Blog Blueprint

You’ll use the same techniques you learned about when writing the authentication blueprint to write the blog blueprint. The blog should list all posts, allow logged in users to create posts, and allow the author of a post to edit or delete it.

As you implement each view, keep the development server running. As you save your changes, try going to the URL in your browser and testing them out.

The Blueprint

Define the blueprint and register it in the application factory.

`flaskr/blog.py`

```
from flask import (
    Blueprint, flash, g, redirect, render_template, request, url_for
)
from werkzeug.exceptions import abort

from flaskr.auth import login_required
from flaskr.db import get_db

bp = Blueprint('blog', __name__)
```

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 blog
    app.register_blueprint(blog.bp)
    app.add_url_rule('/', endpoint='index')

    return app
```

Unlike the auth blueprint, the blog blueprint does not have a `url_prefix`. So the `index` view will be at `/`, the `create` view at `/create`, and so on. The blog is the main feature of Flaskr, so it makes sense that the blog index will be the main index.

However, the endpoint for the `index` view defined below will be `blog.index`. Some of the authentication views referred to a plain `index` endpoint. **`app.add_url_rule()`** associates the endpoint name `'index'` with the `/` url so that `url_for('index')` or `url_for('blog.index')` will both work, generating the same `/` URL either way.

In another application you might give the blog blueprint a `url_prefix` and define a separate `index` view in the application factory, similar to the `hello` view. Then the `index` and `blog.index` endpoints and URLs would be different.

Index

The index will show all of the posts, most recent first. A `JOIN` is used so that the author information from the `user` table is available in the result.

flaskr/blog.py

```
@bp.route('/')
def index():
    db = get_db()
    posts = db.execute(
        'SELECT p.id, title, body, created, author_id, username'
        ' FROM post p JOIN user u ON p.author_id = u.id'
        ' ORDER BY created DESC'
    ).fetchall()
    return render_template('blog/index.html', posts=posts)
```

flaskr/templates/blog/index.html

```
{% extends 'base.html' %}

{% block header %}
  <h1>{% block title %}Posts{% endblock %}</h1>
  {% if g.user %}
    <a class="action" href="{{ url_for('blog.create') }}">New</a>
  {% endif %}
{% endblock %}

{% block content %}
  {% for post in posts %}
    <article class="post">
      <header>
        <div>
          <h1>{{ post['title'] }}</h1>
          <div class="about">by {{ post['username'] }} on {{ post['crea
        </div>
        {% if g.user['id'] == post['author_id'] %}
```

```
          <a class="action" href="{{ url_for('blog.update', id=post['id
        {% endif %}
      </header>
      <p class="body">{{ post['body'] }}</p>
    </article>
    {% if not loop.last %}
      <hr>
    {% endif %}
  {% endfor %}
{% endblock %}
```

When a user is logged in, the `header` block adds a link to the `create` view. When the user is the author of a post, they'll see an "Edit" link to the `update` view for that post. `loop.last` is a special variable available inside Jinja for loops. It's used to display a line after each post except the last one, to visually separate them.

Create

The `create` view works the same as the auth `register` view. Either the form is displayed, or the posted data is validated and the post is added to the database or an error is shown.

The `login_required` decorator you wrote earlier is used on the blog views. A user must be logged in to visit these views, otherwise they will be redirected to the login page.

flaskr/blog.py

```
@bp.route('/create', methods=('GET', 'POST'))
@login_required
def create():
    if request.method == 'POST':
        title = request.form['title']
        body = request.form['body']
        error = None

        if not title:
            error = 'Title is required.'

        if error is not None:
            flash(error)
        else:
            db = get_db()
            db.execute(
                'INSERT INTO post (title, body, author_id)'
                ' VALUES (?, ?, ?)',
                (title, body, g.user['id'])
            )
            db.commit()
            return redirect(url_for('blog.index'))

    return render_template('blog/create.html')
```

flaskr/templates/blog/create.html

```
{% extends 'base.html' %}

{% block header %}
  <h1>{% block title %}New Post{% endblock %}</h1>
{% endblock %}

{% block content %}
  <form method="post">
    <label for="title">Title</label>
    <input name="title" id="title" value="{{ request.form['title'] }}"
    <label for="body">Body</label>
    <textarea name="body" id="body">{{ request.form['body'] }}</textare
    <input type="submit" value="Save">
  </form>
{% endblock %}
```

Update

Both the `update` and `delete` views will need to fetch a `post` by `id` and check if the author matches the logged in user. To avoid duplicating code, you can write a function to get the `post` and call it from each view.

flaskr/blog.py

```
def get_post(id, check_author=True):
    post = get_db().execute(
        'SELECT p.id, title, body, created, author_id, username'
        ' FROM post p JOIN user u ON p.author_id = u.id'
        ' WHERE p.id = ?',
        (id,)
    ).fetchone()

    if post is None:
        abort(404, "Post id {0} doesn't exist.".format(id))

    if check_author and post['author_id'] != g.user['id']:
        abort(403)
    return post
```

`abort()` will raise a special exception that returns an HTTP status code. It takes an optional message to show with the error, otherwise a default message is used. `404` means "Not Found", and `403` means "Forbidden". (`401` means "Unauthorized", but you redirect to the login page instead of returning that status.)

The `check_author` argument is defined so that the function can be used to get a `post` without checking the author. This would be useful if you wrote a view to show an individual post on a page, where the user doesn't matter because they're not modifying the post.

flaskr/blog.py

```
@bp.route('/<int:id>/update', methods=('GET', 'POST'))
@login_required
def update(id):
    post = get_post(id)

    if request.method == 'POST':
        title = request.form['title']
        body = request.form['body']
        error = None

        if not title:
            error = 'Title is required.'

        if error is not None:
            flash(error)
        else:
            db = get_db()
            db.execute(
                'UPDATE post SET title = ?, body = ?'
                ' WHERE id = ?',
                (title, body, id)
            )
            db.commit()
            return redirect(url_for('blog.index'))

    return render_template('blog/update.html', post=post)
```

Unlike the views you've written so far, the `update` function takes an argument, `id`. That corresponds to the `<int:id>` in the route. A real URL will look like `/1/update`. Flask will capture the `1`, ensure it's an **`int`**, and pass it as the `id` argument. If you don't specify `int:` and instead do `<id>`, it will be a string. To generate a URL to the update page, **`url_for()`**

needs to be passed the `id` so it knows what to fill in: `url_for('blog.update', id=post['id'])`. This is also in the `index.html` file above.

The `create` and `update` views look very similar. The main difference is that the `update` view uses a `post` object and an `UPDATE` query instead of an `INSERT`. With some clever refactoring, you could use one view and template for both actions, but for the tutorial it's clearer to keep them separate.

`flaskr/templates/blog/update.html`

```
{% extends 'base.html' %}

{% block header %}
  <h1>{% block title %}Edit "{{ post['title'] }}"{% endblock %}</h1>
{% endblock %}

{% block content %}
  <form method="post">
    <label for="title">Title</label>
    <input name="title" id="title"
      value="{{ request.form['title'] or post['title'] }}" required>
    <label for="body">Body</label>
    <textarea name="body" id="body">{{ request.form['body'] or post['bo
    <input type="submit" value="Save">
  </form>
  <hr>
  <form action="{{ url_for('blog.delete', id=post['id']) }}" method="po
    <input class="danger" type="submit" value="Delete" onclick="return
  </form>
{% endblock %}
```

This template has two forms. The first posts the edited data to the current page (`/<id>/update`). The other form contains only a button and specifies an `action` attribute that posts to the delete view instead. The button uses some JavaScript to show a confirmation dialog before submitting.

The pattern `{{ request.form['title'] or post['title'] }}` is used to choose what data appears in the form. When the form hasn't been submitted, the original `post` data appears, but if invalid form data was posted you want to display that so the user can fix the error, so `request.form` is used instead. **`request`** is another variable that's automatically available in templates.

Delete

The delete view doesn't have its own template, the delete button is part of `update.html` and posts to the `/<id>/delete` URL. Since there is no template, it will only handle the `POST` method and then redirect to the `index` view.

`flaskr/blog.py`

```python
@bp.route('/<int:id>/delete', methods=('POST',))
@login_required
def delete(id):
    get_post(id)
    db = get_db()
    db.execute('DELETE FROM post WHERE id = ?', (id,))
    db.commit()
    return redirect(url_for('blog.index'))
```

Congratulations, you've now finished writing your application! Take some time to try out everything in the browser. However, there's still more to do before the project is complete.

Continue to Make the Project Installable.