Previous Next

Authentication using Flask-Login

Estimated Time: 2-3 hours

In the previous three assignments you built a blogging engine. But there is one obvious problem that remains: anyone can edit your blog. In order to stop this you need to add a system which only allows users to edit your blog when they are authenticated.

In this assignment you will be building a login system based on the Flask-Login module which will allow you to limit the people who are able to make changes to the blog.

Setting a secret key

The first step you need to take is to set up a secret key in `config.py`:

```
class DevelopmentConfig(object):
    SQLALCHEMY_DATABASE_URI = "postgresql://ubuntu:thinkful@localhost:
    DEBUG = True
    SECRET_KEY = os.environ.get("BLOGFUL_SECRET_KEY", os.urandom(12))
```

The `SECRET_KEY` is used to cryptographically secure your application's sessions. However, it's not a good idea to store your application's secret key inside the application configuration itself. Therefore, we use `os.environ.get`

> to obtain the secret key from an environment variable, falling back on a key generated at random on startup.

An important part of your development process should be to export any needed environment variables. In your terminal prompt, enter the following:

```
export BLOGFUL_SECRET_KEY="your_secret_key_here"
```

This sets an environment variable called `BLOGFUL_SECRET_KEY`. If you don't do this, the random key will be used, which means the key will change every time the server restarts - which, in turn, means that all your users' sessions will be wiped out. Also, if you have multiple server processes (common on heavily-used web sites), they'll all have different keys, so your users will find themselves getting logged out randomly. For a production web site, this is unacceptable, and setting the environment variable will be critical; for now, it's not as important, but we do it as best-practice.

Adding a user

The next stage of the login system is to create a user model. In `database.py` add the following model:

```
from flask_login import UserMixin

class User(Base, UserMixin):
    __tablename__ = "users"

    id = Column(Integer, primary_key=True)
    name = Column(String(128))
    email = Column(String(128), unique=True)
    password = Column(String(128))
```

Here you create the `User` model which inherits from the declarative base. It also inherits from Flask-Login's `UserMixin` class, which adds a series of

default methods. Flask-Login relies on these default methods to make authentication work. The model has four columns: an integer id, a name, a unique email address which you will use to identify the user, and a password.

Now that you have the model you want to add a user to the database which you can use to test that the login system is working. Add a command to the `manage.py` script which will add a new user:

```
from getpass import getpass

from werkzeug.security import generate_password_hash

from blog.database import User

@manager.command
def adduser():
    name = input("Name: ")
    email = input("Email: ")
    if session.query(User).filter_by(email=email).first():
        print("User with that email address already exists")
        return

    password = ""
    while len(password) < 8 or password != password_2:
        password = getpass("Password: ")
        password_2 = getpass("Re-enter password: ")
    user = User(name=name, email=email,
                password=generate_password_hash(password))
    session.add(user)
    session.commit()
```

Here you ask the user for their name, email address, and their password twice. You check to make sure that the user is not already stored in the database, and you make sure that the passwords match. Then you create the user object and add it to the database.

Notice how you use the `generate_password_hash` function from Werkzeug in order to *hash* the password. Hashing is a process which converts the plain text password to a string of characters, for example the string baseball is

converted to the hash a2c901c8c6dea98958c219f6f2d038c44dc5d362 using the SHA1 hashing algorithm.

For passwords you'll use *one-way hashes*. This means that the hashing process only works in one direction; you can always obtain the string a2c901... from the string baseball, but it is practically impossible to obtain the string baseball if you only know the hash a2c901... The benefit of hashing passwords is that if a malicious user obtains access to your database then they only know the user's hashes, and cannot obtain the passwords easily. Additionally, these are *salted hashes*, which means that some additional random data is mixed in with the password; generating several hashes for the same password will generate several distinct hashes. The verification step uses the salt and a provided password to ascertain whether or not the password is correct, without revealing anything about a wrong password.

Setting up the login system

Now that you have a user added, you can start to set up the login system. First of all you need to perform a little bit of configuration. In a new file called *login.py* add the following code:

```
from flask_login import LoginManager

from . import app
from .database import session, User

login_manager = LoginManager()
login_manager.init_app(app)

login_manager.login_view = "login_get"
login_manager.login_message_category = "danger"

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

Here you do a couple of things. First you create an instance of the `LoginManager` class from Flask-Login, and initialize it. Next you set a couple of attributes of the object. The `login_view` is the name of the view which an unauthorized user will be redirected to when they try to access a protected resource. The `login_message_category` is a category used to classify any error messages from Flask-Login. You will use this in conjunction with Bootstrap's alerts system to give the user information about the login process.

Finally you create a function which tells Flask-Login how to access an object representing a user via their ID.

The only thing left to do now is to incorporate all of this new code into the application. In `__init__.py`, add the following line to the bottom:

```
from . import login
```

Next, create the `login_get` view which we referred to earlier. In *views.py* add this view:

```
@app.route("/login", methods=["GET"])
def login_get():
    return render_template("login.html")
```

Then, create the `login.html` template which accompanies it:

```
{% extends "base.html" %}
{% block content %}
<form role="form" method="POST">
    <div class="form-group">
        <label for="email">Email address</label>
        <input type="email" class="form-control" id="email" name="emai
    </div>
    <div class="form-group">
        <label for="password">Password</label>
```

```
            <input type="password" class="form-control" id="password" name
        </div>
        <button type="submit" class="btn btn-default">Submit</button>
    </form>
    {% endblock %}
```

Try visiting the `/login` page. You should see the login form, and when you submit you should see that there is no POST route for /login. Now, add that route so you can authenticate yourself:

```
from flask import flash
from flask_login import login_user
from werkzeug.security import check_password_hash
from .database import User

@app.route("/login", methods=["POST"])
def login_post():
    email = request.form["email"]
    password = request.form["password"]
    user = session.query(User).filter_by(email=email).first()
    if not user or not check_password_hash(user.password, password):
        flash("Incorrect username or password", "danger")
        return redirect(url_for("login_get"))

    login_user(user)
    return redirect(request.args.get('next') or url_for("entries"))
```

Here you read the email address and password which the user entered from the `request.form` dictionary. Next you query to find the user object with the matching email address. You check that the user exists, and use Werkzeug's `check_password_hash` function to compare the password the user entered with the hash stored in the database.

If the username or password is incorrect you use Flask's `flash` function to store a message which you can use when you render the next page. In the next section you will look at how to display these messages to the user. You then redirect the user back to the login page.

If the username and password are correct then you call Flask-Login's `login_user` function. This sends a *cookie* (a small chunk of data) to the user's browser which is used to identify the user. When the user then tries to access a protected resource, Flask-Login will make sure that they have the cookie set and are allowed to access the resource.

Finally when the user is logged in you redirect the user. Normally you redirect the user to the entries page. However if there is a `next` parameter in the URL's *query string* then you redirect to that address. Flask-Login uses this so that the user can access the intended resource after logging in.

For example imagine that the `/entry/add` resource requires us to login. When you visit `/entry/add` Flask-Login will redirect you to `/login?next=/entry/add`. You then log in, and the view will read the `next` parameter and send us on to `/entry/add`, complying with the initial request.

Try visiting the `/login` page again, and log in with an incorrect password. You should be sent back to the login page. Now try entering the correct password. You should be logged in and redirected to the entries page.

Handling error messages

In the `login_user` view you used the `flash` function to send a message to the user if they entered the wrong username or password. So now let's add some code to the base template which will show these messages. In *base.html* add the following code below the navbar:

```
{% with messages = get_flashed_messages(with_categories=true) %}
{% if messages %}
{% for category, message in messages %}
<div class="alert alert-{{ category }}">
    {{ message }}
</div>
{% endfor %}
```

```
{% endif %}
{% endwith %}
```

Here you call the `get_flashed_messages` function, telling it to also give categories with the messages. The categories allow us to classify the severity of messages; you might remember that earlier you told Flask-Login to give you messages with the category set to `danger`. You then loop through the messages, creating a new alert for each one.

Try logging in to the site again using incorrect information. You should see the flashed message as an alert at the top of the page.

Protecting resources

The final stage of the login system is to actually protect the resources which require authentication. This is done using the `login_required` decorator from Flask-Login. So to prevent unauthorized users from adding entries you can edit the views to include the decorator:

```
from flask_login import login_required

@app.route("/entry/add", methods=["GET"])
@login_required
def add_entry_get():
    return render_template("add_entry.html")

@app.route("/entry/add", methods=["POST"])
@login_required
def add_entry_post():
    entry = Entry(
        title=request.form["title"],
        content=request.form["content"],
    )
    session.add(entry)
    session.commit()
    return redirect(url_for("entries"))
```

Notice how you decorate both the GET and the POST views to prevent access to the form, and to stop a user from manually constructing the form data and sending it to the POST endpoint.

Try visiting the `/entry/add` page whilst not logged in. You should find that you are redirected to the login page. Then when you log in you should be redirected to the add entry form, and be able to post content. Your login system is complete.

What does the UserMixin class from Flask-Login contain? How do we use it?

> *The UserMixin class contains a series of methods which Flask-Login uses to check whether a user should be authenticated. Generally you inherit from it to add these methods to a user class.*

What do we mean when we talk about creating a one-way hash?

> *A hash is a value which is generated from a input using a hashing algorithm. Hashing an input should always give the same resulting value. One-way hashes have the property that whilst it is simple to generate the hash from the input, it is practically impossible to recreate the input from the hash.*

How can you use Werkzeug to check whether a correct password has been entered when using password hashing?

> *By calling* `check_password_hash(hashed_password, user_password)`*, where* `hashed_password` *is a stored password hash, and* `user_password` *is the password enterred by the user.*

What is Flask-Login's `user_loader` decorator used for?

> *It specifies a function which will return the user object for a corresponding user ID.*

What role do cookies play in Flask-Login?

> *They store data which is used to identify the user as the navigate through the site.*

What does the `flash` function do? How do we work with it within a template?

> *The flash function stores a message which we can access when rendering later pages. We can use the `get_flashed_messages` function in the template to access the list of messages.*

How do we make a view only accessible to a logged in user?

> *Decorate the view function with the `@login_required` decorator.*

☆ ☆ ☆ ☆ ☆ · Report a typo or other issue

✔ Mark as completed

Previous Next