Fork me on GitHub

Flask-Login

Flask-Login provides user session management for Flask. It handles the common tasks of logging in, logging out, and remembering your users' sessions over extended periods of time.

It will:

- Store the active user's ID in the session, and let you log them in and out easily.
- Let you restrict views to logged-in (or logged-out) users.
- Handle the normally-tricky "remember me" functionality.
- Help protect your users' sessions from being stolen by cookie thieves.
- Possibly integrate with Flask-Principal or other authorization extensions later on.

However, it does not:

- Impose a particular database or other storage method on you. You are entirely in charge of how the user is loaded.
- Restrict you to using usernames and passwords, OpenIDs, or any other method of authenticating.
- Handle permissions beyond "logged in or not."
- Handle user registration or account recovery.

v: latest

Fork me on GitHub

Installation

Install the extension with pip:

```
$ pip install flask-login
```

Configuring your Application

The most important part of an application that uses Flask-Login is the `LoginManager` class. You should create one for your application somewhere in your code, like this:

```
login_manager = LoginManager()
```

The login manager contains the code that lets your application and Flask-Login work together, such as how to load a user from an ID, where to send users when they need to log in, and the like.

Once the actual application object has been created, you can configure it for login with:

```
login_manager.init_app(app)
```

How it Works

You will need to provide a `user_loader` callback. This callback is used to reload the user object from the user ID stored in the session. It should take the `unicode` ID of a user, and return the corresponding user object. For example:

```
@login_manager.user_loader
def load_user(user_id):
    return User.get(user_id)
```

It should return None (**not raise an exception**) if the ID is not valid. (In that case, the ID will manually be removed from the session and processing will continue.)

Your User Class

The class that you use to represent users needs to implement these properti v: latest methods:

Fork me on GitHub

`is_authenticated`
: This property should return `True` if the user is authenticated, i.e. they have provided valid credentials. (Only authenticated users will fulfill the criteria of `login_required`.)

`is_active`
: This property should return `True` if this is an active user - in addition to being authenticated, they also have activated their account, not been suspended, or any condition your application has for rejecting an account. Inactive accounts may not log in (without being forced of course).

`is_anonymous`
: This property should return `True` if this is an anonymous user. (Actual users should return `False` instead.)

`get_id()`
: This method must return a `unicode` that uniquely identifies this user, and can be used to load the user from the `user_loader` callback. Note that this **must** be a `unicode` - if the ID is natively an `int` or some other type, you will need to convert it to `unicode`.

To make implementing a user class easier, you can inherit from `UserMixin`, which provides default implementations for all of these properties and methods. (It's not required, though.)

Login Example

Once a user has authenticated, you log them in with the `login_user` function.

For example:

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    # Here we use a class of some kind to represent and validate our
    # client-side form data. For example, WTForms is a library that will
    # handle this for us, and we use a custom LoginForm to validate.
    form = LoginForm()
    if form.validate_on_submit():
        # Login and validate the user.
        # user should be an instance of your `User` class
        login_user(user)

        flask.flash('Logged in successfully.')

        next = flask.request.args.get('next')
        # is_safe_url should check if the url is safe for redirects.
        # See http://flask.pocoo.org/snippets/62/ for an example.
        if not is_safe_url(next):
            return flask.abort(400)
```

v: latest


```
        return flask.redirect(next or flask.url_for('index'))
    return flask.render_template('login.html', form=form)
```

Warning: You MUST validate the value of the `next` parameter. If you do not, your application will be vulnerable to open redirects. For an example implementation of `is_safe_url` see this Flask Snippet.

It's that simple. You can then access the logged-in user with the `current_user` proxy, which is available in every template:

```
{% if current_user.is_authenticated %}
  Hi {{ current_user.name }}!
{% endif %}
```

Views that require your users to be logged in can be decorated with the `login_required` decorator:

```
@app.route("/settings")
@login_required
def settings():
    pass
```

When the user is ready to log out:

```
@app.route("/logout")
@login_required
def logout():
    logout_user()
    return redirect(somewhere)
```

They will be logged out, and any cookies for their session will be cleaned up.

Customizing the Login Process

By default, when a user attempts to access a `login_required` view without being logged in, Flask-Login will flash a message and redirect them to the log in view. (If the login view is not set, it will abort with a 401 error.)

The name of the log in view can be set as `LoginManager.login_view`. For example:

```
login_manager.login_view = "users.login"
```

The default message flashed is `Please log in to access this page.` To customize the message, set `LoginManager.login_message`:

```
login_manager.login_message = u"Bonvolu ensaluti por uzi tiun paĝon."
```

v: latest

To customize the message category, set `LoginManager.login_message_category`:


```
login_manager.login_message_category = "info"
```

When the log in view is redirected to, it will have a `next` variable in the query string, which is the page that the user was trying to access. Alternatively, if `USE_SESSION_FOR_NEXT` is `True`, the page is stored in the session under the key `next`.

If you would like to customize the process further, decorate a function with `LoginManager.unauthorized_handler`:

```
@login_manager.unauthorized_handler
def unauthorized():
    # do stuff
    return a_response
```

Login using Authorization header

Caution:

This method will be deprecated; use the `request_loader` below instead.

Sometimes you want to support Basic Auth login using the `Authorization` header, such as for api requests. To support login via header you will need to provide a `header_loader` callback. This callback should behave the same as your `user_loader` callback, except that it accepts a header value instead of a user id. For example:

```
@login_manager.header_loader
def load_user_from_header(header_val):
    header_val = header_val.replace('Basic ', '', 1)
    try:
        header_val = base64.b64decode(header_val)
    except TypeError:
        pass
    return User.query.filter_by(api_key=header_val).first()
```

By default the `Authorization` header's value is passed to your `header_loader` callback. You can change the header used with the `AUTH_HEADER_NAME` configuration.

Custom Login using Request Loader

Sometimes you want to login users without using cookies, such as using header values or an api key passed as a query argument. In these cases, you should use the `request_loader` callback. This callback should behave the same as your `user_loader` callback, except that it accepts the Flask request instead of a user_id.

Fork me on GitHub

For example, to support login from both a url argument and from Basic Auth using the `Authorization` header:

```
@login_manager.request_loader
def load_user_from_request(request):

    # first, try to login using the api_key url arg
    api_key = request.args.get('api_key')
    if api_key:
        user = User.query.filter_by(api_key=api_key).first()
        if user:
            return user

    # next, try to login using Basic Auth
    api_key = request.headers.get('Authorization')
    if api_key:
        api_key = api_key.replace('Basic ', '', 1)
        try:
            api_key = base64.b64decode(api_key)
        except TypeError:
            pass
        user = User.query.filter_by(api_key=api_key).first()
        if user:
            return user

    # finally, return None if both methods did not login the user
    return None
```

Anonymous Users

By default, when a user is not actually logged in, `current_user` is set to an `AnonymousUserMixin` object. It has the following properties and methods:

- `is_active` and `is_authenticated` are `False`
- `is_anonymous` is `True`
- `get_id()` returns `None`

If you have custom requirements for anonymous users (for example, they need to have a permissions field), you can provide a callable (either a class or factory function) that creates anonymous users to the `LoginManager` with:

```
login_manager.anonymous_user = MyAnonymousUser
```

Remember Me

By default, when the user closes their browser the Flask Session is deleted and the user is logged out. “Remember Me” prevents the user from accidentally being logged out when they close their browser. This does **NOT** mean remembering or pr… the user’s username or password in a login form after the user has logged out.

v: latest

"Remember Me" functionality can be tricky to implement. However, Flask-Login makes it nearly transparent - just pass `remember=True` to the `login_user` call. A cookie will be saved on the user's computer, and then Flask-Login will automatically restore the user ID from that cookie if it is not in the session. The amount of time before the cookie expires can be set with the `REMEMBER_COOKIE_DURATION` configuration or it can be passed to `login_user`. The cookie is tamper-proof, so if the user tampers with it (i.e. inserts someone else's user ID in place of their own), the cookie will merely be rejected, as if it was not there.

That level of functionality is handled automatically. However, you can (and should, if your application handles any kind of sensitive data) provide additional infrastructure to increase the security of your remember cookies.

Alternative Tokens

Using the user ID as the value of the remember token means you must change the user's ID to invalidate their login sessions. One way to improve this is to use an alternative session token instead of the user's ID. For example:

```
@login_manager.user_loader
def load_user(session_token):
    return User.query.filter_by(session_token=session_token).first()
```

Then the `get_id` method of your User class would return the session token instead of the user's ID:

```
def get_id(self):
    return unicode(self.session_token)
```

This way you are free to change the user's session token to a new randomly generated value when the user changes their password, which would ensure their old authentication sessions will cease to be valid. Note that the session token must still uniquely identify the user... think of it as a second user ID.

Fresh Logins

When a user logs in, their session is marked as "fresh," which indicates that they actually authenticated on that session. When their session is destroyed and they are logged back in with a "remember me" cookie, it is marked as "non-fresh." `login_required` does not differentiate between freshness, which is fine for most pages. However, sensitive actions like changing one's personal information should require a fresh login. (Actions like changing one's password should always require a password re-entry regardless.)

v: latest

Fork me on GitHub

`fresh_login_required`, in addition to verifying that the user is logged in, will also ensure that their login is fresh. If not, it will send them to a page where they can re-enter their credentials. You can customize its behavior in the same ways as you can customize `login_required`, by setting `LoginManager.refresh_view`, `needs_refresh_message`, and `needs_refresh_message_category`:

```
login_manager.refresh_view = "accounts.reauthenticate"
login_manager.needs_refresh_message = (
    u"To protect your account, please reauthenticate to access this page."
)
login_manager.needs_refresh_message_category = "info"
```

Or by providing your own callback to handle refreshing:

```
@login_manager.needs_refresh_handler
def refresh():
    # do stuff
    return a_response
```

To mark a session as fresh again, call the `confirm_login` function.

Cookie Settings

The details of the cookie can be customized in the application settings.

REMEMBER_COOKIE_NAME	The name of the cookie to store the "remember me" information in. **Default:** `remember_token`
REMEMBER_COOKIE_DURATION	The amount of time before the cookie expires, as a `datetime.timedelta` object or integer seconds. **Default:** 365 days (1 non-leap Gregorian year)
REMEMBER_COOKIE_DOMAIN	If the "Remember Me" cookie should cross domains, set the domain value here (i.e. `.example.com` would allow the cookie to be used on all subdomains of `example.com`). **Default:** None
REMEMBER_COOKIE_PATH	Limits the "Remember Me" cookie to a certain path. **Default:** /
REMEMBER_COOKIE_SECURE	Restricts the "Remember Me" cookie's scope to secure channels (typically HTTPS). **Default:** None
REMEMBER_COOKIE_HTTPONLY	Prevents the "Remember Me" cookie from being accessed b client-side scripts. **Default:**

v: latest

REMEMBER_COOKIE_REFRESH_EACH_REQUEST	If set to True the cookie will be refreshed on every request, which bumps the lifetime. Works like Flask's SESSION_REFRESH_EACH_REQUEST. **Default:** False

Fork me on GitHub

Session Protection

While the features above help secure your "Remember Me" token from cookie thieves, the session cookie is still vulnerable. Flask-Login includes session protection to help prevent your users' sessions from being stolen.

You can configure session protection on the LoginManager, and in the app's configuration. If it is enabled, it can operate in either basic or strong mode. To set it on the LoginManager, set the session_protection attribute to "basic" or "strong":

```
login_manager.session_protection = "strong"
```

Or, to disable it:

```
login_manager.session_protection = None
```

By default, it is activated in "basic" mode. It can be disabled in the app's configuration by setting the SESSION_PROTECTION setting to None, "basic", or "strong".

When session protection is active, each request, it generates an identifier for the user's computer (basically, a secure hash of the IP address and user agent). If the session does not have an associated identifier, the one generated will be stored. If it has an identifier, and it matches the one generated, then the request is OK.

If the identifiers do not match in basic mode, or when the session is permanent, then the session will simply be marked as non-fresh, and anything requiring a fresh login will force the user to re-authenticate. (Of course, you must be already using fresh logins where appropriate for this to have an effect.)

If the identifiers do not match in strong mode for a non-permanent session, then the entire session (as well as the remember token if it exists) is deleted.

Disabling Session Cookie for APIs

When authenticating to APIs, you might want to disable setting the Flask Session cookie. To do this, use a custom session interface that skips saving the session depending on a flag you set on the request. For example:

v: latest

Fork me on GitHub

```
from flask import g
from flask.sessions import SecureCookieSessionInterface
from flask_login import user_loaded_from_header

class CustomSessionInterface(SecureCookieSessionInterface):
    """Prevent creating session from API requests."""
    def save_session(self, *args, **kwargs):
        if g.get('login_via_header'):
            return
        return super(CustomSessionInterface, self).save_session(*args,
                                                                 **kwargs)

app.session_interface = CustomSessionInterface()

@user_loaded_from_header.connect
def user_loaded_from_header(self, user=None):
    g.login_via_header = True
```

This prevents setting the Flask Session cookie whenever the user authenticated using your `header_loader`.

Localization

By default, the `LoginManager` uses `flash` to display messages when a user is required to log in. These messages are in English. If you require localization, set the `localize_callback` attribute of `LoginManager` to a function to be called with these messages before they're sent to `flash`, e.g. `gettext`. This function will be called with the message and its return value will be sent to `flash` instead.

API Documentation

This documentation is automatically generated from Flask-Login's source code.

Configuring Login

class flask_login.**LoginManager**(*app=None*, *add_context_processor=True*) [source]

This object is used to hold the settings used for logging in. Instances of `LoginManager` are *not* bound to specific apps, so you can create one in the main body of your code and then bind it to your app in a factory function.

setup_app(*app*, *add_context_processor=True*) [source]

This method has been deprecated. Please use `LoginManager.init_app()` instead.

unauthorized() [source]

This is called when the user is required to log in. If you register a cal… with `LoginManager.unauthorized_handler()`, then it will be called.

v: latest

Fork me on GitHub

Otherwise, it will take the following actions:

- Flash LoginManager.login_message to the user.
- If the app is using blueprints find the login view for the current blueprint using `blueprint_login_views`. If the app is not using blueprints or the login view for the current blueprint is not specified use the value of login_view.
- Redirect the user to the login view. (The page they were attempting to access will be passed in the `next` query string variable, so you can redirect there if present instead of the homepage. Alternatively, it will be added to the session as `next` if USE_SESSION_FOR_NEXT is set.)

If LoginManager.login_view is not defined, then it will simply raise a HTTP 401 (Unauthorized) error instead.

This should be returned from a view or before/after_request function, otherwise the redirect will have no effect.

needs_refresh() [source]

This is called when the user is logged in, but they need to be reauthenticated because their session is stale. If you register a callback with needs_refresh_handler, then it will be called. Otherwise, it will take the following actions:

- Flash LoginManager.needs_refresh_message to the user.
- Redirect the user to LoginManager.refresh_view. (The page they were attempting to access will be passed in the `next` query string variable, so you can redirect there if present instead of the homepage.)

If LoginManager.refresh_view is not defined, then it will simply raise a HTTP 401 (Unauthorized) error instead.

This should be returned from a view or before/after_request function, otherwise the redirect will have no effect.

General Configuration

user_loader(*callback*) [source]

This sets the callback for reloading a user from the session. The function you set should take a user ID (a `unicode`) and return a user object, or `None` if the user does not exist.

Parameters: **callback** (*callable*) – The callback for retrieving a u
object.

v: latest

header_loader(*callback*) [source]

This function has been deprecated. Please use `LoginManager.request_loader()` instead.

This sets the callback for loading a user from a header value. The function you set should take an authentication token and return a user object, or None if the user does not exist.

Parameters: **callback** (*callable*) – The callback for retrieving a user object.

anonymous_user

A class or factory function that produces an anonymous user, which is used when no one is logged in.

`unauthorized` Configuration

login_view

The name of the view to redirect to when the user needs to log in. (This can be an absolute URL as well, if your authentication machinery is external to your application.)

login_message

The message to flash when a user is redirected to the login page.

unauthorized_handler(*callback*) [source]

This will set the callback for the `unauthorized` method, which among other things is used by `login_required`. It takes no arguments, and should return a response to be sent to the user instead of their normal view.

Parameters: **callback** (*callable*) – The callback for unauthorized users.

`needs_refresh` Configuration

refresh_view

The name of the view to redirect to when the user needs to reauthenticate.

needs_refresh_message

The message to flash when a user is redirected to the reauthentication page.

needs_refresh_handler(*callback*) [source]

This will set the callback for the `needs_refresh` method, which among other things is used by `fresh_login_required`. It takes no arguments, and should return a response to be sent to the user instead of their normal view

v: latest

Parameters: **callback** (*callable*) – The callback for unauthorized users.

Login Mechanisms

flask_login.**current_user**

A proxy for the current user.

flask_login.**login_fresh**() [source]

This returns `True` if the current login is fresh.

flask_login.**login_user**(*user*, *remember=False*, *duration=None*, *force=False*, *fresh=True*) [source]

Logs a user in. You should pass the actual user object to this. If the user's `is_active` property is `False`, they will not be logged in unless `force` is `True`.

This will return `True` if the log in attempt succeeds, and `False` if it fails (i.e. because the user is inactive).

Parameters:
- **user** (*object*) – The user object to log in.
- **remember** (*bool*) – Whether to remember the user after their session expires. Defaults to `False`.
- **duration** (`datetime.timedelta`) – The amount of time before the remember cookie expires. If `None` the value set in the settings is used. Defaults to `None`.
- **force** (*bool*) – If the user is inactive, setting this to `True` will log them in regardless. Defaults to `False`.
- **fresh** (*bool*) – setting this to `False` will log in the user with a session marked as not "fresh". Defaults to `True`.

flask_login.**logout_user**() [source]

Logs a user out. (You do not need to pass the actual user.) This will also clean up the remember me cookie if it exists.

flask_login.**confirm_login**() [source]

This sets the current session as fresh. Sessions become stale when they are reloaded from a cookie.

Protecting Views

flask_login.**login_required**(*func*) [source]

If you decorate a view with this, it will ensure that the current user is logged in and authenticated before calling the actual view. (If they are not, it calls the `LoginManager.unauthorized` callback.) For example:

```
@app.route('/post')
@login_required
```

v: latest

Fork me on GitHub

```
def post():
    pass
```

If there are only certain times you need to require that your user is logged in, you can do so with:

```
if not current_user.is_authenticated:
    return current_app.login_manager.unauthorized()
```

...which is essentially the code that this function adds to your views.

It can be convenient to globally turn off authentication when unit testing. To enable this, if the application configuration variable `LOGIN_DISABLED` is set to `True`, this decorator will be ignored.

> Note:
> Per W3 guidelines for CORS preflight requests, HTTP `OPTIONS` requests are exempt from login checks.

Parameters: **func** (*function*) – The view function to decorate.

`flask_login.` **`fresh_login_required`**(*func*) [source]

If you decorate a view with this, it will ensure that the current user's login is fresh - i.e. their session was not restored from a 'remember me' cookie. Sensitive operations, like changing a password or e-mail, should be protected with this, to impede the efforts of cookie thieves.

If the user is not authenticated, `LoginManager.unauthorized()` is called as normal. If they are authenticated, but their session is not fresh, it will call `LoginManager.needs_refresh()` instead. (In that case, you will need to provide a `LoginManager.refresh_view`.)

Behaves identically to the `login_required()` decorator with respect to configutation variables.

> Note:
> Per W3 guidelines for CORS preflight requests, HTTP `OPTIONS` requests are exempt from login checks.

Parameters: **func** (*function*) – The view function to decorate.

User Object Helpers

class `flask_login.` **`UserMixin`** [source]

v: latest

This provides default implementations for the methods that Flask-Login expects user objects to have.

class flask_login. **AnonymousUserMixin** [source]

This is the default object for representing an anonymous user.

Utilities

flask_login. **login_url**(*login_view*, *next_url=None*, *next_field='next'*) [source]

Creates a URL for redirecting to a login page. If only `login_view` is provided, this will just return the URL for it. If `next_url` is provided, however, this will append a `next=URL` parameter to the query string so that the login view can redirect back to that URL. Flask-Login's default unauthorized handler uses this function when redirecting to your login url. To force the host name used, set `FORCE_HOST_FOR_REDIRECTS` to a host. This prevents from redirecting to external sites if request headers Host or X-Forwarded-For are present.

Parameters:
- **login_view** (*str*) – The name of the login view. (Alternately, the actual URL to the login view.)
- **next_url** (*str*) – The URL to give the login view for redirection.
- **next_field** (*str*) – What field to store the next URL in. (It defaults to `next`.)

Signals

See the Flask documentation on signals for information on how to use these signals in your code.

flask_login. **user_logged_in**

Sent when a user is logged in. In addition to the app (which is the sender), it is passed `user`, which is the user being logged in.

flask_login. **user_logged_out**

Sent when a user is logged out. In addition to the app (which is the sender), it is passed `user`, which is the user being logged out.

flask_login. **user_login_confirmed**

Sent when a user's login is confirmed, marking it as fresh. (It is not called for a normal login.) It receives no additional arguments besides the app.

flask_login. **user_unauthorized**

Sent when the `unauthorized` method is called on a `LoginManager`. It receives no additional arguments besides the app.

v: latest

Fork me on GitHub

flask_login.**user_needs_refresh**

Sent when the `needs_refresh` method is called on a `LoginManager`. It receives no additional arguments besides the app.

flask_login.**session_protected**

Sent whenever session protection takes effect, and a session is either marked non-fresh or deleted. It receives no additional arguments besides the app.

v: latest