Flask Principal

"I am that I am"

Introduction

Flask-Principal provides a very loose framework to tie in providers of two types of service, often located in different parts of a web application:

1. Authentication providers
2. User information providers

For example, an authentication provider may be oauth, using Flask-OAuth and the user information may be stored in a relational database. Looseness of the framework is provided by using signals as the interface.

The major components are the Identity, Needs, Permission, and the IdentityContext.

1. The Identity represents the user, and is stored/loaded from various locations (eg session) for each request. The Identity is the user's avatar to the system. It contains the access rights that the user has.

2. A Need is the smallest grain of access control, and represents a specific parameter for the situation. For example "has the admin role", "can edit blog posts".

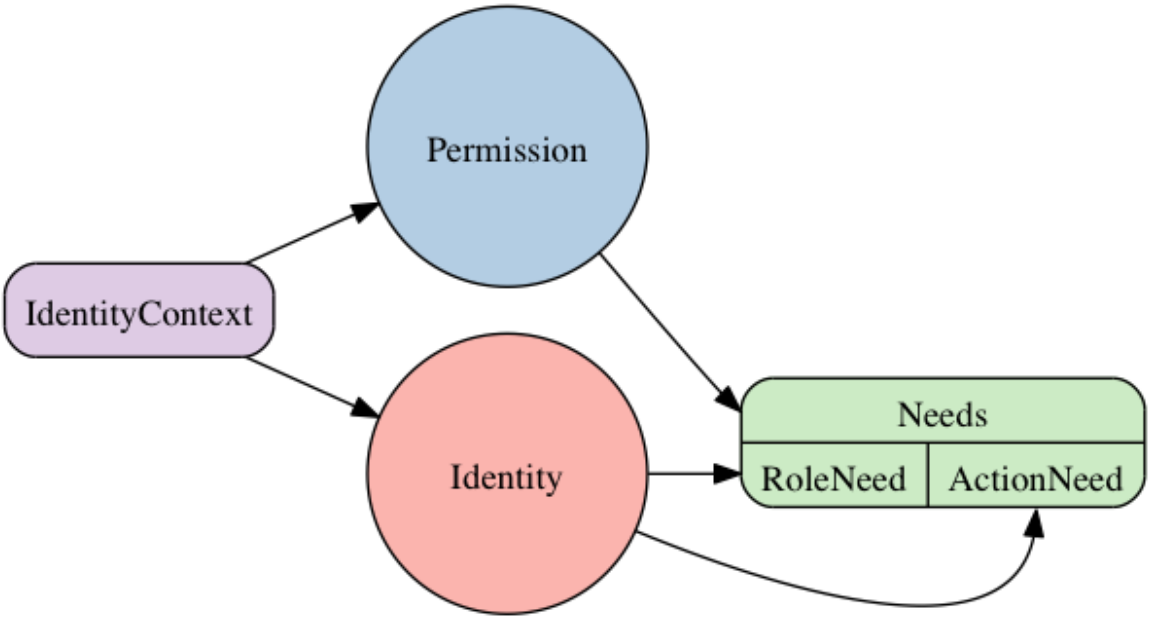
   Needs are any tuple, or probably could be object you like, but a tuple fits perfectly. The predesigned Need types (for saving your typing) are either pairs of (method, value) where method is used to specify common things such as *"role"*, *"user"*, etc. And the value is the value. An example of such is *('role', 'admin')*. Which would be a Need for a admin role. Or Triples for use-cases such as "The permission to edit a particular instance of an object or row", which might be represented as the triple *('article', 'edit', 46)*, where 46 is the key/ID for that row/object.

   Essentially, how and what Needs are is very much down to the user, and is designed loosely so that any effect can be achieved by using custom instances as Needs.

   Whilst a Need is a permission to access a resource, an Identity should provide a set of Needs that it has access to.

3. A Permission is a set of requirements, any of which should be present for access to a resource.

4. An IdentityContext is the context of a certain identity against a certain Permission. It can be used as a context manager, or a decorator.



Links

- documentation
- source
- *changelog*

Protecting access to resources

For users of Flask-Principal (not authentication providers), access restriction is easy to define as both a decorator and a context manager. A simple quickstart example is presented with commenting:

```
from flask import Flask, Response
from flask.ext.principal import Principal, Permission, RoleNeed

app = Flask(__name__)

# load the extension
principals = Principal(app)

# Create a permission with a single Need, in this case a RoleNeed.
admin_permission = Permission(RoleNeed('admin'))

# protect a view with a principal for that need
@app.route('/admin')
@admin_permission.require()
def do_admin_index():
    return Response('Only if you are an admin')

# this time protect with a context manager
@app.route('/articles')
def do_articles():
```

```
    with admin_permission.require():
        return Response('Only if you are admin')
```

Authentication providers

Authentication providers should use the *identity-changed* signal to indicate that a request has been authenticated. For example, the following code is a hypothetical example of how one might combine the popular Flask-Login extension with Flask-Principal:

```
from flask import Flask, current_app, request, session
from flask.ext.login import LoginManager, login_user, logout_user, \
     login_required, current_user
from flask.ext.wtf import Form, TextField, PasswordField, Required, Email
from flask.ext.principal import Principal, Identity, AnonymousIdentity, \
     identity_changed

app = Flask(__name__)

Principal(app)

login_manager = LoginManager(app)

@login_manager.user_loader
def load_user(userid):
    # Return an instance of the User model
    return datastore.find_user(id=userid)

class LoginForm(Form):
    email = TextField()
    password = PasswordField()

@app.route('/login', methods=['GET', 'POST'])
def login():
    # A hypothetical login form that uses Flask-WTF
    form = LoginForm()

    # Validate form input
    if form.validate_on_submit():
        # Retrieve the user from the hypothetical datastore
        user = datastore.find_user(email=form.email.data)

        # Compare passwords (use password hashing production)
        if form.password.data == user.password:
            # Keep the user info in the session using Flask-Login
            login_user(user)

            # Tell Flask-Principal the identity changed
            identity_changed.send(current_app._get_current_object(),
                                  identity=Identity(user.id))

            return redirect(request.args.get('next') or '/')

    return render_template('login.html', form=form)
```

```
@app.route('/logout')
@login_required
def logout():
    # Remove the user information from the session
    logout_user()

    # Remove session keys set by Flask-Principal
    for key in ('identity.name', 'identity.auth_type'):
        session.pop(key, None)

    # Tell Flask-Principal the user is anonymous
    identity_changed.send(current_app._get_current_object(),
                          identity=AnonymousIdentity())

    return redirect(request.args.get('next') or '/')
```

User Information providers

User information providers should connect to the *identity-loaded* signal to add any additional information to the Identity instance such as roles. The following is another hypothetical example using Flask-Login and could be combined with the previous example. It shows how one might use a role based permission scheme:

```
from flask.ext.login import current_user
from flask.ext.principal import identity_loaded, RoleNeed, UserNeed

@identity_loaded.connect_via(app)
def on_identity_loaded(sender, identity):
    # Set the identity user object
    identity.user = current_user

    # Add the UserNeed to the identity
    if hasattr(current_user, 'id'):
        identity.provides.add(UserNeed(current_user.id))

    # Assuming the User model has a list of roles, update the
    # identity with the roles that the user provides
    if hasattr(current_user, 'roles'):
        for role in current_user.roles:
            identity.provides.add(RoleNeed(role.name))
```

Granular Resource Protection

Now lets say, for example, you only want the author of a blog post to be able to edit said article. This can be achieved by creating the necessary *Need* and *Permission*objects, and adding more logic into the *identity_loaded* signal handler. For example:

```
from collections import namedtuple
from functools import partial

from flask.ext.login import current_user
from flask.ext.principal import identity_loaded, Permission, RoleNeed, \
     UserNeed

BlogPostNeed = namedtuple('blog_post', ['method', 'value'])
EditBlogPostNeed = partial(BlogPostNeed, 'edit')

class EditBlogPostPermission(Permission):
    def __init__(self, post_id):
        need = EditBlogPostNeed(unicode(post_id))
        super(EditBlogPostPermission, self).__init__(need)

@identity_loaded.connect_via(app)
def on_identity_loaded(sender, identity):
    # Set the identity user object
    identity.user = current_user

    # Add the UserNeed to the identity
    if hasattr(current_user, 'id'):
        identity.provides.add(UserNeed(current_user.id))

    # Assuming the User model has a list of roles, update the
    # identity with the roles that the user provides
    if hasattr(current_user, 'roles'):
        for role in current_user.roles:
            identity.provides.add(RoleNeed(role.name))

    # Assuming the User model has a list of posts the user
    # has authored, add the needs to the identity
    if hasattr(current_user, 'posts'):
        for post in current_user.posts:
            identity.provides.add(EditBlogPostNeed(unicode(post.id)))
```

The next step will be to protect the endpoint that allows a user to edit an article. This is done by creating a permission object on the fly using the ID of the resource, in this case the blog post:

```
@app.route('/posts/<post_id>', methods=['PUT', 'PATCH'])
def edit_post(post_id):
    permission = EditBlogPostPermission(post_id)

    if permission.can():
        # Save the edits ...
        return render_template('edit_post.html')

    abort(403)  # HTTP Forbidden
```

API

Starting the extension

class `flask_principal.`**`Principal`**(*app=None*, *use_sessions=True*, *skip_static=False*)

Principal extension

Parameters:	• **app** – The flask application to extend • **use_sessions** – Whether to use sessions to extract and store identification. • **skip_static** – Whether to ignore static endpoints.

`identity_loader`(*f*)

Decorator to define a function as an identity loader.

An identity loader function is called before request to find any provided identities. The first found identity is used to load from.

For example:

```
app = Flask(__name__)

principals = Principal(app)

@principals.identity_loader
def load_identity_from_weird_usecase():
    return Identity('ali')
```

`identity_saver`(*f*)

Decorator to define a function as an identity saver.

An identity loader saver is called when the identity is set to persist it for the next request.

For example:

```
app = Flask(__name__)

principals = Principal(app)

@principals.identity_saver
def save_identity_to_weird_usecase(identity):
    my_special_cookie['identity'] = identity
```

`set_identity`(*identity*)

Set the current identity.

Parameters:	**identity** – The identity to set

Main Types

class `flask_principal.`**`Permission`**(**needs*)

Represents needs, any of which must be present to access a resource

Parameters:	**needs** – The needs for this permission

allows(*identity*)

Whether the identity can access this permission.

Parameters:	**identity** – The identity

can()

Whether the required context for this permission has access

This creates an identity context and tests whether it can access this permission

difference(*other*)

Create a new permission consisting of requirements in this permission and not in the other.

issubset(*other*)

Whether this permission needs are a subset of another

Parameters:	**other** – The other permission

require(*http_exception=None*)

Create a principal for this permission.

The principal may be used as a context manager, or a decroator.

If `http_exception` is passed then `abort()` will be called with the HTTP exception code. Otherwise a `PermissionDenied` exception will be raised if the identity does not meet the requirements.

Parameters:	**http_exception** – the HTTP exception code (403, 401 etc)

reverse()

Returns reverse of current state (needs->excludes, excludes->needs)

test(*http_exception=None*)

Checks if permission available and raises relevant exception if not. This is useful if you just want to check permission without wrapping everything in a require() block.

This is equivalent to:

```
with permission.require():
    pass
```

union(*other*)

Create a new permission with the requirements of the union of this and other.

Parameters:	**other** – The other permission

class `flask_principal.`**`Identity`**(*id*, *auth_type=None*)

Represent the user's identity.

Parameters:	• **id** – The user id • **auth_type** – The authentication type used to confirm the user's identity.

The identity is used to represent the user's identity in the system. This object is created on login, or on the start of the request as loaded from the user's session.

Once loaded it is sent using the *identity-loaded* signal, and should be populated with additional required information.

Needs that are provided by this identity should be added to the *provides* set after loading.

`can`(*permission*)

Whether the identity has access to the permission.

Parameters:	**permission** – The permission to test provision for.

class `flask_principal.`**`AnonymousIdentity`**

An anonymous identity

class `flask_principal.`**`IdentityContext`**(*permission*, *http_exception=None*)

The context of an identity for a permission.

> **Note**
> The principal is usually created by the flaskext.Permission.require method call for normal use-cases.

The principal behaves as either a context manager or a decorator. The permission is checked for provision in the identity, and if available the flow is continued (context manager) or the function is executed (decorator).

`can`()

Whether the identity has access to the permission

`http_exception` = *None*

The permission of this principal

`identity`

The identity of this principal

Predefined Need Types

class `flask_principal.`**`Need`**

A required need

This is just a named tuple, and practically any tuple will do.

The `method` attribute can be used to look up element 0, and the `value` attribute can be used to look up element 1.

`flask_principal.`**`RoleNeed`**

`flask_principal.`**`UserNeed`**

class `flask_principal.`**`ItemNeed`**

A required item need

An item need is just a named tuple, and practically any tuple will do. In addition to other Needs, there is a type, for example this could be specified as:

```
ItemNeed('update', 27, 'posts')
('update', 27, 'posts') # or like this
```

And that might describe the permission to update a particular blog post. In reality, the developer is free to choose whatever convention the permissions are.

Signals

`identity_changed`

Signal sent when the identity for a request has been changed.

`identity_loaded`

Signal sent when the identity has been initialised for a request.