Objectifs de la feuille

- Introduction
- Héritage de templates
- URLs dynamiques
- Affichage des données
- Utilisation de Bootstrap

Introduction

Flask utilise le moteur de templates Jinja2 par défaut. Ce dernier nous permet d'utiliser certaines méthodes dans les templates pour nous simplifier la vie : lien vers d'autres pages, import d'autres templates... Nous allons voir tout cela en détail dans ce chapitre.
Nous avons réussi à afficher le template index.html et utiliser une feuille de style. Mais nous pourrions faire mieux ! Le template n'est pas qu'un fichier HTML classique car vous avez accès à de nombreux objets qui vous faciliteront la vie.
Vous avez déjà utilisé la méthode url_for( ) qui fait partie des objets disponibles dans le template pour la construction d'urls. Aussi, vous savez que config est un dictionnaire comprenant la configuration de l'application (dans notre cas, les éléments qui se trouvent dans config.py). Voici les autres :

- request : permet de connaître les détails de la requête HTTP reçue.
- session : la session actuelle. Vous pouvez en effet stocker des valeurs dans la session pour la retrouver ultérieurement. Il s'agit d'une pratique très courante pour authentifier un utilisateur et se souvenir de son passage.
- g : variables globales.
- get_flashed_messages( ) : permet de récupérer les messages flash stockés dans la session, souvent utilisés pour afficher des messages d'information (succès, erreur, etc.) à l'utilisateur après une redirection.

Nous y reviendrons plus tard.

Héritage de templates

A l'intérieur d'un même site, il est préférable que toutes les pages utilisent la même charte de présentation. Vous remarquez aisément qu'une partie du code se répète dans nos templates. Pour ce faire, on utilise un template de base qui contient le squelette de la charte de présentation, puis on spécialise ce template pour chaque type de pages du site. Considérons notre template index.html ; Nous allons le scinder en 2 parties :

- base.html contenant le template de base :

```
<html>
        <head>
                <title>{{title}}</title>
                <link rel="stylesheet" href="{{ url_for ('static', filename ='style.css') }}">
        </head>
        {% block main %} {% endblock  %}
</html >
```

- et index.html qui spécialise ce template de base pour notre page index.

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

{% block main %}
<body >
        <h1>Bienvenu {{name}} !!</h1>
</body >
{% endblock %}
```

Vérifiez que cela fonctionne comme avant. Qu'avons nous fait ? Jinja2 nous permet de créer un template que l'on pourra étendre à volonté, comme une classe enfant qui hériterait d'une classe parent : vous pouvez changer les valeurs par défaut du template parent. C'est assez simple !

Le template base.html contient le squelette de toutes nos pages. Dans notre cas, la section head mais on pourrait y rajouter un menu de navigation, un footer, etc. Nous y reviendrons plus tard. Le template index.html contient alors ce qui sera amené à changer dans le template d'origine : le contenu du body.

Nous indiquons que nous souhaitons étendre base.html dans le template index.html en utilisant le mot-clé extends "base.html". Le template parent agit un peu comme un texte à trou : nous indiquons les endroits qui seront amenés à changer en utilisant le mot-clé {% block nom_du_bloc %} {% endblock %}. Jinja2 sait maintenant que du contenu peut être inséré à cet endroit. Dans le template enfant, index.html, nous englobons le contenu à envoyer dans le même bloc (ici le nom du bloc est main).

Vous pouvez créer autant de blocs que vous le voulez ! Il vous suffit de changer le nom du bloc.

C'est à vous de jouer !

Je vous laisse en faire de même pour les templates about.html et contact.html.

En résumé, Jinja est un moteur de template. Il permet d'écrire du code Python dans un template (en plus du HTML). Pour indiquer à Jinja qu'on utilise des objets Python, il faut utiliser les syntaxes{{ }}, {% %} ou {{# #}} selon le code écrit. Pour insérer des objets Python dans un template, il faut les passer en paramètres de la fonction render_template au niveau de la vue. Jinja permet d'étendre des templates grâce à l'héritage. Pour éviter de dupliquer du code, vous pouvez importer des templates HTML au sein d'un autre template. Les templates de votre projet utilisent désormais Jinja. Passons maintenant à la dynamisation de vos pages.

Créez une page grâce aux URL dynamiques

Notre page index.html est bien jolie mais il est grand temps de la rendre dynamique ! Il va s'en dire que notre internaute ne s'appellera pas toujours Cricri ! Il faudrait donc adapter le message au prénom de l'internaute. Vous devez par conséquent récupérer le prénom. Il faut donc passer ce prénom en paramètre de l'url. Par exemple :

localhost:5000/index?name=Cricri

Décomposons cette url :

- localhost:5000 : nom du domaine et port utilisé.
- /index: chéma utilisé pour identifier la route.
- ? : séparateur entre le schéma et les paramètres.
- name=Cricri : association entre le nom du paramètre (name) et sa valeur (Cricri).

C'est ce que nous appelons une URL dynamique car la valeur du paramètre est amenée à varier, voire même un paramètre inexistant. Pour récupérer ce paramètre optionnel de notre URL dynamique, Flask contient l'objet request qui représente la requête exécutée par le client.

Il contient de nombreux paramètres intéressants :

- request.path renvoie la route demandée
- request.method renvoie la méthode HTTP utilisée
- request.args renvoie un dictionnaire contenant les paramètres présents dans l'URL.

Nous allons utiliser request.args dans la vue index( ) du fichier views.py :

```
from .app import app
from flask import render_template, request

@app.route('/')
@app.route('/index/')
def index():
        # si pas de paramètres
        if len(request.args)==0:
                return render_template("index.html",title="R3.01 Dev Web avec Flask",name="Cricri")
        else :
                param_name = request.args.get('name')
                return render_template("index.html",title="R3.01 Dev Web avec Flask",name=param_name)
```

Voilà pour ce qui est des paramètres optionnels. La méthode render_template( ), que nous utilisons dans la vue, est un peu spéciale. Vous pouvez lui donner en paramètres toutes les variables que vous souhaitez utiliser dans le template.

Utilisation des données

Pour rappel, le fichier data.yml est placé dans le sous-répertoire monApp/data. Vous devez récupérer sur CELENE le fichier images.tar contenant les images des premières de couverture des livres. Décompressez le fichier et placez les images dans monApp/static/images. Les données alimentent les modèles utilisés par notre application que nous avons plaçé conventionnellement dans le fichier monApp/models.py. En Flask avec SQLAlchemy, la récupération de données doit se faire dans la vue.

Le nom de vos routes doit être parlant ! Vous et l'utilisateur devez être en mesure de comprendre, en lisant l'URL, l'action qui sera effectuée. Par exemple, évitez d'utiliser une route qui aurait cette structure : /liste_des_auteurs. Préférez l'url suivante : /auteurs/. Il s'agit d'ailleurs d'une des bonnes pratiques les plus importantes à respecter lorsque l'on crée une application web : l'interface uniforme de l'architecture REST (acronyme pour **RE**presentational **S**tate **T**ransfer). Cette architecture impose six contraintes que vous pouvez choisir de respecter :

- Séparation client / serveur
- Stateless (sans état)
- Cacheable (cachable)
- Layered system (système à plusieurs couches)
- Uniform interface (interface uniforme)
- Code on demand

Comment appliquer REST dans un projet Flask ? La question demanderait un cours entier ! Intéressez-vous en premier à l'interface uniforme et prenez le temps de bien comprendre comment créer de bonnes URL. Modifions notre vue views.py en conséquence :

```
from flask import render_template
from monApp.models import Auteur

@app.route('/auteurs/')
def getAuteurs():
        lesAuteurs = Auteur.query.all()
        return render_template('auteurs_list.html', title="R3.01 Dev Web avec Flask", auteurs=lesAuteurs)
```

Nous nous apprêtons donc à renvoyer au template auteurs_list.html une liste d'auteurs lesAuteurs dans son gabarit auteurs. Dans le template auteurs_list.html, nous allons afficher les auteurs sous la forme d'un tableau. Ce sera l'occasion pour nous de manipuler les blocs suivants du moteur de template Jinja2 :

- {% if %}...{% endif %}
- {% for %}...{% endfor %}

Voici notre template auteurs_list.html :

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

{% block main %}
	<h1>Liste des auteurs</h1>
	<p>Nombre d'auteurs : {{ auteurs|length }}</p>
	{% if auteurs|length == 0 %}
		<p>Aucun auteur trouvé.</p>
	{% endif %}
	<table>
		<thead>
			<tr>
				<th>ID</th>
				<th>Nom</th>
				<th>Nombre de livres</th>
			</tr>
		</thead>
		<tbody>
		{% for unAuteur in auteurs %}
			<tr>
				<td>{{ unAuteur.idA }}</td>
				<td>{{ unAuteur.Nom }}</td>
				<td>{{ unAuteur.livres.count() }}</td>
			</tr>
		{% endfor %}
		</tbody>
	</table>
{% endblock %}
```

Notez au passage la différence entre auteurs|length et unAuteur.livres.count( ). Les deux expressions servent à compter, mais pas du tout la même chose et fonctionnent différemment :

- dans le premier cas, auteurs est une liste Python issue de la requête Auteur.query.all(). Jinja2 appelle len(auteurs) pour connaître combien d'auteurs nous avons dans notre vue. auteurs|length fournit donc le nombre d'éléments dans la variable auteurs.

- dans le second cas, la relation SQLAlchemy a été déclarée avec lazy='dynamic' dans le modèle, ce qui fait que unAuteur.livres est une AppenderQuery (ou query dynamiquement construite) et non une liste. La méthode count( ) envoie une requête SQL du type SELECT COUNT(*) FROM livres WHERE auteur_id = ? pour connaître combien de livres possède chaque auteur, sans charger tous les objets Livre en mémoire. unAuteur.livres.count( ) fournit donc le nombre de livres associés à un seul auteur (unAuteur).

Terminez le travail avec un peu de css pour faire moins moche :

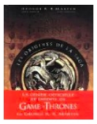
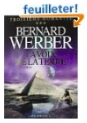
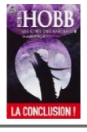
```
td, th{
        padding: 0.5em;
        text-align: center;
        border-collapse:initial;
        border: thin solid black;
}
```

C'est à vous de jouer !

Voyons si vous avez tout compris. Je vous laisse nous présenter la liste des livres dans une autre vue. Voici à quoi elle doit ressembler :

Liste des livres

Nombre de livres : 100

ID	Titre	Prix	Url	Première de Couverture	Auteur
1	La Compassion du Diable	19.9	http://www.amazon.fr/La-Compassion-Diable-Fabio-Mitchelli/dp/2954271078		Fabio M. Mitchelli
2	Le Trône de fer l'Intégrale (A game o...	22.9	http://www.amazon.fr/Tr%C3%B4ne-lInt%C3%A9grale-game-Thrones-Tome/dp/2756415936		George R-R Martin
3	Game of Thrones : les origines	39.95	http://www.amazon.fr/Game-Thrones-George-R-R-Martin/dp/2364802709		George R-R Martin
4	La Voix de la Terre	22.9	http://www.amazon.fr/La-Voix-Terre-Bernard-Werber/dp/2226259880		Bernard Werber
5	Les Cités des Anciens, Tome 8 : Le pu...	6.7	http://www.amazon.fr/Les-Cit%C3%A9s-Anciens-Tome-dargent/dp/2290085987		Robin Hobb

Utilisation de Bootstrap

Bootstrap est un Framework HTML+CSS+JS pour créer des sites web responsive. Il vous appartient de vous former vous-même à l'usage de bootstrap. Cet exercice vous montre simplement comment installer bootstrap et adapter le template de base à son usage :

```
$ pip install Flask-Bootstrap5
```

Par défaut, le plugin sert les fichiers de bootstrap en renvoyant aux sites officiels. Le plugin contient aussi des copies de ces fichiers. Si on veut qu'il nous serve ces copies locales (par exemple si on n'a pas accès au net), il faut rajouter la config suivante dans `config.py` :

```
BOOTSTRAP_SERVE_LOCAL = True
```

Pour activer le plugin, dans le fichier `app.py` nous ajoutons :

```
from flask_bootstrap5 import Bootstrap
Bootstrap(app)
```

Modifions maintenant le template de base. Pour cela nous pouvons nous appuyer sur le code des examples proposés sur *http://getbootstrap.com/getting-started/*. Seulement 2 fragments nous intéressent :

- la barre de navigation,
- le div du contenu principal.

J'ai copié-collé ci-dessous le code correspondant dans les blocks appropriés. Voici le template `base.html` :

```
<html>
	<head>
		<meta charset="UTF-8">
		<meta name="viewport" content="width=device-width, initial-scale=1.0">
		<title>{% block title %}{% endblock %}</title>
		<link rel="stylesheet" href="{{ url_for('static', filename='css/style.css') }}">
		<!-- Bootstrap CSS -->
		<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css" rel="stylesheet">
	</head>

	<body>
		{% block navbar %}
		<nav class="navbar navbar-expand-md navbar-dark bg-dark fixed-top">
		    <div class="container-fluid">
		        <a class="navbar-brand">MaBibliothèque</a>
		            <div class="collapse navbar-collapse" id="mainNavbar">
		                <ul class="navbar-nav me-auto mb-2 mb-md-0">
		                    <li class="nav-item"><a class="nav-link" href="{{ url_for('index') }}">Home</a></li>
		                    <li class="nav-item dropdown"><a class="nav-link dropdown-toggle" href="#"
		                        id="auteursDropdown" role="button" data-bs-toggle="dropdown"
		                        aria-expanded="false">Auteurs</a>
		                          <ul class="dropdown-menu" aria-labelledby="auteursDropdown">
		                              <li><a class="dropdown-item"
		                                      href="{{ url_for('getAuteurs') }}">Liste des auteurs</a>
		                              </li>
		                          </ul>
		                    </li>
		                    <li class="nav-item dropdown"><a class="nav-link dropdown-toggle" href="#"
		                        id="livresDropdown" role="button" data-bs-toggle="dropdown"
		                        aria-expanded="false">Livres</a>
		                          <ul class="dropdown-menu" aria-labelledby="livresDropdown">
		                              <li><a class="dropdown-item"
		                                      href="{{ url_for('getLivres') }}">Liste des livres</a>
		                              </li>
		                          </ul>
		                    </li>
		                    <li class="nav-item"><a class="nav-link" href="{{ url_for('about') }}">À propos</a>
		                    </li>
		                    <li class="nav-item"><a class="nav-link" href="{{ url_for('contact') }}">Contact</a>
		                    </li>
		                </ul>
		            </div>
		        </div>
		    </nav> {% endblock %}
```

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

            <!-- Bootstrap JS Bundle with Popper -->
            <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/js/bootstrap.bundle.min.js"></
script>

        </body>
</html>
```

Vérifions si ça marche : . . . en fait le h1 des templates enfants est caché derrière la barre de navigation. Il faut ajouter un petit décalage vertical dans le CSS pour l'élémemt `<main>` du block `content`. Pour que cela soit facile, nous ajoutons un `id` à cet élément dans le template :

```
<main id="main" role="main" class="container">
```

Puis dans static/style.css nous ajoutons :

```
#main {
    margin - top: 5em;
}
```

Si la police de caractère est moche, on peut aussi la changer :

```
body {
    font - family: sans - serif;
}
```

Ajoutez également une classe bootstrap pour que les tableaux des templates auteurs_list.html et livres_list.html soient plus sympas visuellement :

```
<table class="table table-striped table-bordered table-hover">
```

Vérifiez que tout fonctionne.