

Petit Blog avec Flask

Introduction

Flask est un petit framework web Python léger qui fournit des outils et des fonctionnalités facilitant la création d'applications web en Python. Il offre aux développeurs une certaine flexibilité et est plutôt accessible pour les néophytes puisqu'il est possible de construire rapidement une application web en utilisant un seul fichier Python. Flask est également extensible et n'impose pas une architecture particulière.

Dans le cadre de ce tutoriel, le framework Bootstrap est utilisé pour styliser l'application rapidement. Bootstrap propose du code css simple à intégrer dans une page web, responsive et designé. C'est une aide appréciable qui permet de se concentrer sur l'apprentissage du fonctionnement de Flask.

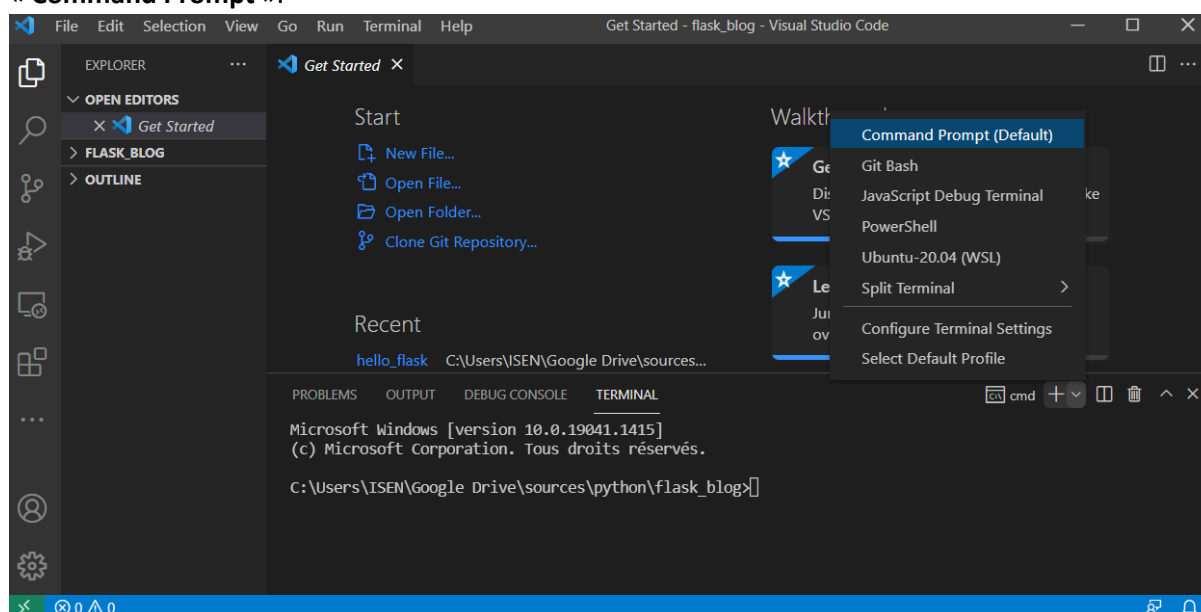
Flask utilise le moteur de modèle Jinja pour construire dynamiquement des pages HTML en utilisant des concepts Python familiers tels que les variables, les boucles, les listes, etc.

Dans ce tutoriel, vous allez construire un petit blog web en utilisant Flask et MySQL. Les utilisateurs de l'application peuvent consulter tous les articles de votre base de données et cliquer sur le titre d'un article pour en voir le contenu, avec la possibilité d'ajouter un nouvel article à la base de données et de modifier ou supprimer un article existant.

Environnement de travail

Commencez par créer un répertoire qui va contenir tout votre projet, *flask_blog* par exemple, et ouvrez le dans VS CODE.

Ouvrez le terminal (menu **Terminal/New Terminal**). Faites attention à bien utiliser une *invite de commandes* et pas un *Powershell*, ce dernier ne gère pas les environnements virtuels. Le mot *cmd* doit être affiché à droite de votre Terminal. Si ce ne pas le cas utilisez la flèche pour sélectionner « **Command Prompt** ».



Vérifiez, dans le *Terminal*, que vous êtes bien positionné dans votre répertoire projet.

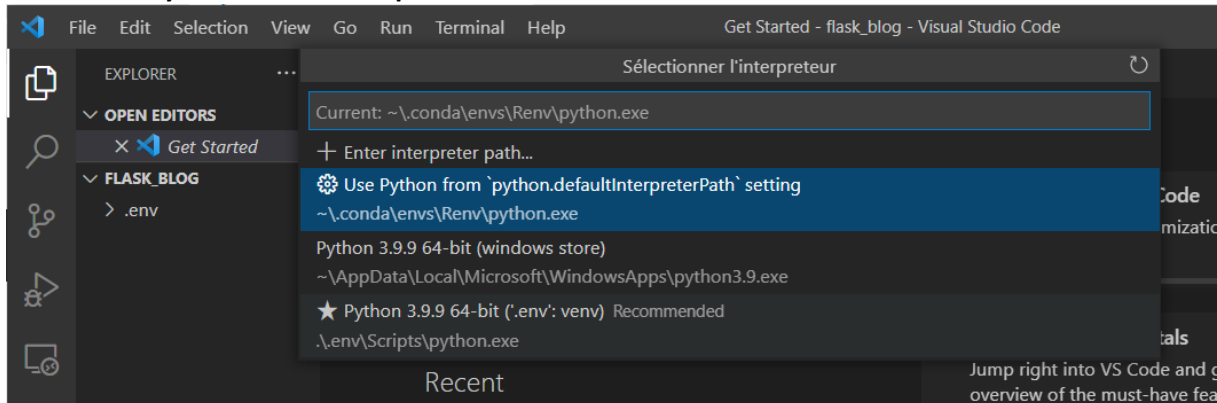
```
C:\Users\ISEN\sources\python\flask_blog>
```

Créez un environnement virtuel et activez-le :

```
C:\Users\ISEN\sources\python\flask_blog> python -m venv .env
```

```
C:\Users\ISEN\sources\python\flask_blog> .env\scripts\activate
```

L'environnement démarré, sélectionnez le bon interpréteur, le `.env` dans notre cas, grâce à la commande **Python : Select Interpreter** dans la **Command Palette** :



Il ne vous reste plus qu'à installer Flask et MySQL Connector après avoir mis à jour **PIP**:

```
C:\Users\ISEN\sources\python\flask_blog> pip install --upgrade pip
```

```
C:\Users\ISEN\sources\python\flask_blog> pip install flask
```

```
C:\Users\ISEN\sources\python\flask_blog> pip install mysql-connector-python
```

Créer une première page web

Dans cette étape, vous allez créer une petite application web dans un fichier Python et l'exécuter pour démarrer le serveur, qui affichera certaines informations sur le navigateur.

Pour ça, ajoutez un fichier `app.py` dans votre projet. Ce dernier contient le code suivant :

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def index():
    return 'Hello, World!'

if __name__ == "__main__":
    app.run(debug=True)
```

Dans le bloc de code précédent, vous importez d'abord l'objet **Flask** du paquet **flask**. Vous l'utilisez ensuite pour créer votre instance d'application **Flask** de nom **app**. Vous passez la variable spéciale `__name__` qui contient le nom du module Python actuel.

@app.route() est un décorateur qui indique que le retour d'une fonction Python est une réponse HTTP, protocole utilisé par un navigateur web. **@app.route()** prend en paramètre l'URL à laquelle la fonction doit répondre, ici la racine du site web, indiqué par `/`.

App.run(debug=True) démarre le serveur en mode test et par défaut sur le *localhost*, port 5000. Si vous voulez modifier le port, il faut ajouter l'argument **port=5001**, par exemple.

Vous pouvez faire un **run** du code et si tout se passe bien une url s'affiche dans le *Terminal*. **CTRL+CLICK** dessus pour afficher votre page dans le navigateur par défaut.

CTRL+C, dans le Terminal, pour stopper le serveur.

Les modèle HTML

Les navigateurs affichent du HTML. Voyons comme faire de belle appli web en incorporant ce type de fichiers.

Flask fournit une fonction **render_template()** qui fait appel au moteur Jinja pour retourner une page HTML dans une réponse HTTP. Vous pouvez ainsi intégrer des fichiers *.html* à votre Python.

Le code de *app.py* devient :

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
def index():
    return render_template('index.html')

if __name__ == "__main__":
    app.run(debug=True)
```

En lisant ce code, vous comprenez que vous devez intégrer un fichier *index.html*. Flask impose que ce fichier soit placé dans un répertoire */templates*. Dans votre projet *flask_blog*, créez le répertoire */templates* et ajoutez y le fichier *index.html* suivant :

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <title>FlaskBlog</title>
</head>

<body>
  <h1>Welcome to FlaskBlog</h1>
</body>

</html>
```

C'est du HTML tout ce qu'il y a de plus classique. Testez votre page en lançant *app.py*.

Pour utiliser du CSS, ajoutez un répertoire */static*, destiné à recevoir toutes les ressources du site. Vous pouvez alors créer, dans ce dossier, le fichier *style.css* suivant :

```
h1 {
  border: 2px #eee solid;
  color: brown;
  text-align: center;
  padding: 10px;
}
```

Rien de nouveau, côté CSS. Par contre, dans *index.html*, vous devez indiquer à Jinja qu'il y a une URL statique à calculer, en ajoutant cette ligne dans l'entête du fichier:

```
<link rel="stylesheet" href="{{ url_for('static', filename= 'style.css') }}">
```

La balise **link** est bien celle qu'il faut utiliser en HTML pour référencer un fichier de style. Par contre, vous devez injecter du code Python, grâce aux `{{}}`, pour que le moteur de rendu puisse calculer l'URL du fichier *.css* grâce à la fonction `url_for()`.

Testez l'appli pour voir si le style est bien appliqué.

Héritage de modèles

Une des richesses des moteurs de rendu comme Jinja est de proposer un système de templates permettant de factoriser du code HTML. Cela garantit l'intégrité graphique du site sans avoir à recopier les mêmes balises dans toutes les pages web.

Ajoutez le fichier *base.html* dans */templates* :

```
<!doctype html>
<html lang="en">

  <head>
    <!-- Required meta tags -->
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
    <!-- Bootstrap CSS -->
    <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css"
  integrity="sha384-
ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQU0hcWr7x9JvoRxT2MZw1T"
crossorigin="anonymous">
    <title>{% block title %} {% endblock %}</title>
  </head>

  <body>
    <nav class="navbar navbar-expand-md navbar-light bg-light">
      <a class="navbar-brand" href="{{ url_for('index') }}">FlaskBlog</a>

      <button class="navbar-toggler" type="button" data-toggle="collapse" data-
target="#navbarNav" aria-controls="navbarNav" aria-expanded="false" aria-
label="Toggle navigation">
        <span class="navbar-toggler-icon"></span>
      </button>

      <div class="collapse navbar-collapse" id="navbarNav">
        <ul class="navbar-nav">
          <li class="nav-item active">
            <a class="nav-link" href="#">About</a>
          </li>
        </ul>
      </div>
    </nav>

    <div class="container">
      {% block content %} {% endblock %}
    </div>
```

```
<!-- Optional JavaScript -->
<!-- jQuery first, then Popper.js, then Bootstrap JS -->

<script src=https://code.jquery.com/jquery-3.3.1.slim.min.js
integrity="sha384
q8i/X+965Dz00rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo"
crossorigin="anonymous"></script>

<script
src=https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.7/umd/popper.min.js
integrity="sha384
U02eT0CpHqdSjQ6hJty5KVphtPhzWj9W01c1HTMGa3JDZwrnQq4sF86dIHNDz0W1"
crossorigin="anonymous"></script>

<script
src=https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/bootstrap.min.js
integrity="sha384
JjSmVgyd0p3pXB1rRibZUAYoIIy60rQ6VrjIEaFf/nJGzIxFDsf4x0xIM+B07jRM"
crossorigin="anonymous"></script>
</body>
</html>
```

Il y a encore des injections dans le modèle. Tout d'abord les blocs `{% block title %}` et `{% block content %}`, qui indique, lors de l'héritage, où le contenu doit être injecté. Ils se terminent tous par `{% endblock %}`, et leurs noms, ici **title** et **content**, sont libres.

Tous les url des liens sont calculées par le moteur de rendu, grâce, une fois de plus, à la fonction `url_for()` qui cette fois-ci prend en argument le nom de la fonction Python dont il faut récupérer l'URL. Ici le paramètre `'index'` fait référence à la méthode `index()` de `app.py`, dont on veut récupérer l'URL de la route `/` en paramètre de `@app.route('/')`.

A noter que BOOTSTRAP est utilisé dans `base.html` pour la mise en page.

Il faut donc réécrire `./templates/index.html`, pour utiliser le modèle :


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

{% block content %}
  <h1>{% block title %} Welcome to FlaskBlog {% endblock %}</h1>
{% endblock %}
```

Le code `extends 'base.html'` indique que le modèle hérite de `base.html`. Les blocs de contenu sont définis ici, avec les mêmes noms que dans le modèle racine. Ce sont ces contenus qui vont être injectés dans le modèle.

La base de données

Les articles du blog vont être sauvegarder en base. Créez une base `flask_blog` sous MySQL. Y ajouter la table suivante :

#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra
1	id 	int(11)			No	None		AUTO_INCREMENT
2	created	timestamp			No	CURRENT_TIMESTAMP		
3	title	varchar(50)	utf8_general_ci		No	None		
4	content	text	utf8_general_ci		No			

id est la clé primaire,
created : date de création du poste,
title : le titre du post,
content : le texte du post.

Ajoutez les 2 posts suivant :

Title	content
Premier	Trop super Flask, en plus j'ai un prof au top
Deuxième	Je ne suis pas d'accord !

Afficher tous les posts

La base avec les posts est en place. Il faut, maintenant, la lire pour afficher les messages sur le site. Pour ça, nous allons ajouter un fichier, *data.py*, qui contient le code d'accès à la base :

```
import mysql.connector as mysqlpyth

bdd = None
cursor = None

def connexion():
    global bdd
    global cursor

    bdd = mysqlpyth.connect(user='root', password='root', host='localhost',
                             port="8081", database='flask_blog')
    cursor = bdd.cursor()

def deconnexion():
    global bdd
    global cursor

    cursor.close()
    bdd.close()

def lire_posts():
    global cursor

    connexion()
    query = "SELECT * FROM posts"
    cursor.execute(query)
    posts = []

    for enregistrement in cursor :
        post = {}
        post['id'] = enregistrement[0]
        post['created'] = enregistrement[1]
        post['title'] = enregistrement[2]
        post['content'] = enregistrement[3]
        posts.append(post)

    deconnexion()
    return posts
```

La fonction **connexion()** ouvre la connexion vers la base *flask_blog*, la fonction **deconnexion()** la ferme et la fonction **lire_posts()** retourne tous les posts lus en base sous forme d'une liste de dictionnaires.

Il faut maintenant modifier le site pour y afficher les posts. Tout d'abord, modifier le code de *index.html* pour que les posts s'y affichent :

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

{% block content %}
    <h1>{% block title %} Welcome to FlaskBlog {% endblock %}</h1>
    {% for post in posts %}
        <a href="{{ url_for('post', post_id=post['id']) }}">
            <h2>{{ post['title'] }}</h2>
        </a>

        <span class="badge badge-primary">{{ post['created'] }}</span>
        <hr>
    {% endfor %}
{% endblock %}
```

Grâce à l'injection de code, il est possible de parcourir une liste de posts, avec un **for**, est d'afficher les valeurs des dictionnaires, ayant pour clés *title* et *created*. La fonction **url_for()** permet de générer des URL, mais on y revient lorsqu'on travaillera sur l'affichage d'un post unique. Donc la page HTML a besoin d'une liste de posts. Cette liste, c'est au serveur de la passer à la page et le serveur est codé dans *app.py* :

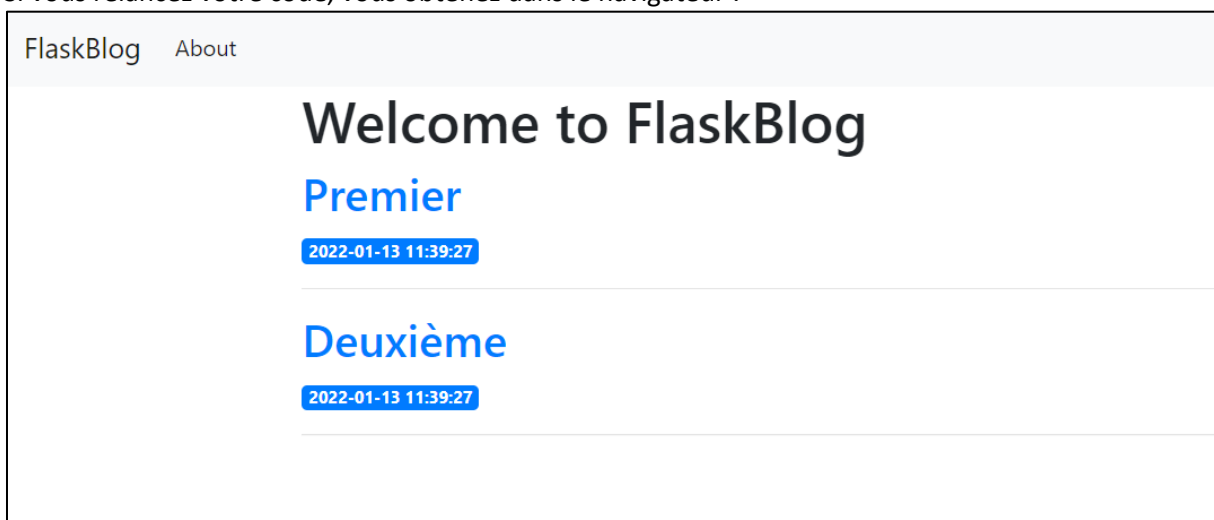
```
from flask import Flask, render_template
import data

app = Flask(__name__)

@app.route('/')
def index():
    posts = data.lire_posts()
    return render_template('index.html', posts = posts)

if __name__ == "__main__":
    app.run(debug=True)
```

Si vous relancez votre code, vous obtenez dans le navigateur :



Afficher un post

Pour afficher un post, il faut ajouter une nouvelle page et donc une nouvelle fonction dans notre serveur web *app.py*. Cette page doit s'ouvrir à partir de l'URL défini dans *index.html* :

```
<a href="{{ url_for('post', post_id=post['id']) }}">
```

En lisant cette ligne de code, la fonction `url_for()` va générer une URL pour appeler la fonction `post()`, dans le fichier *app.py* et qui prend en paramètre `post_id`. Il faut donc ajouter cette fonction :

```
@app.route('/<int:post_id>')
def post(post_id):
    post = data.get_post(post_id)
    return render_template('post.html', post = post)
```

Dans *data.py*, il faut donc une fonction qui retrouve un post à partir de son *id* :

```
def get_post(id):
    global cursor

    connexion()
    query = "SELECT * FROM posts WHERE id=" + str(id)
    cursor.execute(query)
    post = {}

    for enregistrement in cursor :
        post['id'] = enregistrement[0]
        post['created'] = enregistrement[1]
        post['title'] = enregistrement[2]
        post['content'] = enregistrement[3]

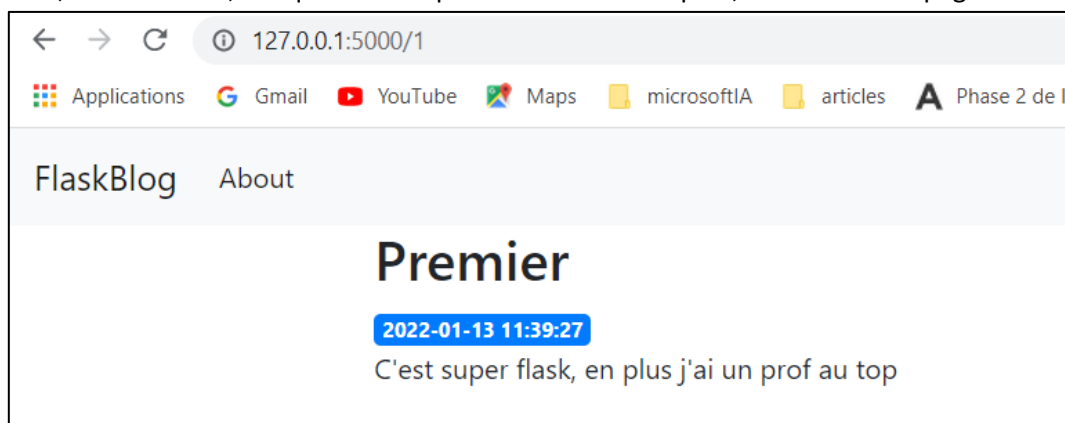
    deconnexion()
    return post
```

Enfin, il faut ajouter une nouvelle page HTML, *post.html* dans le dossier */templates* :

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

{% block content %}
    <h2>{% block title %} {{ post['title'] }} {% endblock %}</h2>
    <span class="badge badge-primary">{{ post['created'] }}</span>
    <p>{{ post['content'] }}</p>
{% endblock %}
```

Désormais, sur votre site, lorsque vous cliquez sur le titre d'un post, vous ouvrez la page suivante :



Saisir un article

Pour que notre blog soit complet, il nous faut ajouter un formulaire permettant de rédiger de nouveaux posts.

Commençons par ajouter *create.html* dans */templates* :

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

{% block content %}

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

<form method="GET" action="{{ url_for('add') }}">
  <div class="form-group">
    <label for="title">Title</label>

    <input type="text" name="title"
      placeholder="Post title" class="form-control"></input>
  </div>

  <div class="form-group">
    <label for="content">Content</label>

    <textarea name="content" placeholder="Post content"
      class="form-control"></textarea>
  </div>

  <div class="form-group">
    <button type="submit" class="btn btn-primary">Submit</button>
  </div>
</form>

{% endblock %}
```

La page contient un formulaire pour la saisie du titre et du contenu. La **method= « GET »** permet l'envoi des données saisies vers serveur, à l'URL **action= « {{url_for('add')}} »**. C'est le bouton **« submit »** qui déclenche l'envoi des données vers le serveur.

Puis mettre à jour le menu de *base.html* pour accéder à cette nouvelle page :

```
<li class="nav-item">
  <a class="nav-link" href="{{url_for('create')}}">New Post</a>
</li>
```

Et évidemment, ajouter le code qui affiche le formulaire dans le serveur *app.py* :

```
@app.route('/create')
def create():
    return render_template('create.html')
```

Vous devriez obtenir le formulaire suivant :

Enregistrer le post

Enfin, dernière étape, il faut enregistrer les saisies dans la base de données, tout d'abord en ajoutant une nouvelle fonction dans *data.py* :

```
def set_post(title, content):
    global cursor
    global bdd

    connexion()

    query = 'INSERT INTO posts(title, content) VALUES'
    (''+title+'',''+content+'');

    cursor.execute(query)
    bdd.commit()

    deconnexion()
```

set_post(title, content) prend en paramètre le titre et le contenu du post à ajouter dans la table *posts* de la base de données. Vous reconnaissez sans doute la requête d'ajout, **INSERT INTO**. Une fois la requête exécutée, il faut **commit()** les modifications vers la base sinon elles ne sont pas prises en compte.

Cette nouvelle fonction est appelée à partir d'une nouvelle page web, dans *app.py* :

```
@app.route('/add', methods=['GET'])
def add():
    title = request.values.get('title')
    content = request.values.get('content')
    data.set_post(title, content)
    return render_template('add.html')
```

La route et la méthode sont bien celles définies dans le formulaire *create.html*. Cette nouvelle fonction va donc récupérer les saisies dans le formulaire, grâce à l'objet **request**, et les passer à **set_post()** pour qu'elles soient enregistrées en base. *add.html* est un page pour informer que tout s'est bien passé :

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

{% block content %}
    <h2>{% block title %} Ajout de message {% endblock %}</h2>
    <p>Votre message a bien été ajouté.</p>
{% endblock %}
```

Il ne vous reste plus qu'à tester et lorsque vous cliquez sur le bouton submit, vous retrouvez vos saisies dans l'URL :

<http://127.0.0.1:5000/add?title=Troisi%C3%A8me&content=Mettez+moi+%C3%A7a+en+base>

C'est la méthode GET qui ajoute ces informations et c'est comme ça que notre serveur Python récupère les saisies à traiter.