

# Blueprint du blog

Vous utiliserez les mêmes techniques que celles que vous avez apprises lors de l'écriture du modèle d'authentification pour écrire le modèle du blog. Le blog doit lister tous les messages, permettre aux utilisateurs connectés de créer des messages et permettre à l'auteur d'un message de le modifier ou de le supprimer.

À mesure que vous implémentez chaque vue, laissez le serveur de développement fonctionner. Lorsque vous enregistrez vos modifications, essayez d'aller à l'URL dans votre navigateur et de les tester.

## Le *blueprint*

Définir le *blueprint* et l'enregistrer dans la fabrique d'applications.

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__)
```


Importez et enregistrez le *blueprint* depuis la fabrique en utilisant `app.register_blueprint()`. Placez le nouveau code à la fin de la fonction de la fabrique avant de retourner l'application.

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
```

Contrairement au *blueprint* pour l'authentification, le *blueprint* du blog n'a pas d'URL  `v: latest` ▼. Donc la vue `index` sera à `/`, la vue `create` à `/create`, et ainsi de suite. Le blog est la fonctionnalité principale de Flaskr, il est donc logique que l'index du blog soit l'index principal.

Cependant, le point de terminaison pour la vue `index` définie ci-dessous sera `blog.index`.

Certaines vues d'authentification se référaient à un point de terminaison `index` simple.

`app.add_url_rule()` associe le nom du point de terminaison `'index'` à l'url `/` de sorte que `url_for('index')` ou `url_for('blog.index')` fonctionneront tous les deux, générant la même URL `/` dans les deux cas.

Dans une autre application, vous pourriez donner au *blueprint* du blog un `url_prefix` et définir une vue `index` distincte dans la fabrique d'application, similaire à la vue `hello`. Les URLs et les points de terminaison `index` et `blog.index` seraient alors différents.

## Index

L'index montrera tous les messages, les plus récents en premier. Un `JOIN` est utilisé pour que les informations sur l'auteur provenant de la table `user` soient disponibles dans le résultat.

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['created'] }}
</div>
<div class="about">by {{ post['username'] }} on {{ post['created'] }}
</div>
{% if g.user['id'] == post['author_id'] %}
<a class="action" href="{{ url_for('blog.update', id=post['id']) }}">Update</a>
{% endif %}
</div>
<div class="body">{{ post['body'] }}</div>
<div class="meta">
<div class="created">{{ post['created'] }}</div>
<div class="username">{{ post['username'] }}</div>
</div>
</article>
{% endfor %}
</div>
<div class="latest">
<a href="{{ url_for('blog.index') }}">v: latest</a>
</div>
```

```

</header>
<p class="body">{{ post['body'] }}</p>
</article>
{% if not loop.last %}
    <hr>
{% endif %}
{% endfor %}
{% endblock %}

```

Lorsqu'un utilisateur est connecté, le bloc `header` ajoute un lien vers la vue `create`. Lorsque l'utilisateur est l'auteur d'un message, il verra un lien « Editer » vers la vue `update` de ce message. `loop.last` est une variable spéciale disponible dans [Jinja for loops](#). Elle est utilisée pour afficher une ligne après chaque message, sauf le dernier, afin de les séparer visuellement.

## Créer

La vue `create` fonctionne de la même manière que la vue `register` d'authentification. Soit le formulaire est affiché, soit les données postées sont validées et le message est ajouté à la base de données, soit une erreur est affichée.

Le décorateur `login_required` que vous avez écrit plus tôt est utilisé sur les vues du blog. Un utilisateur doit être connecté pour visiter ces vues, sinon il sera redirigé vers la page de connexion.

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'))

```

 v: latest ▾

```
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'] }}</textarea>
        <input type="submit" value="Save">
    </form>
{% endblock %}
```

## Mise à jour

Les vues `update` et `delete` devront toutes deux récupérer un `post` par `id` et vérifier si l’auteur correspond à l’utilisateur connecté. Pour éviter de dupliquer le code, vous pouvez écrire une fonction pour récupérer le `post` et l’appeler depuis chaque vue.


```
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, f"Post id {id} doesn't exist.")

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

    return post
```

`abort()` lèvera une exception spéciale qui renverra un code d’état HTTP. Il prend  `v: latest` ▼ optionnel à afficher avec l’erreur, sinon un message par défaut est utilisé. 404 signifie « Not

Found », et 403 signifie « Forbidden ». (401 signifie « Non autorisé », mais vous redirigez vers la page de connexion au lieu de renvoyer ce statut).

L'argument `check_author` est défini pour que la fonction puisse être utilisée pour obtenir un `post` sans vérifier l'auteur. Ce serait utile si vous écriviez une vue pour montrer un article individuel sur une page, où l'utilisateur n'a pas d'importance parce qu'il ne modifie pas l'article.

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)
```

Contrairement aux vues que vous avez écrites jusqu'à présent, la fonction `update` prend un argument, `id`. Cela correspond au `<int:id>` dans l'URL. Une vraie URL ressemblera à `/1/update`. Flask va capturer le `1`, s'assurer que c'est un `int`, et le passer comme argument `id`. Si vous ne spécifiez pas `int:` et faites plutôt `<id>`, ce sera une chaîne de caractères. Pour générer une URL vers la page pour mettre à jour, il faut fournir à `url_for()` l'argument `id` pour qu'il sache quoi remplir : `url_for('blog.update', id=post['id'])`. Ceci est également dans le fichier `index.html` ci-dessus.

Les vues `create` et `update` sont très similaires. La principale différence est que la vue `update` utilise un objet `post` et une requête `UPDATE` au lieu d'une `INSERT`. Avec une refactorisation intelligente, vous pourriez utiliser une vue et un modèle pour les deux actions, mais pour le tuto-

 v: latest ▼

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['body'] }}</textarea>
  <input type="submit" value="Save">
</form>
<hr>
<form action="{{ url_for('blog.delete', id=post['id']) }}" method="post">
  <input class="danger" type="submit" value="Delete" onclick="return confirm('Are you sure you want to delete this post?');">
</form>
{% endblock %}
```

Ce modèle a deux formes. Le premier affiche les données modifiées sur la page actuelle (`/<id>/update`). L'autre formulaire ne contient qu'un bouton et spécifie un attribut `action` qui affiche la vue de suppression à la place. Le bouton utilise du JavaScript pour afficher une boîte de dialogue de confirmation avant l'envoi.

Le motif `{{ request.form['title'] or post['title'] }}` est utilisé pour choisir les données qui apparaissent dans le formulaire. Lorsque le formulaire n'a pas été soumis, les données originales `post` apparaissent, mais si des données de formulaire invalides ont été postées, vous voulez les afficher pour que l'utilisateur puisse corriger l'erreur, donc `request.form` est utilisé à la place. `request` est une autre variable qui est automatiquement disponible dans les modèles.

## Supprimer

La vue de suppression n'a pas son propre modèle, le bouton de suppression fait partie de `update.html` et renvoie à l'URL `/<id>/delete`. Puisqu'il n'y a pas de modèle, il ne traitera que la méthode `POST` et redirigera ensuite vers la vue `index`.

flaskr/blog.py

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

 v: latest ▾

```
db = get_db()
db.execute('DELETE FROM post WHERE id = ?', (id,))
db.commit()
return redirect(url_for('blog.index'))
```

Félicitations, vous avez maintenant fini d'écrire votre application ! Prenez le temps de tout essayer dans le navigateur. Cependant, il reste encore beaucoup à faire avant que le projet ne soit complet.

Passez à [Rendre le projet installable.](#)