CODING NOMADS

12) Users and User Authentication Lesson

Using Flask-Login

14 min to complete · By Brandon Gigous

Contents

Note: This lesson builds on a project that's been gradually worked on throughout this course.

Once you get your authentication blueprint in place, it's time to get your `User` model even more ready for authenticating users. This lesson will teach you about the Flask-Login module and how it helps you with managing users.

Flask-Login

Flask-Login is a lightweight Flask extension that makes working with authentication with respect to the user session a piece of cake.

Pictured is the unit of effort required to use Flask-Login to manage user logins

What Flask-Login does in technical speak is it records and updates, as needed, the user's authenticated state in the user session. The user session was talked about in Section 8 on forms, in terms of how information recorded in forms is remembered. Updating the user session with the user's state of authentication means you can know when the user is signed in or not, which you can use to give access to certain pages, or not. There are of course other extensions to manage the user session when it comes to authentication, but Flask-Login is probably the most simple.

To install it, it's another simple pip:

```
(env) $ pip install flask-login
```

User Model Prep for Authentication

It's a good thing you have a user model, because Flask-Login needs one of those in order to help with the user session. But as your `User` model exists now, it's not enough for Flask-Login to work well. The good news is you'll only need a few properties and a function to get everything ready:

- `is_authenticated` : Must be `True` if the user has valid login credentials and is currently logged in, otherwise `False` .
- `is_active` : Must be `True` if the user is allowed to log in or `False` if not. Basically it means that if an account is not disabled, `is_active` will return `True` .
- `is_anonymous` : This must always be `False` for regular users who login and is only `True` for any user that is not logged in
- `get_id()` : This function returns a unique identifer for user

And now for the fun part: Mixins! While those brownie mixes you get might as well be called "mixins" themselves, this is not that. Flask-Login gives you a very convenient class to help you with authenticated-or-not users. You can "tack on" this `UserMixin` class to the `User` model. Once you do, you have all those properties and the `get_id()` function handled for you!

Using `UserMixin` In The User Model

To use the `UserMixin` in the `User` Model, all you need to do is add the `UserMixin` as a base class:

```
from flask_login import UserMixin

class User(UserMixin, db.Model):
    # ...
```

That's it! You're all done, and all of those properties and function above are implemented for you.

Now that you got your user mixin mixed in and learned about Flask-Login, now it's time to use it. Next, you'll learn about the Flask-Login `LoginManager`, which will use your user mixin.

Using Flask-Login LoginManager

To get Flask-Login "initialized" is a little different from other plugins. The reason is that Flask-Login doesn't need to be initialized, but your application will need an instance of its `LoginManager` class in order to manage logins. I'm sure you can tell what it does by the name, but to be clear it's a class that loads users and can direct users to the login page if they need to log in.

To initialize your own `LoginManager`, you will create it in your `app/__init__.py` and initialize it in the application factory like you do with other extensions:

```
login_manager = LoginManager()
login_manager.login_view = 'auth.login'

def create_app(config_name):
    # ...
    login_manager.init_app(app)
    # ...
```

An instance of the `LoginManager` is created, then the `login_view` attribute is set. This property is specified as an endpoint that the `LoginManager` will direct a user to if the user tries to access a protected page. In other words, if you have a page that only registered members of the site can see and an anonymous user tries to access it, a redirect will be triggered that redirects, in this case, to the `login()` view function you made earlier.

User Loader

One more thing you'll need to get set up is the user loader. In fact, this is just a decorator on the Flask-Login side. You'll need to decorate a function that will, given a user identifier, return a `UserMixin` that corresponds to that identifier. Put more simply in your case, this function needs to return a `User` with the `id` that matches what is passed in. To get that you user, where else would you go but the database? Check it out:

```
#! Put this in app/models.py
from . import login_manager

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

Taking the `login_manager` from before, you use its `user_loader` decorator to decorate the function that contains your custom user loader. Pretty neat, huh? By the way, it doesn't matter *how* you get the user, Flask-Login is agnostic in this way. You could also, theoretically, get your user from an LDAP server or from your uncle Steve, as long as you give the correct one based on the identifer passed in. The `user_id` is actually a string, so it needs to be converted to an integer before being queried from the database.

What might remain unclear is: where does this identifer *come from*? Who calls `load_user()` ? The answer to the latter is Flask-Login, and for the former, it gets the identifer from whatever user is logged in, if there is one. If a user logs in and requests to see another page, the `LoginManager` will call the `load_user()` function since that's what was decorated because it needs information about the user with that identifier.

Protecting Routes

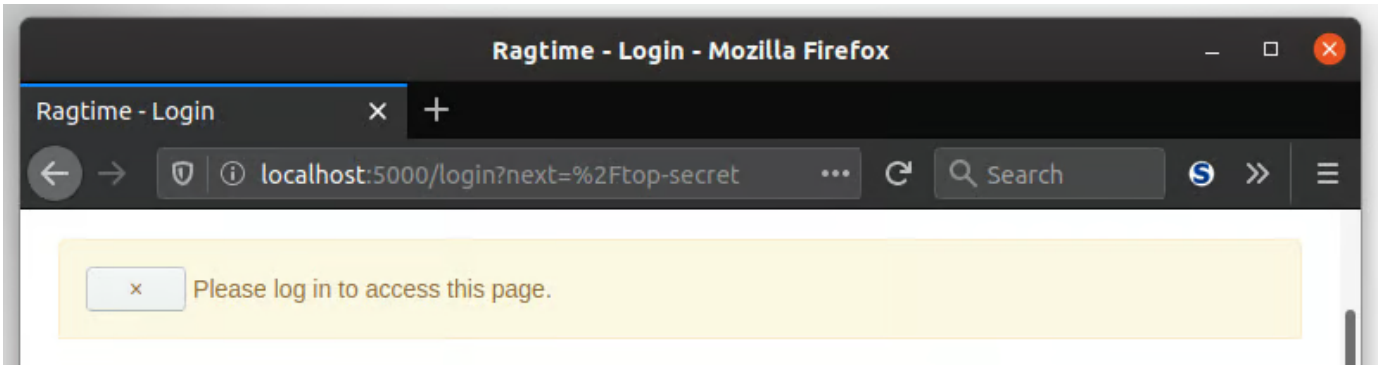
Another cool feature of Flask-Login is the `login_required` decorator. This decorator can be applied to any view function that you want protected from unauthenticated or anonymous users. Let's say you have a view function called `top_secret()` that only the coolest kids can see, those who are registered with your super cool but VIP-only-accessible website. You can decorate it like so:

```
from flask_login import login_required

@main.route('/top-secret')
@login_required
def top_secret():
    return "Welcome, VIP member! "\
```

```
            "You have exclusive access to this page. "\
            "Now let me get your cocktail."
```

Whenever an unauthenticated user tries to access this VIP page, Flask-Login will send them instead to the login page specified by the `LoginManager.login_view` property.



But if they are logged in, then they can see the secret content:

Now how do you know which decorator to apply first? That's two decorators! Is it by which decorator is prettiest? No, although that would be an interesting metric in a programming language. Think of a decorator as decorating whatever is under it. In the example above, `login_required` decorates `top_secret()`. The `route` decorator then decorates *that* decoration. Ultimately, you want your `top_secret()` view function to *be* a view function, so the `route` decorator should be topmost. The opposite would give the wrong result.

In the next lesson, you'll get even more practice with making forms as you'll be tasked with creating your very own login form.

Summary: Using Flask-Login and LoginManager for User Authentication

- Flask-Login is a lightweight Flask extension, it records and updates, as needed, the user's authenticated state in the user session.
- You'll only need a few properties and a function for your User model to get everything ready.
- To use the `UserMixin` in the `User` Model, all you need to do is add the `UserMixin` as a base class:

```
from flask_login import UserMixin

class User(UserMixin, db.Model):
    # ...
```

- To initialize your own `LoginManager` , you will create it in your `app/__init__.py` and initialize it in the application factory like you do with other extensions:

```
login_manager = LoginManager()
login_manager.login_view = 'auth.login'

def create_app(config_name):
    # ...
    login_manager.init_app(app)
    # ...
```

- One more thing you'll need to get set up is the user loader.

```
#! Put this in app/models.py
from . import login_manager

@login_manager.user_loader
```

```
def load_user(user_id):
    return User.query.get(int(user_id))
```

- Another feature of Flask-Login is the `login_required` decorator. This can be applied to any view function that you want protected from unauthenticated or anonymous users.

Previous Next → Video: Flask-Login and Login Manager