 .

2.1.19 1.0.11 (2009-04-10)

- Fix auth_tkt plugin; cookie values are now quoted, making it possible to put spaces and other whitespace, etc in usernames. (thanks to Michael Pedersen).
- Fix corner case issue of an exception raised when attempting to log when there are no identifiers or authenticators.

2.1.20 1.0.10 (2009-01-23)

- The RedirectingFormPlugin now passes along SetCookie headers set into the response by the application within the NotFound response (fixes TG2 "flash" issue).

2.1.21 1.0.9 (2008-12-18)

- The RedirectingFormPlugin now attempts to find a header named `X-Authentication-Failure-Reason` among the response headers set by the application when a challenge is issued. If a value for this header exists (and is non-blank), the value is attached to the redirect URL's query string as the `reason` parameter (or a user-settable key). This makes it possible for downstream applications to issue a response that initiates a challenge with this header and subsequently display the reason in the login form rendered as a result of the challenge.

2.1.22 1.0.8 (2008-12-13)

- The `PluggableAuthenticationMiddleware` constructor accepts a `log_stream` argument, which is typically a file. After this release, it can also be a PEP 333 `Logger` instance; if it is a PEP 333 `Logger` instance, this logger will be used as the repoze.who logger (instead of one being constructed by the middleware, as was previously always the case). When the `log_stream` argument is a PEP 333 Logger object, the `log_level` argument is ignored.

2.1.23 1.0.7 (2008-08-28)

- `repoze.who` and `repoze.who.plugins` were not added to the `namespace_packages` list in setup.py, potentially making 1.0.6 a brownbag release, given that making these packages namespace packages was the only reason for its release.

2.1.24 1.0.6 (2008-08-28)

- Make repoze.who and repoze.who.plugins into namespace packages mainly so we can allow plugin authors to distribute packages in the repoze.who.plugins namespace.

2.1.25 1.0.5 (2008-08-23)

- Fix auth_tkt plugin to set the same cookies in its `remember` method that it does in its `forget` method. Previously, logging out and relogging back in to a site that used auth_tkt identifier plugin was slightly dicey and would only work sometimes.
- The FormPlugin plugin has grown a redirect-on-unauthorized feature. Any response from a downstream application that causes a challenge and includes a Location header will cause a redirect to the value of the Location header.

2.1.26 1.0.4 (2008-08-22)

- Added a key to the '[general]' config section: `remote_user_key`. If you use this key in the config file, it tells who to 1) not perform any authentication if it exists in the environment during ingress and 2) to set the key in the environment for the downstream app to use as the REMOTE_USER variable. The default is `REMOTE_USER`.
- Using unicode user ids in combination with the auth_tkt plugin would cause problems under mod_wsgi.
- Allowed 'cookie_path' argument to InsecureCookiePlugin (and config constructor). Thanks to Gustavo Narea.

2.1.27 1.0.3 (2008-08-16)

- A bug in the middleware's `authenticate` method made it impossible to authenticate a user with a userid that was null (e.g. 0, False), which are valid identifiers. The only invalid userid is now None.
- Applied patch from Olaf Conradi which logs an error when an invalid filename is passed to the HTPasswdPlugin.

2.1.28 1.0.2 (2008-06-16)

- Fix bug found by Chris Perkins: the auth_tkt plugin's "remember" method didn't handle userids which are Python "long" instances properly. Symptom: TypeError: cannot concatenate 'str' and 'long' objects in "paste.auth.auth_tkt".
- Added predicate-based "restriction" middleware support (repoze.who.restrict), allowing configuratio-driven authorization as a WSGI filter. One example predicate, 'authenticated_predicate', is supplied, which requires that the user be authenticated either via 'REMOTE_USER' or via 'repoze.who.identity'. To use the filter to restrict access:

```
    [filter:authenticated_only]
    use = egg:repoze.who#authenticated

  or::

    [filter:some_predicate]
    use = egg:repoze.who#predicate
    predicate = my.module:some_predicate
    some_option = a value
```

2.1.29 1.0.1 (2008-05-24)

- Remove dependency-link to dist.repoze.org to prevent easy_install from inserting that path into its search paths (the dependencies are available from PyPI).

2.1.30 1.0 (2008-05-04)

- The plugin at plugins.form.FormPlugin didn't redirect properly after collecting identification information. Symptom: a downstream app would receive a POST request with a blank body, which would sometimes result in a Bad Request error.
- Fixed interface declarations of 'classifiers.default_request_classifier' and 'classifiers.default_password_compare'.
- Added actual config-driven middleware factory, 'config.make_middleware_with_config'
- Removed fossilized 'who_conf' argument from plugin factory functions.

- Added ConfigParser-based WhoConfig, implementing the spec outlined at http://www.plope.com/static/misc/sphinxtest/intro.html#middleware-configuration-via-config-file, with the following changes:
  - **"Bare" plugins (requiring no configuration options) may be specified** as either egg entry points (e.g., 'egg:distname#entry_point_name') or as dotted-path-with-colon (e.g., 'dotted.name:object_id').
  - Therefore, the separator between a plugin and its classifier is now a semicolon, rather than a colon. E.g.:

```
[plugins:id_plugin]
use = egg:another.package#identify_with_frobnatz
frobnatz = baz

[identifiers]
plugins =
  egg:my.egg#identify;browser
  dotted.name:identifier
  id_plugin
```

2.1.31 0.9.1 (2008-04-27)

- Fix auth_tkt plugin to be able to encode and decode integer user ids.

2.1.32 0.9 (2008-04-01)

- Fix bug introduced in FormPlugin in 0.8 release (rememberer headers not set).
- Add PATH_INFO to started and ended log info.
- Add a SQLMetadataProviderPlugin (in plugins/sql).
- Change constructor of SQLAuthenticatorPlugin: it now accepts only "query", "conn_factory", and "compare_fn". The old constructor accepted a DSN, but some database systems don't use DBAPI DSNs. The new constructor accepts no DSN; the conn_factory is assumed to do all the work to make a connection, including knowing the DSN if one is required. The "conn_factory" should return something that, when called with no arguments, returns a database connection.
- The "make_plugin" helper in plugins/sql has been renamed "make_authenticator_plugin". When called, this helper will return a SQLAuthenticatorPlugin. A bit of helper logic in the "make_authenticator_plugin" allows a connection factory to be computed. The top-level callable referred to by conn_factory in this helper should return a function that, when called with no arguments, returns a datbase connection. The top-level callable itself is called with "who_conf" (global who configuration) and any number of non-top-level keyword arguments as they are passed into the helper, to allow for a DSN or URL or whatever to be passed in.
- A "make_metatata_plugin" helper has been added to plugins/sql. When called, this will make a SQLMetadataProviderPlugin. See the implementation for details. It is similar to the "make_authenticator_plugin" helper.

2.1.33 0.8 (2008-03-27)

- Add a RedirectingFormIdentifier plugin. This plugin is willing to redirect to an external (or downstream application) login form to perform identification. The external login form must post to the "login_handler_path" of the plugin (optimally with a "came_from" value to tell the plugin where to redirect the response to if the authentication works properly). The "logout_handler_path" of this plugin can be visited to perform a logout. The "came_from" value also works there.

- Identifier plugins are now permitted to set a key in the environment named 'repoze.who.application' on ingress (in 'identify'). If an identifier plugin does so, this application is used instead of the "normal" downstream application. This feature was added to more simply support the redirecting form identifier plugin.

2.1.34 0.7 (2008-03-26)

- Change the IMetadataProvider interface: this interface used to have a "metadata" method which returned a dictionary. This method is not part of that API anymore. It's been replaced with an "add_metadata" method which has the signature:

```
 def add_metadata(environ, identity):
     """
     Add metadata to the identity (which is a dictionary)
     """

The return value is ignored.  IMetadataProvider plugins are now
assumed to be responsible for 'scribbling' directly on the identity
that is passed in (it's a dictionary).  The user id can always be
retrieved from the identity via identity['repoze.who.userid'] for
metadata plugins that rely on that value.
```

2.1.35 0.6 (2008-03-20)

- Renaming: repoze.pam is now repoze.who
- Bump ez_setup.py version.
- Add IMetadataProvider plugin type. Chris says 'Whit rules'.

2.1.36 0.5 (2008-03-09)

- Allow "remote user key" (default: REMOTE_USER) to be overridden (pass in remote_user_key to middleware constructor).
- Allow form plugin to override the default form.
- API change: IIdentifiers are no longer required to put both 'login' and 'password' in a returned identity dictionary. Instead, an IIdentifier can place arbitrary key/value pairs in the identity dictionary (or return an empty dictionary).
- API return value change: the "failure" identity which IIdentifiers return is now None rather than an empty dictionary.
- The IAuthenticator interface now specifies that IAuthenticators must not raise an exception when evaluating an identity that does not have "expected" key/value pairs (e.g. when an IAuthenticator that expects login and password inspects an identity returned by an IP-based auth system which only puts the IP address in the identity); instead they fail gracefully by returning None.
- Add (cookie) "auth_tkt" identification plugin.
- Stamp identity dictionaries with a userid by placing a key named 'repoze.pam.userid' into the identity for each authenticated identity.
- If an IIdentifier plugin inserts a 'repoze.pam.userid' key into the identity dictionary, consider this identity "preauthenticated". No authenticator plugins will be asked to authenticate this identity. This is designed for things like the recently added auth_tkt plugin, which embeds the user id into the ticket. This effectively alllows an IIdentifier plugin to become an IAuthenticator plugin when breaking apart the responsibility into two separate plugins

is "make-work". Preauthenticated identities will be selected first when deciding which identity to use for any given request.

- Insert a 'repoze.pam.identity' key into the WSGI environment on ingress if an identity is found. Its value will be the identity dictionary related to the identity selected by repoze.pam on ingress. Downstream consumers are allowed to mutate this dictionary; this value is passed to "remember" and "forget", so its main use is to do a "credentials reset"; e.g. a user has changed his username or password within the application, but we don't want to force him to log in again after he does so.

2.1.37 0.4 (03-07-2008)

- Allow plugins to specify a classifiers list per interface (instead of a single classifiers list per plugin).

2.1.38 0.3 (03-05-2008)

- Make SQLAuthenticatorPlugin's default_password_compare use hexdigest sha instead of base64'ed binary sha for simpler conversion.

2.1.39 0.2 (03-04-2008)

- Added SQLAuthenticatorPlugin (see plugins/sql.py).

2.1.40 0.1 (02-27-2008)

- Initial release (no configuration file support yet).

CHAPTER 3

Support and Development

To report bugs, use the Repoze bug tracker.

If you've got questions that aren't answered by this documentation, contact the Repoze-dev maillist or join the #repoze IRC channel.

Browse and check out tagged and trunk versions of *repoze.who* via the Repoze github repository. To check out the trunk via git, use this command:

```
git clone https://github.com:repoze/repoze.who.git
```

Or, if you are logged in:

> git clone git@github.com:repoze/repoze.who.git

To find out how to become a contributor to *repoze.who*, please see the contributor's page.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

r

repoze.who, 1
repoze.who.interfaces, 12
repoze.who.middleware, 16
repoze.who.plugins.auth_tkt, 20
repoze.who.plugins.basicauth, 21
repoze.who.plugins.htpasswd, 21
repoze.who.plugins.redirector, 21
repoze.who.plugins.sql, 21

Symbols

__call__() (repoze.who.interfaces.IAPIFactory method), 12
__call__() (repoze.who.interfaces.IChallengeDecider method), 13
__call__() (repoze.who.interfaces.IRequestClassifier method), 13

A

add_metadata() (repoze.who.interfaces.IMetadataProvider method), 15
authenticate() (repoze.who.interfaces.IAPI method), 12
authenticate() (repoze.who.interfaces.IAuthenticator method), 14
AuthTktCookiePlugin (class in repoze.who.plugins.auth_tkt), 20

B

BasicAuthPlugin (class in repoze.who.plugins.basicauth), 21

C

challenge() (repoze.who.interfaces.IAPI method), 12
challenge() (repoze.who.interfaces.IChallenger method), 14

F

forget() (repoze.who.interfaces.IAPI method), 12
forget() (repoze.who.interfaces.IIdentifier method), 13

H

HTPasswdPlugin (class in repoze.who.plugins.htpasswd), 21

I

IAPI (interface in repoze.who.interfaces), 12
IAPIFactory (interface in repoze.who.interfaces), 12
IAuthenticator (interface in repoze.who.interfaces), 14
IChallengeDecider (interface in repoze.who.interfaces), 13
IChallenger (interface in repoze.who.interfaces), 14
identify() (repoze.who.interfaces.IIdentifier method), 13
IIdentifier (interface in repoze.who.interfaces), 13
IMetadataProvider (interface in repoze.who.interfaces), 15
IPlugin (interface in repoze.who.interfaces), 13
IRequestClassifier (interface in repoze.who.interfaces), 13

L

login() (repoze.who.interfaces.IAPI method), 12
logout() (repoze.who.interfaces.IAPI method), 12

P

PluggableAuthenticationMiddleware (class in repoze.who.middleware), 16

R

RedirectorPlugin (class in repoze.who.plugins.redirector), 21
remember() (repoze.who.interfaces.IAPI method), 12
remember() (repoze.who.interfaces.IIdentifier method), 14
repoze.who (module), 1
repoze.who.interfaces (module), 12
repoze.who.middleware (module), 16
repoze.who.plugins.auth_tkt (module), 20
repoze.who.plugins.basicauth (module), 21
repoze.who.plugins.htpasswd (module), 21
repoze.who.plugins.redirector (module), 21
repoze.who.plugins.sql (module), 21
RFC
    RFC 2617, 21

S

SQLAuthenticatorPlugin (class in repoze.who.plugins.sql), 21
SQLMetadataProviderPlugin (class in repoze.who.plugins.sql), 22