Flask avec Visual Studio

Source :

D'après https://perso.liris.cnrs.fr/pierre-antoine.champin/2017/progweb-python/cours/cm2.html

I. Présentation de Flask

1. Framework opensource

Flask est un framework opensource permettant le développement d'applications web.

Framework : pour faire simple c'est un ensemble de bibliothèques qui impose un cadre de travail. Les plus utilisés sont jQuery, React.js, Angular, ASP.NET …

2. Spécificités du framework flask

- Programmation en python
- Léger, minimaliste
- Utilisation de templates (Jinja2) qui permet de séparer le Model/Controller et la View (architecture MVC)

3. Comparaison avec django

Django est également un framework web python, mais il se distingue de flask par :

- Plus complet
- Plus « lourd »
- Développement pure web plus rapide
- Préférable pour les applications plus complexes

II. Premier programme

Soit le programme :

```
1. from flask import Flask
2.
3. app = Flask(__name__)
4.
5. @app.route("/")
6. def root():
7.     return "Hello world"
```

1. Explications des lignes

1. importation de flask
3.définition d'une application flask
5. décorateur python (cf plus loin) qui sert à indiquer l'url pour laquelle la vue est définie
6.c'est une fonction appelée *vue*, c'est la réponse à la demande de l'url au format HTML encodé en utf-8 et le status de la réponse est par défaut 200

2. Décorateur

Un décorateur permet la modification du comportement par défaut de fonctions ou de classes (c.f. *Flask – Les décorateurs --)*

3. Route

En développement Web, on appelle ***route*** une URL ou un ensemble d'URLs conduisant à l'exécution d'une fonction donnée.

Dans Flask, les routes sont déclarées via le décorateur **app.route**, comme dans l'exemple ci-dessus.

Une route peut être paramétrée, auquel cas le paramètre sera passé à la fonction vue :

```
@app.route("/")
def home():
    return "Hello, Flask!"
```

III. Les décorateurs et flask

1. Les url et les vue :

Un décorateur permet la modification du comportement par défaut d'une fonction

Le décorateur `@app.route` permet la modification du comportement par défaut d'une fonction et comme la fonction `app.route` va répondre à la demande d'une URL.

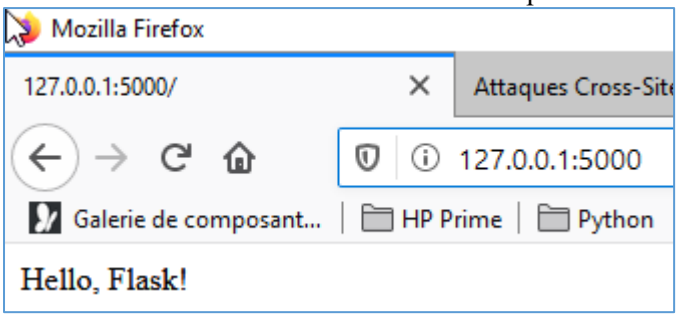
Ainsi

```
@app.route("/")
def home():
```

Cela revient à flask à trouver la fonction, appelée *vue*, correspondant l'url /

```
from flask import Flask
app = Flask(__name__)
@app.route("/")
def home():
    return "Hello, Flask!"
```

Ainsi dans l'exemple précédent la fonction `home` va être décoré avec la fonction `app.route` avec comme argument "\" : pour faire simple, la chaîne `Hello, Flask` va être renvoyée lorsque votre navigateur voudra accéder à la page `127.0.0.1` ou *192.168.1.1* si cette adresse est l'adresse ip de votre machine :



Un autre exemple `@app.route("/hello/")` va permettre à flask d'identifier la vue correspond à l'url `/hello/` comme étant la fonction `hello_there` :

```
@app.route("/hello/")
```

```
def hello_there(name):
```

2. Passage d'arguments à la fonction vue

Il est possible de passer un ou plusieurs paramètres :

```
@app.route("/hello/<name>")
def hello_there(name = "rtk"):
```

Le paramètre s'appelle dans ce cas *name.*

Il est possible de :

- d'avoir plusieurs paramètres, par exemple `/hello/<a>/<b>`
- d'imposer un type aux paramètres, par exemple `/user/<int:ident>`.

3. Cumulation

Plusieurs décorateurs pour une seule fonction : ainsi la fonction hello_there va être appelé pour les urls `/` , `/hello/` et `/hello/` suivi d'un nom

```
@app.route("/")
@app.route("/hello/")
@app.route("/hello/<name>")
def hello_there(name = "rtk"):
```

Dans notre cas, l'ordre n'a pas d'importance.

IV. Génération d'url <-> vue

1. Premiers exemples

Nous avons vu que @app.route permet de définir une vue :

```
@app.route("/hello/<name>")
def hello_there(name = "rtk"):
```

L'inverse est réalisé par la commande flask.url_for. Deux exemples d'utilisation :

1. Dans un fichier template :

```
 ...
<div class="navbar">
            <a href="{{ url_for('home') }}" class="navbar-brand">Home</a>
            <a href="{{ url_for('about') }}" class="navbar-item">About</a>
            <a href="{{ url_for('contact') }}" class="navbar-item">Contact</a>
</div> ...
```

Et le code html généré est :

```
...
    <div class="navbar">
        <a href="/" class="navbar-brand">Home</a>
        <a href="/Apropos/" class="navbar-item">About</a>
        <a href="/contact/" class="navbar-item">Contact</a>
```

```
    </div>
...
```

2. Dans un fichier python :

```
from flask import Flask
from datetime import datetime
from flask import render_template, url_for # NE PAS OUBLIER

@app.route("/fichier/data")
def get_data():
    return app.send_static_file("data.json")
```

Puis dans le code

```
...
    print(url_for('get_data'))
...
```

L'affichage est :

```
/fichier/data
```

2. Avec des paramètres de la fonction

Avec cette route/vue qui relie une url avec une fonction demandant un paramètre *nom* :

```
@app.route("/hello/<name>")
def hello_there(name = "rtk"):
...
```

L'affichage de

```
    print(url_for('hello_there',name='toto'))
```

est :

```
/hello/toto
```

Et même avec des paramètres supplémentaires cad qui ne sont pas des paramètres de la fonction :

```
    print(url_for('hello_there',name='toto',info='informations'))
```

Affiche

```
/hello/toto?info=informations
```

3. url absolue avec _external

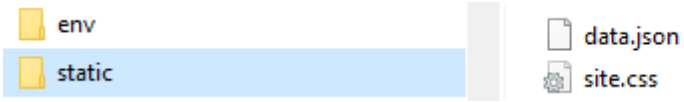
Au lieu de renvoyer une url relative, le paramètre _external permet d'obtenir l'absolue. Observez la différence des deux affichages :

```
    print(url_for('hello_there',name='toto',info='informations'))
    print(url_for('hello_there',name='toto',info='informations',_external=True))
```

```
/hello/toto?info=informations
http://127.0.0.1:5000/hello/toto?info=informations
```

V. Les ressources statiques

Tous les fichiers *statiques* comme par exemple les images, le CSS … doivent se trouver dans un répertoire dont le nom est *static :*



De même il est possible d'obtenir url par *url_for* avec deux paramètres 'static' et filename suivie du nom du fichier :

```
print(url_for('static',filename='site.css'))
```

```
/static/site.css
```

VI. Les templates

Les templates utilisés dans flask sont de type Jinja2 : https://jinja.palletsprojects.com/en/2.11.x/

1. But et principe du template

Un *template* est un modèle de page html qui est écrit une seule fois puis qui sera utilisé, exécuté dans tout le programme ainsi toutes les pages se ressembleront.

Par exemple, le modèle sera "hello.html" :

```
<p>Hello {{name}}</p>
```

Lors de l'exécution du programme, le code va remplacer *{{name}}*, qui indique une variable, par la valeur de la variable *name :*

```
@app.route("/hello/<nom>")
def hello_there(nom):
    return render_template(
        "hello.html",
        name=nom)
```

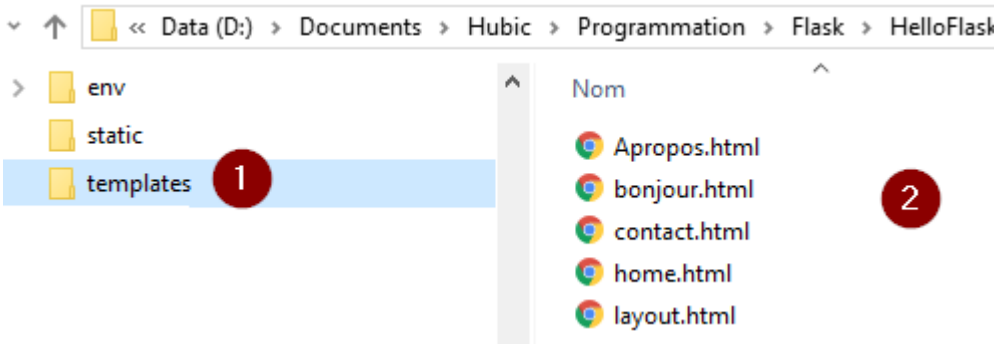
Le résultat serait si le nom est *toto* :

```
<p>Hello toto</p>
```

Les templates permettent de séparer le fond de la forme.

2. Répertoires des templates

Les modèles jinja2 permettant de générer tous les formats web, doivent être déposés dans le répertoire *templates :*



(2) : liste des templates

3. Exemple plus complet de template

a. Soit le fichier bonjour.html :

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Hello, Flask</title>
        <link rel="stylesheet" type="text/css" href="{{ url_for('static', filename='site.c
ss')}}" />
    </head>
    <body>
        {%if name %}
            <span class="message">Bonjour {{ name }} !</span> il est  {{ date.strftime("%A
, %d %B, %Y at %X") }}.
        {% else %}
        <span class="message">Quel est votre nom ? </span>  Vous pouvez le fournir à la su
ite de  /hello/ dans l'URL.
        {% endif %}
    </body>
</html>
```

b. Substitution :

Dans le cas d'utilisation de css, tous les pages html doit faire appel au fichier ccs.
C'est pourquoi on trouve une substitution au début du fichier :

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

Cette ligne est du code html SAUF les accolades {{…}} :

```
{{ url_for('static', filename='site.css')}}
```

Les doubles accolades indiquent une substitution :

```
{{ url_for('static', filename='site.css')}}
```

Va être remplacé par la valeur de `url_for('static', filename='site.css')`
`zPar¶`
Par exemple : /static/site.css

Ainsi le code HTML généré sera :

```
<link rel="stylesheet" type="text/css" href="/static/site.css" />
```

c. Structures de contrôle

Avec jinja2 on retrouve des structures classiques de contrôle comme les boucles ou les conditionnels.

Par exemple pour une **conditionnelle** :

```
{%if name %}
<span class="message">Bonjour {{ name }} !</span> il est  {{ date.strftime("%A, %d %B, %Y
at %X") }}.
{% else %}
<span class="message">Quel est votre nom ? </span>  Vous pouvez le fournir à la suite de
/hello/ dans l'URL.
{% endif %}
```

Une **boucle** :

```
{% if elements %}
  <ul>
  {% for e in elements %}
    <li><a href="{{e.link}}">{{e.name}}</li>
  {% endfor %}
  </ul>
{% else %}
  <p>Aucun élément</p>
{% endif %}
```

VII. Créer plusieurs templates à partir d'une base commune

Une sorte de mécanisme d'héritage/d'extension peut être utilisé avec les templates.

1. *Un template de « base »*

a. Soit le fichier *layout.html* sauvegardé dans le répertoire *templates* :

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>{% block title %}{% endblock %}</title>
        <link rel="stylesheet" type="text/css" href="{{ url_for('static', filename='site-
05.css')}}" />
    </head>

    <body>
        <div class="navbar">
            <a href="{{ url_for('home') }}" class="navbar-brand">Home</a>
```

```
            <a href="{{ url_for('about') }}" class="navbar-item">About</a>
            <a href="{{ url_for('contact') }}" class="navbar-item">Contact</a>
        </div>

        <div class="body-content">
            {% block content %}
            {% endblock %}
            <hr/>
            <footer>
                <p>© 2020</p>
            </footer>
        </div>
    </body>
</html>
```

b. Organisation du fichier

Il a deux parties distingues :

1. Au début de fichier se trouve une navigation entre les différentes pages :

```
        <div class="navbar">
            <a href="{{ url_for('home') }}" class="navbar-brand">Home</a>
            <a href="{{ url_for('about') }}" class="navbar-item">About</a>
            <a href="{{ url_for('contact') }}" class="navbar-item">Contact</a>
        </div>
```

Ceci est valable pour toutes les pages basées sur ce template de « base »

2. Ensuite une partie propre aux futurs fichiers basés sur ce template :

```
            {% block content %}
            {% endblock %}
```

Par ailleurs, l'entête est propre aux fichiers étendus :

```
        <title>{% block title %}{% endblock %}</title>
```

c. Un CSS

Dans le précédent fichier une nouvelle classe CSS a été créée :

```
        <div class="body-content">
```

Sa définition se trouve dans un fichier css :

```
.body-content {
    padding: 5px;
    font-family:'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
}
```

D'autres classes ont été définies dans le css (dans le répertoire *static)*, le fichier complet est :

```
.navbar {
    background-color: lightslategray;
    font-size: 1em;
```

```
    font-
family: 'Trebuchet MS', 'Lucida Sans Unicode', 'Lucida Grande', 'Lucida Sans', Arial, sans
-serif;
    color: white;
    padding: 8px 5px 8px 5px;
}

.navbar a {
    text-decoration: none;
    color: inherit;
}

.navbar-brand {
    font-size: 1.2em;
    font-weight: 600;
}

.navbar-item {
    font-variant: small-caps;
    margin-left: 30px;
}

.body-content {
    padding: 5px;
    font-family:'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
}
```

2. Utilisation de ce template de base :

Par exemple la page de contact (*contact.html)* est généré par l'intermédiaire de ce template :

```
{% extends "layout.html" %}
{% block title %}
Contact
{% endblock %}
{% block content %}
<spand class="message">Page de contact</spand>
{% endblock %}
```

Explications :

1. Ce template est basée sur layout.html présenté plus haut :

```
{% extends "layout.html" %}
```

2. Le bloc titre est définie de cette façon :

```
{% block title %}
Contact
{% endblock %}
```

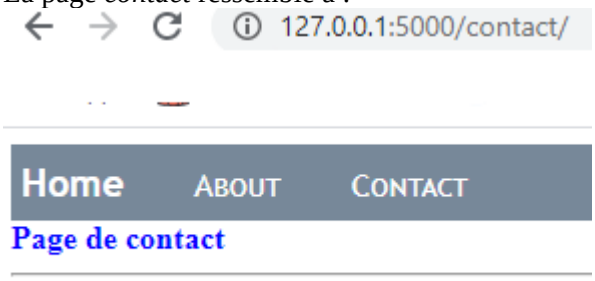
3. De même que le contenu

```
{% block content %}
```

```
<spand class="message">Page de contact</spand>
{% endblock %}
```

a. Résultats

La page *contact* ressemble à :



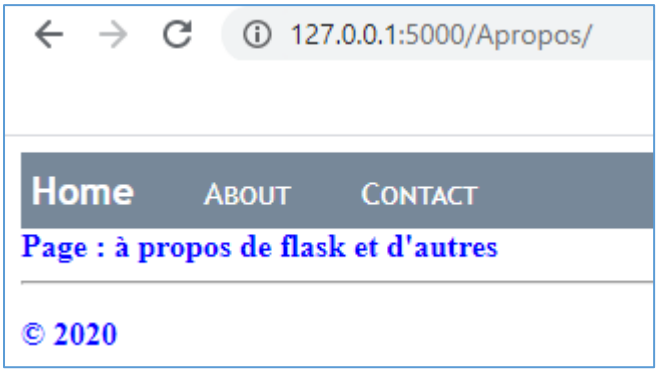
Et le code html est :

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>
Contact
</title>
        <link rel="stylesheet" type="text/css" href="/static/site-05.css" />
    </head>

    <body>
        <div class="navbar">
            <a href="/" class="navbar-brand">Home</a>
            <a href="/Apropos/" class="navbar-item">About</a>
            <a href="/contact/" class="navbar-item">Contact</a>
        </div>

        <div class="body-content">

Page de contact

            <hr/>
            <footer>
                <p>© 2020</p>
            </footer>
        </div>
    </body>
</html>
```

b. Résultats visuels pour toutes pages

L'idée derrière l'utilisation d'un template de base est d'avoir un visuel pour toutes les pages html.
Pour la page d'accueil :

La page About :



VIII. Résumé – schéma d'interaction –

app.py

```python
# app.py
app = Flask(__name__)
@app.route("/")
def home():
    return render_template("home.html")

@app.route("/Apropos/")
def about():
    return render_template('Apropos.html')

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

home.html

Apropos.html

```
<!-- contact.html -->
{% extends "layout-05.html" %}
{% block title %}
Contact...
{% endblock %}
{% block content %}
Page de contact
{% endblock %}
```

layout.html

```html
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>{% block title %}{% endblock %}</title>
        <link rel="stylesheet" type="text/css" href="{{ url_for('static', filename='site-05.css')}}"
    </head>

    <body>
        <div class="navbar">
            <a href="{{ url_for('home') }}" class="navbar-brand">Home</a>
            <a href="{{ url_for('about') }}" class="navbar-item">About</a>
            <a href="{{ url_for('contact') }}" class="navbar-item">Contact</a>
        </div>

        <div class="body-content">
            {% block content %}
            {% endblock %}
            <hr/>
            <footer>
                <p>© 2020</p>
            </footer>
        </div>
    </body>
</html>
```

site.css

```
    font-variant: small-caps;
    margin-left: 30px;
}

.body-content {
    font-weight: 600;
    color: blue;
}
```

¶