aiohttp_auth_autz Documentation

Release

ilex

Sep 07, 2017

Contents:

This library provides authorization and authentication middleware plugins for aiohttp servers.

These plugins are designed to be lightweight, simple, and extensible, allowing the library to be reused regardless of the backend authentication mechanism. This provides a familiar framework across projects.

There are three middleware plugins provided by the library. The `auth_middleware` plugin provides a simple system for authenticating a users credentials, and ensuring that the user is who they say they are.

The `autz_middleware` plugin provides a generic way of authorization using different authorization policies. There is the ACL authorization policy as a part of the plugin.

The `acl_middleware` plugin provides a simple access control list authorization mechanism, where users are provided access to different view handlers depending on what groups the user is a member of. It is recomended to use `autz_middleware` with ACL policy instead of this middleware.

This is a fork of aiohttp_auth library that fixes some bugs and security issues and also introduces a generic authorization `autz` middleware with built in ACL authorization policy.

CHAPTER 1

Install

Install `aiohttp_auth_autz` using `pip`:

```
$ pip install aiohttp_auth_autz
```

CHAPTER 2

License

The library is licensed under a MIT license.

2.1 Getting Started

A simple example how to use authentication and authorization middleware with an aiohttp application.

```
import asyncio

from os import urandom

import aiohttp_auth

from aiohttp import web
from aiohttp_auth import auth, autz
from aiohttp_auth.auth import auth_required
from aiohttp_auth.autz import autz_required
from aiohttp_auth.autz.policy import acl
from aiohttp_auth.permissions import Permission, Group

db = {
    'bob': {
        'password': 'bob_password',
        'groups': ['guest', 'staff']
    },
    'alice': {
        'password': 'alice_password',
        'groups': ['guest']
    }
}

# global ACL context
context = [(Permission.Allow, 'guest', {'view', }),
           (Permission.Deny, 'guest', {'edit', }),
            (Permission.Allow, 'staff', {'view', 'edit', 'admin_view'}),
            (Permission.Allow, Group.Everyone, {'view_home', })]


# create an ACL authorization policy class
class ACLAutzPolicy(acl.AbstractACLAutzPolicy):
    """The concrete ACL authorization policy."""

    def __init__(self, db, context=None):
        # do not forget to call parent __init__
        super().__init__(context)

        self.db = db

    async def acl_groups(self, user_identity):
        """Return acl groups for given user identity.

        This method should return a sequence of groups for given user_identity.

        Args:
            user_identity: User identity returned by auth.get_auth.

        Returns:
            Sequence of acl groups for the user identity.
        """
        # implement application specific logic here
        user = self.db.get(user_identity, None)
        if user is None:
            # return empty tuple in order to give a chance
            # to Group.Everyone
            return tuple()

        return user['groups']


async def login(request):
    # http://127.0.0.1:8080/login?username=bob&password=bob_password
    user_identity = request.GET.get('username', None)
    password = request.GET.get('password', None)
    if user_identity in db and password == db[user_identity]['password']:
        # remember user identity
        await auth.remember(request, user_identity)
        return web.Response(text='Ok')

    raise web.HTTPUnauthorized()


# only authenticated users can logout
# if user is not authenticated auth_required decorator
# will raise a web.HTTPUnauthorized
@auth_required
async def logout(request):
    # forget user identity
    await auth.forget(request)
    return web.Response(text='Ok')


# user should have a group with 'admin_view' permission allowed
```

```python
# if he does not autz_required will raise a web.HTTPForbidden
@autz_required('admin_view')
async def admin(request):
    return web.Response(text='Admin Page')


@autz_required('view_home')
async def home(request):
    text = 'Home page.'
    # check if current user is permitted with 'admin_view' permission
    if await autz.permit(request, 'admin_view'):
        text += ' Admin page: http://127.0.0.1:8080/admin'
    # get current user identity
    user_identity = await auth.get_auth(request)
    if user_identity is not None:
        # user is authenticated
        text += ' Logout: http://127.0.0.1:8080/logout'
    return web.Response(text=text)


# decorators can work with class based views
class MyView(web.View):
    """Class based view."""

    @autz_required('view')
    async def get(self):
        # example of permit using
        if await autz.permit(self.request, 'view'):
            return web.Response(text='View Page')
        return web.Response(text='View is not permitted')


def init_app(loop):
    app = web.Application()

    # Create an auth ticket mechanism that expires after 1 minute (60
    # seconds), and has a randomly generated secret. Also includes the
    # optional inclusion of the users IP address in the hash
    auth_policy = auth.CookieTktAuthentication(urandom(32), 60,
                                               include_ip=True)

    # Create an ACL authorization policy
    autz_policy = ACLAutzPolicy(db, context)

    # setup middlewares in aiohttp fashion
    aiohttp_auth.setup(app, auth_policy, autz_policy)

    app.router.add_get('/', home)
    app.router.add_get('/login', login)
    app.router.add_get('/logout', logout)
    app.router.add_get('/admin', admin)
    app.router.add_route('*', '/view', MyView)

    return app


loop = asyncio.get_event_loop()
app = init_app(loop)
```

```
web.run_app(app, host='127.0.0.1', loop=loop)
```

2.2 Middleware plugins

This library provides authorization and authentication middleware plugins for aiohttp servers.

These plugins are designed to be lightweight, simple, and extensible, allowing the library to be reused regardless of the backend authentication mechanism. This provides a familiar framework across projects.

There are three middleware plugins provided by the library. The `auth_middleware` plugin provides a simple system for authenticating a users credentials, and ensuring that the user is who they say they are.

The `autz_middleware` plugin provides a generic way of authorization using different authorization policies. There is the ACL authorization policy as a part of the plugin.

The `acl_middleware` plugin provides a simple access control list authorization mechanism, where users are provided access to different view handlers depending on what groups the user is a member of. It is recomended to use `autz_middleware` with ACL policy instead of this middleware.

2.2.1 Authentication Middleware Usage

The `auth_middleware` plugin provides a simple abstraction for remembering and retrieving the authentication details for a user across http requests. Typically, an application would retrieve the login details for a user, and call the remember function to store the details. These details can then be recalled in future requests. A simplistic example of users stored in a python dict would be:

```
from aiohttp_auth import auth
from aiohttp import web

# Simplistic name/password map
db = {'user': 'password',
      'super_user': 'super_password'}


async def login_view(request):
    params = await request.post()
    user = params.get('username', None)
    if (user in db and
        params.get('password', None) == db[user]):

        # User is in our database, remember their login details
        await auth.remember(request, user)
        return web.Response(body='OK'.encode('utf-8'))

    raise web.HTTPUnauthorized()
```

User data can be verified in later requests by checking that their username is valid explicity, or by using the `auth_required` decorator:

```
async def check_explicitly_view(request):
    user = await auth.get_auth(request)
    if user is None:
        # Show login page
        return web.Response(body='Not authenticated'.encode('utf-8'))
    return web.Response(body='OK'.encode('utf-8'))

@auth.auth_required
async def check_implicitly_view(request):
    # HTTPUnauthorized is raised by the decorator if user is not valid
    return web.Response(body='OK'.encode('utf-8'))
```

To end the session, the user data can be forgotten by using the `forget` function:

```
@auth.auth_required
async def logout_view(request):
    await auth.forget(request)
    return web.Response(body='OK'.encode('utf-8'))
```

The actual mechanisms for storing the authentication credentials are passed as a policy to the session manager middleware. New policies can be implemented quite simply by overriding the `AbstractAuthentication` class. The `aiohttp_auth` package currently provides two authentication policies, a cookie based policy based loosely on `mod_auth_tkt` (Apache ticket module), and a second policy that uses the `aiohttp_session` class to store authentication tickets.

The cookie based policy (`CookieTktAuthentication`) is a simple mechanism for storing the username of the authenticated user in a cookie, along with a hash value known only to the server. The cookie contains the maximum age allowed before the ticket expires, and can also use the IP address (v4 or v6) of the user to link the cookie to that address. The cookies data is not encrypted, but only holds the username of the user and the cookies expiration time, along with its security hash:

```
def init(loop):
    app = web.Application(loop=loop)

    # Create a auth ticket mechanism that expires after 1 minute (60
    # seconds), and has a randomly generated secret. Also includes the
    # optional inclusion of the users IP address in the hash
    policy = auth.CookieTktAuthentication(urandom(32), 60,
                                          include_ip=True)

    # setup middleware in aiohttp fashion
    auth.setup(app, policy)

    app.router.add_route('POST', '/login', login_view)
    app.router.add_route('GET', '/logout', logout_view)
    app.router.add_route('GET', '/test0', check_explicitly_view)
    app.router.add_route('GET', '/test1', check_implicitly_view)

    return app
```

The `SessionTktAuthentication` policy provides many of the same features, but stores the same ticket credentials in a `aiohttp_session` object, allowing different storage mechanisms such as `Redis` storage, and `EncryptedCookieStorage`:

```
from aiohttp_session import get_session, session_middleware
from aiohttp_session.cookie_storage import EncryptedCookieStorage

def init(loop):
    app = web.Application(loop=loop)

    # setup session middleware in aiohttp fashion
    storage = EncryptedCookieStorage(urandom(32))
    aiohttp_session.setup(app, storage)

    # Create an auth ticket mechanism that expires after 1 minute (60
    # seconds), and has a randomly generated secret. Also includes the
    # optional inclusion of the users IP address in the hash
    policy = auth.SessionTktAuthentication(urandom(32), 60,
                                           include_ip=True)

    # setup aiohttp_auth.auth middleware in aiohttp fashion
    auth.setup(app, policy)

    ...
```

2.2.2 Authorization Middleware Usage

The autz middleware provides follow interface to use in applications:

- Using `autz.permit` coroutine.
- Using `autz.autz_required` decorator for aiohttp handlers.

The `async def autz.permit(request, permission, context=None)` coroutine checks if permission is allowed for a given request with a given context. The authorization checking is provided by authorization policy which is set by setup function. The nature of permission and context is also determined by a policy.

The `def autz_required(permission, context=None)` decorator for aiohttp's request handlers checks if current user has requested permission with a given context. If the user does not have the correct permission it raises `web.HTTPForbidden`.

Note that context can be optional if authorization policy provides a way to specify global application context or if it does not require any. Also context parameter can be used to override global context if it is provided by authorization policy.

To use an authorization policy with autz middleware a class of policy should be created inherited from `autz.abc.AbstractAutzPolicy`. The only thing that should be implemented is `permit` method (see *Custom authorization policy for autz middleware*). The `autz` middleware has a built in ACL authorization policy (see *ACL authorization policy for autz middleware*).

The recomended way to initialize this middleware is through `aiohttp_auth.autz.setup` or `aiohttp_auth.setup` functions. As the `autz` middleware can be used only with authentication `aiohttp_auth.auth` middleware it is preferred to use `aiohttp_auth.setup`.

ACL authorization policy for autz middleware

The `autz` plugin has a built in ACL authorization policy in `autz.policy.acl` module. This module introduces an `AbstractACLAutzPolicy` - the abstract base class to create an ACL authorization policy class. The subclass should define how to retrieve user's groups.

As the library does not know how to get groups for user and it is always up to application, it provides abstract authorization ACL policy class. Subclass should implement `acl_groups` method to use it with `autz_middleware`.

Note that an ACL context can be specified globally while initializing policy or locally through `autz.permit` function's parameter. A local context will always override a global one while checking permissions. If there is no local context and global context is not set then the `permit` method will raise a `RuntimeError`.

A context is a sequence of ACL tuples which consist of an `Allow`/`Deny` action, a group, and a set of permissions for that ACL group. For example:

```
context = [(Permission.Allow, 'view_group', {'view', }),
           (Permission.Allow, 'edit_group', {'view', 'edit'}),]
```

ACL tuple sequences are checked in order, with the first tuple that matches the group the user is a member of, and includes the permission passed to the function, to be the matching ACL group. If no ACL group is found, the `permit` method returns `False`.

Groups and permissions need only be immutable objects, so can be strings, numbers, enumerations, or other immutable objects.

Note: Groups that are returned by `acl_groups` (if they are not `None`) will then be extended internally with `Group.Everyone` and `Group.AuthenticatedUser`.

Usage example:

```
from aiohttp import web
from aiohttp_auth import autz
from aiohttp_auth.autz import autz_required
from aiohttp_auth.autz.policy import acl
from aiohttp_auth.permissions import Permission


# create an acl authorization policy class
class ACLAutzPolicy(acl.AbstractACLAutzPolicy):
    """The concrete ACL authorization policy."""

    def __init__(self, users, context=None):
        # do not forget to call parent __init__
        super().__init__(context)

        # we will retrieve groups using some kind of users dict
        # here you can use db or cache or any other needed data
        self.users = users

    async def acl_groups(self, user_identity):
        """Return acl groups for given user identity.

        This method should return a sequence of groups for given user_identity.

        Args:
            user_identity: User identity returned by auth.get_auth.

        Returns:
            Sequence of acl groups (possibly empty) for the user identity or None.
        """
        # implement application specific logic here
        user = self.users.get(user_identity, None)
        if user is None:
            return None

        return user['groups']


def init(loop):
    app = web.Application(loop=loop)
    ...
    # here you need to initialize aiohttp_auth.auth middleware
    auth_policy = ...
    ...
    users = ...
    # Create application global context.
    # It can be overridden in autz.permit fucntion or in
    # autz_required decorator using local context explicitly.
    context = [(Permission.Allow, 'view_group', {'view', }),
               (Permission.Allow, 'edit_group', {'view', 'edit'})]
    autz_policy = ACLAutzPolicy(users, context)

    # install auth and autz middleware in aiohttp fashion
    aiohttp_auth.setup(app, auth_policy, autz_policy)


# authorization using autz decorator applying to app handler
@autz_required('view')
async def handler_view(request):
    # authorization using permit
    if await autz.permit(request, 'edit'):
        pass


local_context = [(Permission.Deny, 'view_group', {'view', })]

# authorization using autz decorator applying to app handler
# using local_context to override global one.
@autz_required('view', local_context)
async def handler_view_local(request):
    # authorization using permit and local_context to
    # override global one
    if await autz.permit(request, 'edit', local_context):
        pass
```

Custom authorization policy for autz middleware

Tha `autz` middleware makes it possible to use custom athorization policy with the same `autz` public interface for checking user permissions. The follow example shows how to create such simple custom policy:

```
from aiohttp import web
from aiohttp_auth import autz, auth
from aiohttp_auth.autz import autz_required
from aiohttp_auth.autz.abc import AbstractAutzPolicy

class CustomAutzPolicy(AbstractAutzPolicy):

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

    async def permit(self, user_identity, permission, context=None):
        # All we need is to implement this method

        if permission == 'admin':
            # only admin_user_identity is allowed for 'admin' permission
            if user_identity == self.admin_user_identity:
                return True

            # forbid anyone else
            return False

        # allow any other permissions for all users
        return True


def init(loop):
    app = web.Application(loop=loop)
    ...
    # here you need to initialize aiohttp_auth.auth middleware
    auth_policy = ...
    ...
    # create custom authorization policy
    autz_policy = CustomAutzPolicy(admin_user_identity='Bob')

    # install auth and autz middleware in aiohttp fashion
    aiohttp_auth.setup(app, auth_policy, autz_policy)


# authorization using autz decorator applying to app handler
@autz_required('admin')
async def handler_admin(request):
    # only Bob can run this handler

    # authorization using permit
    if await autz.permit(request, 'admin'):
        # only Bob can get here
        pass


@autz_required('guest')
async def handler_guest(request):
    # everyone can run this handler

    # authorization using permit
    if await autz.permit(request, 'guest'):
        # everyone can get here
        pass
```

2.2.3 ACL Middleware Usage

The `acl_middleware`` plugin (provided by the `aiohttp_auth` library), is layered on top of the `auth_middleware` plugin, and provides a access control list (ACL) system similar to that used by the Pyramid WSGI module.

Each user in the system is assigned a series of groups. Each group in the system can then be assigned permissions that they are allowed (or not allowed) to access. Groups and permissions are user defined, and need only be immutable objects, so they can be strings, numbers, enumerations, or other immutable objects.

To specify what groups a user is a member of, a function is passed to the `acl_middleware` factory which taks a `user_id` (as returned from the `auth.get_auth` function) as a parameter, and expects a sequence of permitted ACL groups to be returned. This can be a empty tuple to represent no explicit permissions, or None to explicitly forbid this particular `user_id`. Note that the `user_id` passed may be `None` if no authenticated user exists. Building apon our example, a function may be defined as:

```
from aiohttp import web
from aiohttp_auth import acl, auth
import aiohttp_session

group_map = {'user': (,),
             'super_user': ('edit_group',),}

async def acl_group_callback(user_id):
    # The user_id could be None if the user is not authenticated, but in
    # our example, we allow unauthenticated users access to some things, so
    # we return an empty tuple.
    return group_map.get(user_id, tuple())

def init(loop):
    ...

    app = web.Application(loop=loop)
    # setup session middleware
    storage = aiohttp_session.EncryptedCookieStorage(urandom(32))
    aiohttp_session.setup(app, storage)

    # setup aiohttp_auth.auth middleware
    policy = auth.SessionTktAuthentication(urandom(32), 60, include_ip=True)
    auth.setup(app, policy)

    # setup aiohttp_auth.acl middleware
    acl.setup(app, acl_group_callback)

    ...
```

Note that the ACL groups returned by the function will be modified by the `acl_middleware` to also include the `Group.Everyone` group (if the value returned is not `None`), and also the `Group.AuthenticatedUser` if the `user_id` is not `None`.

Instead of `acl_group_callback` as a coroutine the `AbstractACLGroupsCallback` class can be used (all you need is to override `acl_groups` method):

```
from aiohttp import web
from aiohttp_auth import acl, auth
from aiohttp_auth.acl.abc import AbstractACLGroupsCallback
import aiohttp_session


class ACLGroupsCallback(AbstractACLGroupsCallback):
    def __init__(self, cache):
        # Save here data you need to retrieve groups
        # for example cache or db connection
        self.cache = cache

    async def acl_groups(self, user_id):
        # override abstract method with needed logic
        user = self.cache.get(user_id, None)
        ...
        groups = user.groups() if user else tuple()
        return groups
```

```
def init(loop):
    ...

    app = web.Application(loop=loop)
    # setup session middleware
    storage = aiohttp_session.EncryptedCookieStorage(urandom(32))
    aiohttp_session.setup(app, storage)

    # setup aiohttp_auth.auth middleware
    policy = auth.SessionTktAuthentication(urandom(32), 60, include_ip=True)
    auth.setup(app, policy)

    # setup aiohttp_auth.acl middleware
    cache = ...
    acl_groups_callback = ACLGroupsCallback(cache)
    acl.setup(app, acl_group_callback)

    ...
```

With the groups defined, an ACL context can be specified for looking up what permissions each group is allowed to access. A context is a sequence of ACL tuples which consist of a `Allow`/`Deny` action, a group, and a sequence of permissions for that ACL group. For example:

```
from aiohttp_auth.permissions import Group, Permission

context = [(Permission.Allow, Group.Everyone, ('view',)),
           (Permission.Allow, Group.AuthenticatedUser, ('view', 'view_extra')),
           (Permission.Allow, 'edit_group', ('view', 'view_extra', 'edit')),]
```

Views can then be defined using the `acl_required` decorator, allowing only specific users access to a particular view. The `acl_required` decorator specifies a permission required to access the view, and a context to check against:

```
@acl_required('view', context)
async def view_view(request):
    return web.Response(body='OK'.encode('utf-8'))

@acl_required('view_extra', context)
async def view_extra_view(request):
    return web.Response(body='OK'.encode('utf-8'))

@acl_required('edit', context)
async def edit_view(request):
    return web.Response(body='OK'.encode('utf-8'))
```

In our example, non-logged in users will have access to the view_view, 'user' will have access to both the view_view and view_extra_view, and 'super_user' will have access to all three views. If no ACL group of the user matches the ACL permission requested by the view, the decorator raises `web.HTTPForbidden`.

ACL tuple sequences are checked in order, with the first tuple that matches the group the user is a member of, AND includes the permission passed to the function, declared to be the matching ACL group. This means that if the ACL context was modified to:

```
context = [(Permission.Allow, Group.Everyone, ('view',)),
           (Permission.Deny, 'super_user', ('view_extra')),
           (Permission.Allow, Group.AuthenticatedUser, ('view', 'view_extra')),
           (Permission.Allow, 'edit_group', ('view', 'view_extra', 'edit')),]
```

In this example the 'super_user' would be denied access to the view_extra_view even though they are an `AuthenticatedUser` and in the 'edit_group'.

2.3 API Documentation

2.3.1 Authentication Middleware API

Public Middleware API

Athentication middleware.

`aiohttp_auth.auth.auth.`**`setup`**`(`*app*, *policy*`)`
Setup middleware in aiohttp fashion.

Parameters

- **`app`** – aiohttp Application object.
- **`policy`** – An authentication policy with a base class of AbstractAuthentication.

`aiohttp_auth.auth.auth.`**`auth_middleware`**`(`*policy*`)`
Return an authentication middleware factory.

The middleware is for use by the aiohttp application object.

Parameters **`policy`** – A authentication policy with a base class of AbstractAuthentication.

`aiohttp_auth.auth.auth.`**`get_auth`**`(`*request*`)`
Return the user_id associated with a particular request.

Parameters **`request`** – aiohttp Request object.

Returns The user_id associated with the request, or None if no user is associated with the request.

Raises `RuntimeError` – Middleware is not installed

`aiohttp_auth.auth.auth.`**`remember`**`(`*request*, *user_id*`)`
Called to store and remember the userid for a request.

Parameters

- **`request`** – aiohttp Request object.
- **`user_id`** – String representing the user_id to remember

Raises `RuntimeError` – Middleware is not installed

`aiohttp_auth.auth.auth.`**`forget`**`(`*request*`)`
Called to forget the userid for a request.

Parameters **`request`** – aiohttp Request object.

Raises `RuntimeError` – Middleware is not installed.

Decorators

Authentication decorators.

`aiohttp_auth.auth.decorators.`**`auth_required`**(*func*)

Decorator to check if an user has been authenticated for this request.

Allows views to be decorated like:

```
@auth_required
async def view_func(request):
    pass
```

providing a simple means to ensure that whoever is calling the function has the correct authentication details.

Warning: Changed in version 0.2.0: In versions prior 0.2.0 the `web.HTTPForbidden` was raised (status code 403) if user was not authenticated. Now the `web.HTTPUnauthorized` (status code 401) is raised to distinguish authentication error from authorization one.

Parameters **`func`** – Function object being decorated.

Returns A function object that will raise `web.HTTPUnauthorized()` if the passed request does not have the correct permissions to access the view.

Abstract Authentication Policy

`class` `aiohttp_auth.auth.abstract_auth.`**`AbstractAuthentication`**

Abstract authentication policy class

`forget`(*request*)

Abstract function called to forget the userid for a request

Parameters **`request`** – aiohttp Request object

`get`(*request*)

Abstract function called to get the user_id for the request.

Parameters **`request`** – aiohttp Request object.

Returns The user_id for the request, or None if the user_id is not authenticated.

`process_response`(*request, response*)

Called to perform any processing of the response required (setting cookie data, etc).

Default implementation does nothing.

Parameters

- **`request`** – aiohttp Request object.
- **`response`** – response object returned from the handled view

`remember`(*request, user_id*)

Abstract function called to store the user_id for a request.

Parameters

- **`request`** – aiohttp Request object.
- **`user_id`** – String representing the user_id to remember

Abstract Ticket Authentication Policy

class aiohttp_auth.auth.ticket_auth.**TktAuthentication**(*secret*, *max_age*, *reissue_time=None*, *include_ip=False*, *cookie_name='AUTH_TKT'*)

Ticket authentication mechanism based on the ticket_auth library.

This class is an abstract class that creates a ticket and validates it. Storage of the ticket data itself is abstracted to allow different implementations to store the cookie differently (encrypted, server side etc).

__init__(*secret*, *max_age*, *reissue_time=None*, *include_ip=False*, *cookie_name='AUTH_TKT'*)
Initializes the ticket authentication mechanism.

Parameters

- **secret** – Byte sequence used to initialize the ticket factory.
- **max_age** – Integer representing the number of seconds to allow the ticket to remain valid for after being issued.
- **reissue_time** – Integer representing the number of seconds before a valid login will cause a ticket to be reissued. If this value is 0, a new ticket will be reissued on every request which requires authentication. If this value is None, no tickets will be reissued, and the max_age will always expire the ticket.
- **include_ip** – If true, requires the clients ip details when calculating the ticket hash
- **cookie_name** – Name to use to reference the ticket details.

cookie_name
Returns the name of the cookie stored in the session

forget(*request*)
Called to forget the userid for a request

This function calls the forget_ticket() function to forget the ticket associated with this request.

Parameters request – aiohttp Request object

forget_ticket(*request*)
Abstract function called to forget the ticket data for a request.

Parameters request – aiohttp Request object.

get(*request*)
Gets the user_id for the request.

Gets the ticket for the request using the get_ticket() function, and authenticates the ticket.

Parameters request – aiohttp Request object.

Returns The userid for the request, or None if the ticket is not authenticated.

get_ticket(*request*)
Abstract function called to return the ticket for a request.

Parameters request – aiohttp Request object.

Returns A ticket (string like) object, or None if no ticket is available for the passed request.

process_response(*request*, *response*)
If a reissue was requested, only reissue if the response was a valid 2xx response

remember (*request*, *user_id*)
Called to store the userid for a request.

This function creates a ticket from the request and user_id, and calls the abstract function remember_ticket() to store the ticket.

Parameters

- **request** – aiohttp Request object.
- **user_id** – String representing the user_id to remember

remember_ticket (*request*, *ticket*)
Abstract function called to store the ticket data for a request.

Parameters

- **request** – aiohttp Request object.
- **ticket** – String like object representing the ticket to be stored.

Concrete Ticket Authentication Policies

class aiohttp_auth.auth.cookie_ticket_auth.**CookieTktAuthentication** (*secret*, *max_age*, *reissue_time=None*, *include_ip=False*, *cookie_name='AUTH_TKT'*)
Ticket authentication mechanism based on the ticket_auth library, with ticket data being stored as a cookie in the response.

__init__ (*secret*, *max_age*, *reissue_time=None*, *include_ip=False*, *cookie_name='AUTH_TKT'*)
Initializes the ticket authentication mechanism.

Parameters

- **secret** – Byte sequence used to initialize the ticket factory.
- **max_age** – Integer representing the number of seconds to allow the ticket to remain valid for after being issued.
- **reissue_time** – Integer representing the number of seconds before a valid login will cause a ticket to be reissued. If this value is 0, a new ticket will be reissued on every request which requires authentication. If this value is None, no tickets will be reissued, and the max_age will always expire the ticket.
- **include_ip** – If true, requires the clients ip details when calculating the ticket hash
- **cookie_name** – Name to use to reference the ticket details.

cookie_name
Returns the name of the cookie stored in the session

forget (*request*)
Called to forget the userid for a request

This function calls the forget_ticket() function to forget the ticket associated with this request.

Parameters **request** – aiohttp Request object

forget_ticket(*request*)
Called to forget the ticket data a request

Parameters **request** – aiohttp Request object.

get(*request*)
Gets the user_id for the request.

Gets the ticket for the request using the get_ticket() function, and authenticates the ticket.

Parameters **request** – aiohttp Request object.

Returns The userid for the request, or None if the ticket is not authenticated.

get_ticket(*request*)
Called to return the ticket for a request.

Parameters **request** – aiohttp Request object.

Returns A ticket (string like) object, or None if no ticket is available for the passed request.

process_response(*request*, *response*)
Called to perform any processing of the response required.

This function stores any cookie data in the COOKIE_AUTH_KEY as a cookie in the response object. If the value is a empty string, the associated cookie is deleted instead.

This function requires the response to be a aiohttp Response object, and assumes that the response has not prepared if the remember or forget functions are called during the request.

Parameters

- **request** – aiohttp Request object.
- **response** – response object returned from the handled view

Raises `RuntimeError` – Raised if response has already prepared.

remember(*request*, *user_id*)
Called to store the userid for a request.

This function creates a ticket from the request and user_id, and calls the abstract function remember_ticket() to store the ticket.

Parameters

- **request** – aiohttp Request object.
- **user_id** – String representing the user_id to remember

remember_ticket(*request*, *ticket*)
Called to store the ticket data for a request.

Ticket data is stored in COOKIE_AUTH_KEY in the request object, and written as cookie data to the response during the process_response() function.

Parameters

- **request** – aiohttp Request object.
- **ticket** – String like object representing the ticket to be stored.

class aiohttp_auth.auth.session_ticket_auth.**SessionTktAuthentication**(*secret*, *max_age*, *reissue_time=None*, *include_ip=False*, *cookie_name='AUTH_TKT'*)

Ticket authentication mechanism based on the ticket_auth library, with ticket data being stored in the aiohttp_session object.

__init__(*secret*, *max_age*, *reissue_time=None*, *include_ip=False*, *cookie_name='AUTH_TKT'*)
Initializes the ticket authentication mechanism.

Parameters

- **secret** – Byte sequence used to initialize the ticket factory.
- **max_age** – Integer representing the number of seconds to allow the ticket to remain valid for after being issued.
- **reissue_time** – Integer representing the number of seconds before a valid login will cause a ticket to be reissued. If this value is 0, a new ticket will be reissued on every request which requires authentication. If this value is None, no tickets will be reissued, and the max_age will always expire the ticket.
- **include_ip** – If true, requires the clients ip details when calculating the ticket hash
- **cookie_name** – Name to use to reference the ticket details.

cookie_name
Returns the name of the cookie stored in the session

forget(*request*)
Called to forget the userid for a request

This function calls the forget_ticket() function to forget the ticket associated with this request.

Parameters **request** – aiohttp Request object

forget_ticket(*request*)
Called to forget the ticket data a request

Parameters **request** – aiohttp Request object.

get(*request*)
Gets the user_id for the request.

Gets the ticket for the request using the get_ticket() function, and authenticates the ticket.

Parameters **request** – aiohttp Request object.

Returns The userid for the request, or None if the ticket is not authenticated.

get_ticket(*request*)
Called to return the ticket for a request.

Parameters **request** – aiohttp Request object.

Returns A ticket (string like) object, or None if no ticket is available for the passed request.

process_response(*request*, *response*)
If a reissue was requested, only reissue if the response was a valid 2xx response

remember(*request*, *user_id*)
Called to store the userid for a request.

This function creates a ticket from the request and user_id, and calls the abstract function remember_ticket() to store the ticket.

Parameters

- **request** – aiohttp Request object.
- **user_id** – String representing the user_id to remember

remember_ticket (*request*, *ticket*)

Called to store the ticket data for a request.

Ticket data is stored in the aiohttp_session object

Parameters

- **request** – aiohttp Request object.
- **ticket** – String like object representing the ticket to be stored.

2.3.2 Authorization Middleware API

Setup auth and autz

`aiohttp_auth.`**setup** (*app*, *auth_policy*, *autz_policy*)

Setup auth and autz middleware in aiohttp fashion.

Parameters

- **app** – aiohttp Application object.
- **auth_policy** – An authentication policy with a base class of AbstractAuthentication.
- **autz_policy** – An authorization policy with a base class of AbstractAutzPolicy

Public Middleware API

Authorization middleware.

`aiohttp_auth.autz.autz.`**setup** (*app*, *autz_policy*)

Setup an authorization middleware in `aiohttp` fashion.

Note that `aiohttp_auth.auth` middleware should be installed too to use `autz` middleware. So the preferred way to install this middleware is to use global `aiohttp_auth.setup` function.

Parameters

- **app** – aiohttp `Application` object.
- **autz_policy** – A subclass of `aiohttp_auth.autz.abc.AbstractAutzPolicy`.

`aiohttp_auth.autz.autz.`**autz_middleware** (*autz_policy*)

Return authorization middleware factory.

Return `aiohttp_auth.autz` middleware factory for use by the `aiohttp` application object. This middleware can be used only with `aiohttp_auth.auth` middleware installed.

The `autz` middleware provides follow interface to use in applications:

- Using `autz.permit` coroutine.
- Using `autz.autz_required` decorator for `aiohttp` handlers.

Note that the recomended way to initialize this middleware is through `aiohttp_auth.autz.setup` or `aiohttp_auth.setup` functions. As the `autz` middleware can be used only with authentication `aiohttp_auth.auth` middleware it is preferred to use `aiohttp_auth.setup`.

Parameters **`autz_policy`** – a subclass of `aiohttp_auth.autz.abc.AbstractAutzPolicy`.

Returns An `aiohttp` middleware factory.

`aiohttp_auth.autz.autz.`**`permit`**(*request*, *permission*, *context=None*)

Check if permission is allowed for given request with given context.

The authorization checking is provided by authorization policy which is set by setup function. The nature of permission and context is also determined by the given policy.

Note that this coroutine uses `aiohttp_auth.auth.get_auth` coroutine to determine `user_identity` for given request. So that middleware should be installed too.

Note that some additional exceptions could be raised by certain policy while checking the permission.

Parameters

- **`request`** – aiohttp `Request` object.
- **`permission`** – The specific permission requested.
- **`context`** – A context provided for checking permissions. Could be optional if authorization policy provides a way to specify a global application context.

Returns `True` if permission is allowed `False` otherwise.

Raises `RuntimeError` – If `auth` or `autz` middleware is not installed.

Decorators

Authorization decorators.

`aiohttp_auth.autz.decorators.`**`autz_required`**(*permission*, *context=None*)

Create decorator to check if user has requested permission.

This function constructs a decorator that can be used to check a aiohttp's view for authorization before calling it. It uses the `autz.permit` function to check the request against the passed permission and context. If the user does not have the correct permission to run this function, it raises `web.HTTPForbidden`.

Note that context can be optional if authorization policy provides a way to specify global application context. Also context parameter can be used to override global context if it is provided by authorization policy.

Note that some exceptions could be raised by certain policy while checking the permission.

Parameters

- **`permission`** – The specific permission requested.
- **`context`** – A context provided for checking permissions. Could be optional if authorization policy provides a way to specify a global application context.

Returns A decorator which will check the request passed has the permission for the given context. The decorator will raise `web.HTTPForbidden` if the user does not have the correct permissions to access the view.

ACL Authorization Policy

ACL authorization policy.

This module introduces `AbstractACLAutzPolicy` - an abstract base class to create ACL authorization policy class. The subclass should define how to retrieve user's groups.

class aiohttp_auth.autz.policy.acl.**AbstractACLAutzPolicy**(*context=None*)

Abstract base class for ACL authorization policy.

As the library does not know how to get groups for user and it is always up to application, it provides abstract authorization ACL policy class. Subclass should implement `acl_groups` method to use it with `autz_middleware`.

Note that an ACL context can be specified globally while initializing policy or locally through `autz.permit` function's parameter. A local context will always override a global one while checking permissions. If there is no local context and global context is not set then the `permit` method will raise a `RuntimeError`.

A context is a sequence of ACL tuples which consist of an `Allow`/`Deny` action, a group, and a set of permissions for that ACL group. For example:

```
context = [(Permission.Allow, 'view_group', {'view', }),
           (Permission.Allow, 'edit_group', {'view', 'edit'}),]
```

ACL tuple sequences are checked in order, with the first tuple that matches the group the user is a member of, and includes the permission passed to the function, to be the matching ACL group. If no ACL group is found, the `permit` method returns `False`.

Groups and permissions need only be immutable objects, so can be strings, numbers, enumerations, or other immutable objects.

Note: Groups that are returned by `acl_groups` (if they are not `None`) will then be extended internally with `Group.Everyone` and `Group.AuthenticatedUser`.

Usage example:

```
from aiohttp import web
from aiohttp_auth import autz
from aiohttp_auth.autz import autz_required
from aiohttp_auth.autz.policy import acl
from aiohttp_auth.permissions import Permission


class ACLAutzPolicy(acl.AbstractACLAutzPolicy):
    def __init__(self, users, context=None):
        super().__init__(context)

        # we will retrieve groups using some kind of users dict
        # here you can use db or cache or any other needed data
        self.users = users

    async def acl_groups(self, user_identity):
        # implement application specific logic here
        user = self.users.get(user_identity, None)
        if user is None:
            return None

        return user['groups']
```

```
def init(loop):
    app = web.Application(loop=loop)
    ...
    # here you need to initialize aiohttp_auth.auth middleware
    ...
    users = ...
    # Create application global context.
    # It can be overridden in autz.permit fucntion or in
    # autz_required decorator using local context explicitly.
    context = [(Permission.Allow, 'view_group', {'view', }),
               (Permission.Allow, 'edit_group', {'view', 'edit'})]
    autz.setup(app, ACLAutzPolicy(users, context))


# authorization using autz decorator applying to app request handler
@autz_required('view')
async def handler_view(request):
    # authorization using permit
    if await autz.permit(request, 'edit'):
        pass
```

`__init__`(*context=None*)

Initialize ACL authorization policy.

Parameters **`context`** – global ACL context, default to `None`. Should be a list of ACL rules.

`acl_groups`(*user_identity*)

Return ACL groups for given user identity.

Subclass should implement this method to return a sequence of groups for given `user_identity`.

Parameters **`user_identity`** – User identity returned by `auth.get_auth`.

Returns Sequence of ACL groups for the user identity (could be empty to give a chance to `Group.Everyone` and `Group.AuthenticatedUser`) or `None` (`permit` will always return `False`).

`permit`(*user_identity*, *permission*, *context=None*)

Check if user is allowed for given permission with given context.

Parameters

- **`user_identity`** – Identity of the user returned by `aiohttp_auth.auth.get_auth` function
- **`permission`** – The specific permission requested.
- **`context`** – A context provided for checking permissions. Could be optional if a global context is specified through policy initialization.

Returns `True` if permission is allowed, `False` otherwise.

Raises `RuntimeError` – If there is neither global context nor local one.

2.3.3 ACL Middleware API

Public Middleware API

ACL middleware.

`aiohttp_auth.acl.acl.`**`setup`**`(`*app*, *groups_callback*`)`

Setup middleware in aiohttp fashion.

Parameters

- **`app`** – aiohttp Application object.
- **`groups_callback`** – This is a callable which takes a user_id (as returned from the auth.get_auth function), and expects a sequence of permitted ACL groups to be returned. This can be a empty tuple to represent no explicit permissions, or None to explicitly forbid this particular user_id. Note that the user_id passed may be None if no authenticated user exists.

`aiohttp_auth.acl.acl.`**`acl_middleware`**`(`*callback*`)`

Return ACL middleware factory.

The middleware is for use by the aiohttp application object.

Parameters **`callback`** – This is a callable which takes a user_id (as returned from the auth.get_auth function), and expects a sequence of permitted ACL groups to be returned. This can be a empty tuple to represent no explicit permissions, or None to explicitly forbid this particular user_id. Note that the user_id passed may be None if no authenticated user exists.

Returns A aiohttp middleware factory.

`aiohttp_auth.acl.acl.`**`get_permitted`**`(`*request*, *permission*, *context*`)`

Check permission for the given request with the given context.

Return True if the one of the groups in the request has the requested permission.

The function takes a request, a permission to check for and a context. A context is a sequence of ACL tuples which consist of a Allow/Deny action, a group, and a sequence of permissions for that ACL group. For example:

```
context = [(Permission.Allow, 'view_group', ('view',)),
           (Permission.Allow, 'edit_group', ('view', 'edit')),]
```

ACL tuple sequences are checked in order, with the first tuple that matches the group the user is a member of, and includes the permission passed to the function, to be the matching ACL group. If no ACL group is found, the function returns False.

Groups and permissions need only be immutable objects, so can be strings, numbers, enumerations, or other immutable objects.

Parameters

- **`request`** – aiohttp Request object.
- **`permission`** – The specific permission requested.
- **`context`** – A sequence of ACL tuples.

Returns The function gets the groups by calling get_user_groups() and returns true if the groups are Allowed the requested permission, false otherwise.

Raises `RuntimeError` – If the ACL middleware is not installed.

`aiohttp_auth.acl.acl.`**`get_user_groups`**`(`*request*`)`

Return the groups that the user in this request has access to.

This function gets the user id from the auth.get_auth function, and passes it to the ACL callback function to get the groups.

Parameters **`request`** – aiohttp Request object.

Returns If the ACL callback function returns None, this function returns None. Otherwise this function returns the sequence of group permissions provided by the callback, plus the Everyone group. If user_id is not None, the AuthnticatedUser group is added to the groups returned by the function.

Raises `RuntimeError` – If the ACL middleware is not installed.

`aiohttp_auth.acl.acl.`**`extend_user_groups`**`(`*user_id*, *groups*`)`

Extend user groups with specific Groups.

Parameters

- **`user_id`** – User identity from get_auth.
- **`groups`** – User groups.

Returns If groups is None, this function returns None. Otherwise this function extends groups with the Everyone group. If user_id is not None, the AuthnticatedUser group is added to the groups returned by the function.

`aiohttp_auth.acl.acl.`**`get_groups_permitted`**`(`*groups*, *permission*, *context*`)`

Check if one of the groups has the requested permission.

Parameters

- **`groups`** – A set of ACL groups.
- **`permission`** – The specific permission requested.
- **`context`** – A sequence of ACL tuples.

Returns True if the groups are Allowed the requested permission, False otherwise.

Decorators

ACL middleware decorators.

`aiohttp_auth.acl.decorators.`**`acl_required`**`(`*permission*, *context*`)`

Create decorator to check given permission with given context.

Return a decorator that checks if a user has the requested permission from the passed acl context.

This function constructs a decorator that can be used to check a aiohttp's view for authorization before calling it. It uses the `get_permission()` function to check the request against the passed permission and context. If the user does not have the correct permission to run this function, it raises `web.HTTPForbidden`.

Parameters

- **`permission`** – The specific permission requested.
- **`context`** – Either a sequence of ACL tuples, or a callable that returns a sequence of ACL tuples. For more information on ACL tuples, see `get_permission()`.

Returns A decorator which will check the request passed has the permission for the given context. The decorator will raise HTTPForbidden if the user does not have the correct permissions to access the view.

AbstractACLGroupsCallback Class

class aiohttp_auth.acl.abc.**AbstractACLGroupsCallback**

Abstract base class for acl_groups_callback callabel.

User should create class deriving from this one, override acl_groups method and register object of that class with setup function as acl_groups_callback.

Usage example:

```
class ACLGroupsCallback(AbstractACLGroupsCallback):

    def __init__(self, cache):
        # store some kind of cache
        self.cache = cache

    async def acl_groups(self, user_id):
        # implement logic to return user's groups.
        user = await self.cache.get(user_id)
        return user.groups()


def init(loop):
    app = web.Application(loop=loop)
    ...
    cache = ...
    acl_groups_callback = ACLGroupsCallback(cache)

    acl.setup(app, acl_groups_callback)
    ...
```

acl_groups(*user_id*)

Return ACL groups for given user identity.

Note that the ACL groups returned by this method will be modified by the acl_middleware to also include the Group.Everyone group (if the value returned is not None), and also the Group.AuthenticatedUser if the user_id is not None.

Parameters **user_id** – User identity (as returned from the auth.get_auth function). Note that the user_id passed may be None if no authenticated user exists.

Returns A sequence of permitted ACL groups. This can be a empty tuple to represent no explicit permissions, or None to explicitly forbid this particular user_id.

2.4 Changelog

2.4.1 0.2.2 (2017-04-18)

- Move to `aiohttp` 2.x.
- Add support of middlewares decorators for `aiohttp.web.View` handlers.
- Add `uvloop` as IO loop for tests.

2.4.2 0.2.1 (2017-02-16)

- `autz` middleware:

  - Simplify `acl` authorization policy by moving permit logic into `policy.acl.AbstractACLAutzPolicy`.
  - Remove `policy.acl.AbstractACLContext` class.
  - Remove `policy.acl.NaiveACLContext` class.
  - Remove `policy.acl.ACLContext` class.

2.4.3 0.2.0 (2017-02-14)

- `acl` middleware:
  - Add `setup` function for `acl` middleware to install it in aiohttp fashion.
  - Fix bug in `acl_required` decorator.
  - Fix a possible security issue with `acl` groups. The issue is follow: the default behavior is to add `user_id` to groups for authenticated users by the acl middleware, but if `user_id` is equal to some of acl groups that user suddenly has the permissions he is not allowed for. So to avoid this kind of issue `user_id` is not added to groups any more.
  - Introduce `AbstractACLGroupsCallback` class in `acl` middleware to make it possible easily create callable object by inheriting from the abstract class and implementing `acl_groups` method. It can be useful to store additional information (such database connection etc.) within such class. An instance of this subclass can be used in place of `acl_groups_callback` parameter.
- `auth` middleware:
  - Add `setup` function for `auth` middleware to install it in aiohttp fashion.
  - `auth.auth_required` raised now a `web.HTTPUnauthorized` instead of a `web.HTTPForbidden`.
- Introduce generic authorization middleware `autz` that performs authorization through the same interface (`autz.permit` coroutine and `autz_required` decorator) but using different policies. Middleware has the ACL authorization as the built in policy which works in the same way as `acl` middleware. Users are free to add their own custom policies or to modify ACL one.
- Add global `aiohttp_auth.setup` function to install `auth` and `autz` middlewares at once in aiohttp fashion.
- Add docs.
- Rewrite tests using `pytest` and `pytest-aiohttp`.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index