Blueprints and Views

A view function is the code you write to respond to requests to your application. Flask uses patterns to match the incoming request URL to the view that should handle it. The view returns data that Flask turns into an outgoing response. Flask can also go the other direction and generate a URL to a view based on its name and arguments.

Create a Blueprint

A **`Blueprint`** is a way to organize a group of related views and other code. Rather than registering views and other code directly with an application, they are registered with a blueprint. Then the blueprint is registered with the application when it is available in the factory function.

Flaskr will have two blueprints, one for authentication functions and one for the blog posts functions. The code for each blueprint will go in a separate module. Since the blog needs to know about authentication, you'll write the authentication one first.

`flaskr/auth.py`

```
import functools

from flask import (
    Blueprint, flash, g, redirect, render_template, request, session, url_
)
from werkzeug.security import check_password_hash, generate_password_hash

from flaskr.db import get_db

bp = Blueprint('auth', __name__, url_prefix='/auth')
```

This creates a **`Blueprint`** named `'auth'`. Like the application object, the blueprint needs to know where it's defined, so `__name__` is passed as the second argument. The `url_prefix` will be prepended to all the URLs associated with the blueprint.

Import and register the blueprint from the factory using **`app.register_blueprint()`**. Place the new code at the end of the factory function before returning the app.

`flaskr/__init__.py`

```
def create_app():
    app = ...
    # existing code omitted
    from . import auth
    app.register_blueprint(auth.bp)

    return app
```

The authentication blueprint will have views to register new users and to log in and log out.

The First View: Register

When the user visits the `/auth/register` URL, the `register` view will return HTML with a form for them to fill out. When they submit the form, it will validate their input and either show the form again with an error message or create the new user and go to the login page.

For now you will just write the view code. On the next page, you'll write templates to generate the HTML form.

`flaskr/auth.py`

```
@bp.route('/register', methods=('GET', 'POST'))
def register():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']
        db = get_db()
        error = None

        if not username:
            error = 'Username is required.'
        elif not password:
            error = 'Password is required.'
        elif db.execute(
            'SELECT id FROM user WHERE username = ?', (username,)
        ).fetchone() is not None:
            error = 'User {} is already registered.'.format(username)

        if error is None:
            db.execute(
                'INSERT INTO user (username, password) VALUES (?, ?)',
                (username, generate_password_hash(password))
            )
            db.commit()
            return redirect(url_for('auth.login'))

        flash(error)

    return render_template('auth/register.html')
```

Here's what the `register` view function is doing:

1. **`@bp.route`** associates the URL `/register` with the `register` view function. When Flask receives a request to `/auth/register`, it will call the `register` view and use the return value as the response.

2. If the user submitted the form, **`request.method`** will be `'POST'`. In this case, start validating the input.

3. **`request.form`** is a special type of **`dict`** mapping submitted form keys and values. The user will input their `username` and `password`.

4. Validate that `username` and `password` are not empty.

5. Validate that `username` is not already registered by querying the database and checking if a result is returned. **`db.execute`** takes a SQL query with `?` placeholders for any user input, and a tuple of values to replace the placeholders with. The database library will take care of escaping the values so you are not vulnerable to a *SQL injection attack*.

   `fetchone()` returns one row from the query. If the query returned no results, it returns `None`. Later, **`fetchall()`** is used, which returns a list of all results.

6. If validation succeeds, insert the new user data into the database. For security, passwords should never be stored in the database directly. Instead, **`generate_password_hash()`** is used to securely hash the password, and that hash is stored. Since this query modifies data, **`db.commit()`** needs to be called afterwards to save the changes.

7. After storing the user, they are redirected to the login page. **`url_for()`** generates the URL for the login view based on its name. This is preferable to writing the URL directly as it allows you to change the URL later without changing all code that links to it. **`redirect()`** generates a redirect response to the generated URL.

8. If validation fails, the error is shown to the user. **`flash()`** stores messages that can be retrieved when rendering the template.

9. When the user initially navigates to `auth/register`, or there was an validation error, an HTML page with the registration form should be shown. **`render_template()`** will render a template containing the HTML, which you'll write in the next step of the tutorial.

Login

This view follows the same pattern as the `register` view above.

`flaskr/auth.py`

```
@bp.route('/login', methods=('GET', 'POST'))
def login():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']
        db = get_db()
        error = None
        user = db.execute(
            'SELECT * FROM user WHERE username = ?', (username,)
        ).fetchone()

        if user is None:
            error = 'Incorrect username.'
        elif not check_password_hash(user['password'], password):
            error = 'Incorrect password.'

        if error is None:
            session.clear()
            session['user_id'] = user['id']
            return redirect(url_for('index'))

        flash(error)

    return render_template('auth/login.html')
```

There are a few differences from the `register` view:

1. The user is queried first and stored in a variable for later use.
2. **`check_password_hash()`** hashes the submitted password in the same way as the stored hash and securely compares them. If they match, the password is valid.
3. **`session`** is a **`dict`** that stores data across requests. When validation succeeds, the user's `id` is stored in a new session. The data is stored in a *cookie* that is sent to the browser, and the browser then sends it back with subsequent requests. Flask securely *signs* the data so that it can't be tampered with.

Now that the user's `id` is stored in the **`session`**, it will be available on subsequent requests. At the beginning of each request, if a user is logged in their information should be loaded and made available to other views.

`flaskr/auth.py`

```
@bp.before_app_request
def load_logged_in_user():
    user_id = session.get('user_id')
    if user_id is None:
        g.user = None
    else:
        g.user = get_db().execute(
            'SELECT * FROM user WHERE id = ?', (user_id,)
        ).fetchone()
```

`bp.before_app_request()` registers a function that runs before the view function, no matter what URL is requested. `load_logged_in_user` checks if a user id is stored in the `session` and gets that user's data from the database, storing it on `g.user`, which lasts for the length of the request. If there is no user id, or if the id doesn't exist, `g.user` will be `None`.

Logout

To log out, you need to remove the user id from the `session`. Then `load_logged_in_user` won't load a user on subsequent requests.

flaskr/auth.py

```
@bp.route('/logout')
def logout():
    session.clear()
    return redirect(url_for('index'))
```

Require Authentication in Other Views

Creating, editing, and deleting blog posts will require a user to be logged in. A *decorator* can be used to check this for each view it's applied to.

flaskr/auth.py

```
def login_required(view):
    @functools.wraps(view)
    def wrapped_view(**kwargs):
        if g.user is None:
            return redirect(url_for('auth.login'))

        return view(**kwargs)

    return wrapped_view
```

This decorator returns a new view function that wraps the original view it's applied to. The new function checks if a user is loaded and redirects to the login page otherwise. If a user is loaded the original view is called and continues normally. You'll use this decorator when writing the blog views.

Endpoints and URLs

The **`url_for()`** function generates the URL to a view based on a name and arguments. The name associated with a view is also called the *endpoint*, and by default it's the same as the name of the view function.

For example, the `hello()` view that was added to the app factory earlier in the tutorial has the name `'hello'` and can be linked to with `url_for('hello')`. If it took an argument, which you'll see later, it would be linked to using `url_for('hello', who='World')`.

When using a blueprint, the name of the blueprint is prepended to the name of the function, so the endpoint for the `login` function you wrote above is `'auth.login'` because you added it to the `'auth'` blueprint.

Continue to Templates.