Flask

Du framework vers le HTML … ou l’inverse !

Compétences

- **Concevoir et développer des composants d'interface utilisateur en intégrant les recommandations de sécurité**
  - Maquetter une application
  - Développer des composants d’accès aux données
  - Développer la partie front-end d’une interface utilisateur web
  - Développer la partie back-end d’une interface utilisateur web
- **Concevoir et développer la persistance des données en intégrant les recommandations de sécurité**
  - Développer des composants dans le langage d’une base de données
- **Concevoir et développer une application multicouche répartie en intégrant les recommandations de sécurité**
  - Collaborer à la gestion d’un projet informatique et à l’organisation de l’environnement de développement
  - Concevoir une application
  - Développer des composants métier
  - Construire une application organisée en couches
  - Préparer et exécuter le déploiement d’une application

Objectifs

Objet : concevoir et développer une application web en utilisant le micro-framework Flask.

Ze Project

Ze Projekt

https://docs.google.com/document/d/1AFEC9vtPFqVD5eO2oAmS7-63BvAOj-vzatEiZBAg_Ag/edit?usp=sharing

Conception

Créer son environnement de travail

- Logiciel de conception : MindMup, Diagrams.net, Google Doc
- Logiciel de gestion : Trello
- Logiciel de planification : Gantt Project
- Editeur de code : Vs Code
    - extensions à installer :
        - **Mithril Emmet** : aide à l'écriture de code HTML
        - **Jupyter**
        - **SQLIte Viewer**
- Versionning : Git et GitHub
- Déploiement : Heroku

Organisation du projet : vocabulaire

- Le rôle des parties prenantes :
  - AMOA
  - MOA
  - MOE
  - AMOE
- Quelques moments de la gestion de projet
  - MOM
  - VABF
  - VSR
  - Maintenance : corrective, évolutive, préventive
- Quelques structures de la gestion de projet :
  - CPF (Chef de Projet Fonctionnel) / CPI (Chef de Projet Informatique)
  - CoDir
  - CoPil
  - CoTech

Organisation du projet : le cahier des charges

Le cahier des charges est un document texte qui a pour objectifs :

- de clarifier la demande du client afin de bien identifier son besoin ;
- de cadrer cette demande afin d'identifier :
  - le temps nécessaire => calendrier
  - les ressources nécessaires => compétences et environnement de travail
  - le coût.

Un cahier des charges peut être écrit par le client, un sous-traitant ou le prestataire.

Organisation du projet : le RGPD

Tout développement qui concerne des données à caractère nominatif, sensible ou confidentiel, doit respecter le RGPD. Dès le début du projet, il convient de prendre ce référentiel en compte.

Ainsi, 3 points sont à étudier :

1. Quelles sont les données de l'application concernées par le RGPD ?
2. Quelles sont leurs finalités : pourquoi sont-elles utilisées ?
3. Quels sont les traitements réalisés par l'application sur ces données.

La CNIL a publié toutes une série de documents sur ces points, notamment un guide à l'attention des développeurs.

Organisation du projet

La planification d'un projet permet :

- de savoir quand le projet commence et quand il se termine ;
- d'avoir une idée précise des actions qui seront réalisées ;
- de connaître l'articulation entre les actions ;
- de savoir qui fait quoi.

Un diagramme de GANTT peut alors être réalisé. GANTT répons au formalisme suivant :

- Phases
- Etapes
- Actions / Jalons

Il peut être complété par une **Matrice RACI**.

Organisation du projet : SCRUM

Scrum est une méthode agile. Elle se base sur le principe que tant que le client n'a pas validé un livrable, on n'avance pas (ou peu) sur les autres livrables. Cette méthode privilégie ainsi l'échange entre les parties prenantes d'un projet (client - prestataire).

Nous allons pratiquer 4 outils de l'agilité :

1. le kanban
2. le user story
3. le daily scrum
4. le planning poker

Organisation du projet : SCRUM

Pour gérer certaines de ces outils, nous allons utiliser Trello.

1. Créez-vous un compte sur Trello
2. Créez un nouveau tableau

Organisation du projet : SCRUM

Nous allons créer un kanban constitué des 6 colonnes suivantes :

- Backlog
- A faire
- En cours
- Réalisé
- Validé
- Abandonné

D’autres méthodes de nommage et de classement existent, par exemple la MOSCOW : Must, Should, Could, Will not.

Organisation du projet : SCRUM

Nous allons identifier certaines actions à réaliser en termes de :

- conception
  - formalisation : UML, GANTT
  - fonctionnalités : user stories
  - technologie : langage, framework, base de données
- réalisation
  - développement
  - bugs
  - suivi
  - évolution

Organisation du projet : GANTT

Afin d'avoir une vision globale du projet, nous allons réaliser un diagramme de GANTT avec GANTT Project.

Conception : UML

Nous allons voir 4 diagrammes UML :

- **Le diagramme d'étude de cas (use case)** : il permet de formaliser les fonctionnalités dans leur globalité et d'indiquer qui peut y avoir accès. Il montre aussi les relations qui existent entre fonctionnalités.
- **Le diagramme d'activité** : il représente une transaction. Il se rapproche de l'algorithme et permet d'anticiper les cas particuliers à gérer.
- **Le diagramme d'architecture** : il présente les composants de l'application.
- **Le diagramme de navigation** : il montre les types de page d'une application ainsi que les liens qui existent entre elles.

Réalisation de la maquette

2 types de maquettes peuvent être conçues :

- le wireframe
- le mockup

Nous allons concevoir le mockup de l'application en portant une attention sur :

- les composants pérennes des pages
- les composants contextuels aux pages
- les pages intermédiaires
- la gestion des erreurs

Conception du dictionnaire

En amont de la conception puis de la réalisation de la base de données, il convient de bien identifier les données qui seront traitées.

Dans ce cadre, la définition d'un dictionnaire est nécessaire.

Ce dictionnaire va nous permettre d'identifier comment structurer notre base de données :

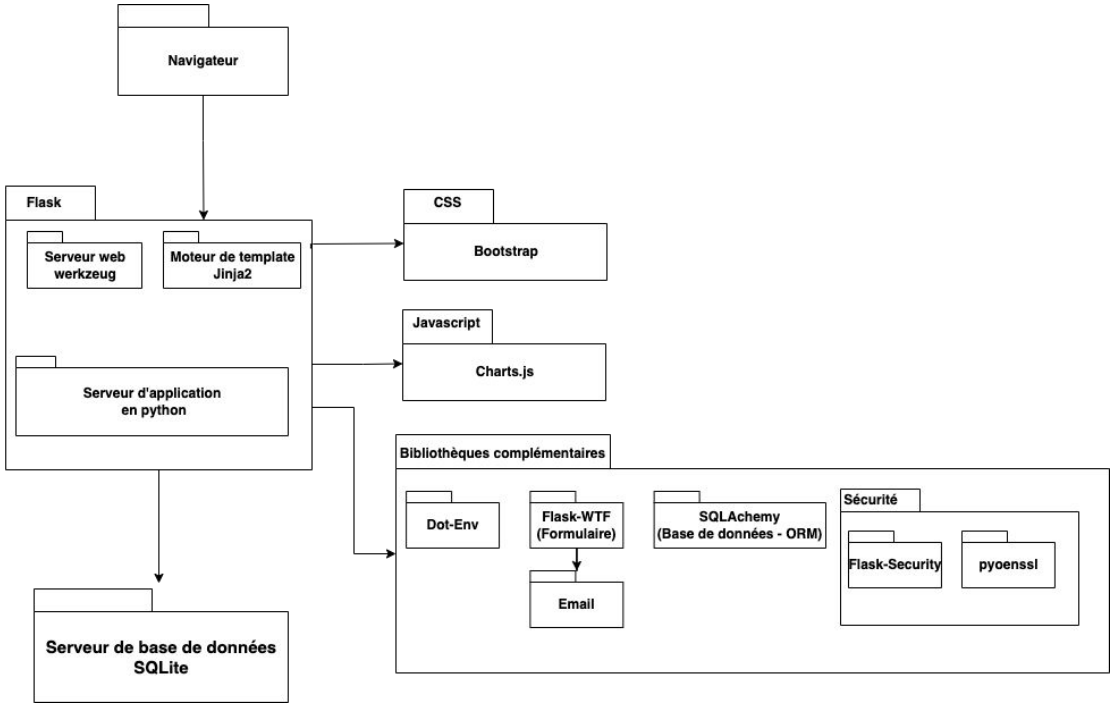
- nombre de tables
- type de données

Livrables réalisés

- ✓ Kanban sous Trello
- ✓ Matrice RACI
- ✓ Diagramme de GANTT
- ✓ UML : use case
- ✓ UML : diagramme d’activité
- ✓ UML : maquette
- ✓ Codebook

Architecture d’une application

Flask et compagnie



Création de pages

Installation et paramétrage

- Sur son disque dur :
  - Créer un dossier appelé **Python**
    - Créer un sous-dossier appelé **Flask**
      - Créer un sous-dossier appelé **Formation**
- Créer un environnement virtuel :
  - **python3 -m venv venv**
- Installer Flask :
  - **pip install flask**
- Créer un fichier requirements.txt :
  - **pip freeze > requirements.txt**

1ers pas

Ma 1ère page

Créez un fichier appelé **app.py** et copiez le code suivant :

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def f_index():
    return "Bonjour ! "
```

Pour exécuter ce code, dans le terminal :

1. Nous allons activer le mode Debug. Ce mode évite d'avoir à relancer le serveur à chaque modification.
   a. **export FLASK_ENV=development / sous Windows : set FLASK_ENV=development**
   b. **export FLASK_APP=app.py / sous Windows set FLASK_APP=app.py**
2. Nous allons lancer l'application :
   a. **flask run** ou **flask --app app run --debug**

En avant … run !

Une autre possibilité pour lancer l'application est d'utiliser :

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

Puis de lancer via le prompt :

python app.py

Terminal : lancement

Si tout se passe bien, sur votre terminal vous devriez avoir les messages suivants qui apparaissent :

```
* Serving Flask app 'app.py' (lazy loading)
* Environment: development
* Debug mode: on
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 000-000-000
```

Il est possible d'y afficher aussi du code :

L’URL : analyse

Pour accéder au site, il faut aller à l’adresse :

- **http** : protocole
- **127.0.0.1** : adresse IP
- **5000** : port

Un ordinateur a une seule adresse IP. Cependant, cet ordinateur peut avoir plusieurs applications avec lesquelles d’autres ordinateurs veulent communiquer (une application web, une messagerie, une application métier, une messagerie instantanée, …). Le port permet de différencier l’accès à ces applications.

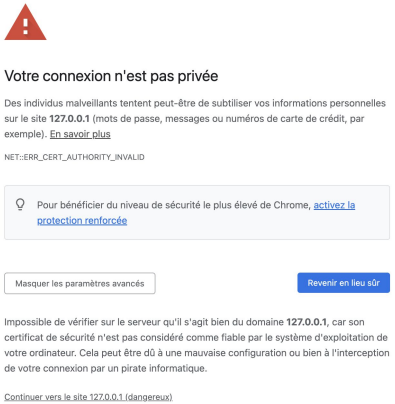
Il est possible de modifier le port en tapant :

flask run --port=6000

L'URL : httpS

Il est possible de simuler une connexion sécurisée avec Flask.

1. Installez pyopenssl :
    a. pip install pyopenssl
2. Pour lancer Flask, tapez :
    a. flask run --cert=adhoc



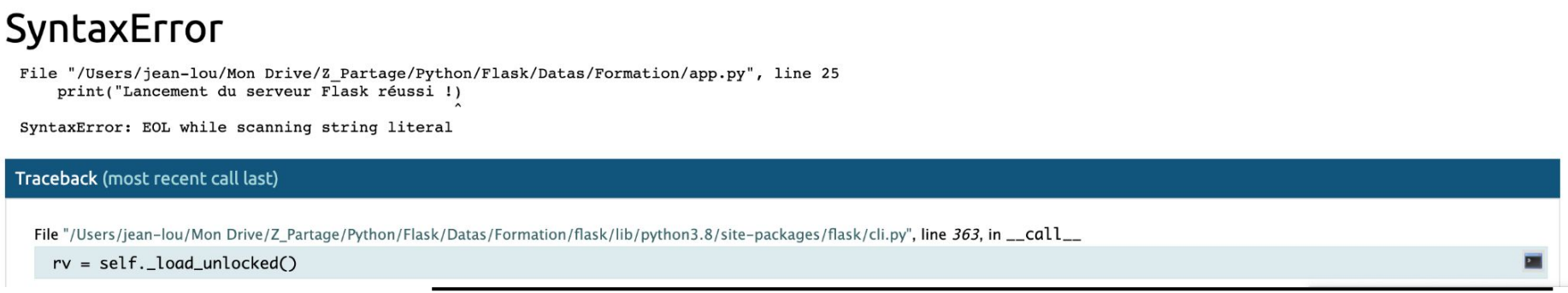
Remarque : votre navigateur va vous indiquer que la connexion n'est pas sécurisée. En effet il ne trouve pas de certificat sécurisé. Mais il est tout de même possible de passer outre en cliquant sur **Continuer vers le site**.

Ressource : document de l'ANSI expliquant les recommandations liées au SSL.

Terminal : le code PIN

Le code PIN vous permet de debugger un code directement dans le navigateur. Pour cela :

1. pointez le curseur de votre souris en fin d'une des lignes d'erreur. Un moniteur apparaît.
2. cliquez sur le moniteur : une fenêtre apparaît vous demandant le code PIN.
3. Saisissez-le. Une nouvelle ligne avec marqué [console ready] apparaît.



Exercice 1

- Créez une nouvelle page que vous allez nommer **qui-suis-je**
  - Cette nouvelle page doit afficher votre nom et votre prénom
- Affichez cette nouvelle page dans un navigateur

Exercice 1

Correction :

```
@app.route("/qui-suis-je")
def f_qui_suis_je():
    return "Jean-Lou Le Bars"
```

Il est possible de mettre des balises HTML dans le **return** :

```
@app.route("/qui-suis-je")
def f_qui_suis_je():
    return "<H1>Jean-Lou Le Bars</H1>"
```

Le fichier de configuration

Il existe un moyen pour éviter d'avoir à retaper les lignes précédentes pour lancer son environnement.

1. Installer la bibliothèque dot-env : **pip install python-dotenv**
2. Créer un fichier **.env** et insérer les lignes de configuration :

```
DEBUG=True
FLASK_ENV=development
FLASK_APP=app.py
```

3. Insérer les instructions suivantes dans le fichier **app.py** :

```
from dotenv import load_dotenv
load_dotenv()
```

4. Taper **Flask run** et tout se lance automatiquement.

Le fichier de configuration

Un autre aspect de la configuration porte sur Flask en lui-même :

- **La gestion du cache** : le cache est une zone mémoire qui conserve certaines données. Il peut de ce fait y avoir une différence entre le code et son exécution (l'exécution se faisant via le cache). Pour ne pas activer le cache de Flask :

```
app.config["CACHE_TYPE"] = "null"
```

Remarque : pour connaître tous les éléments de configuration :

https://flask.palletsprojects.com/en/2.0.x/config/

Calcul de valeurs

Pour traiter un dataset, une solution simple est de réaliser le traitement dans la route : analysons le code suivant

```
@app.route("/code")
def f_code():
    ma_liste = [[10,12,11],[9,14,1],[10,10,11]]
    df = pd.DataFrame(ma_liste, columns= ['Tom','Tim','Tam'])
    return render_template("t_code.html", t_df = df["Tom"].sum())
```

Pour des raisons de propreté de code, il est conseillé de réaliser le traitement dans une fonction, puis d'appeler cette fonction.

URL et argument

Il est aussi possible d'indiquer un argument dans la route :

```
@app.route("/<v_nom>")
def f_nom(v_nom):
    return f"Bonjour {v_nom}"
```

En termes de nommage :

- v_ : pour indiquer que c'est une variable.
- f_ : pour indiquer que c'est une fonction.

Ce qui nous donne le résultat suivant :

Fonctions pratiques

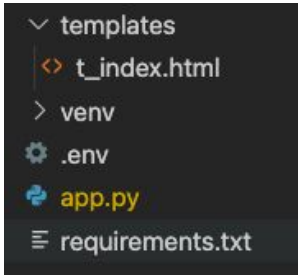
Flask a un shell, qui permet ainsi d'exécuter certaines instructions :

- **flask shell** :commande pour accéder au shell
- **url_map** : liste les routes ainsi que le nom de la méthode.
    - *nom_application.url_map* : dans notre cas, **app.url_map**.

```
Map([<Rule '/formulaire_Inscription' (HEAD, GET, POST, OPTIONS) -> enregistrer_informations>,
 <Rule '/afficher_utilisateurs' (HEAD, GET, POST, OPTIONS) -> afficher_utilisateurs>,
 <Rule '/qui-suis-je' (HEAD, GET, OPTIONS) -> qui_suis_je>,
 <Rule '/' (HEAD, GET, OPTIONS) -> index>,
 <Rule '/static/<filename>' (HEAD, GET, OPTIONS) -> static>])
```

Remarque : il est ainsi possible d'écrire du code Python pour afficher des informations.

Le moteur de templates : Jinja



Jinja est la bibliothèque qui permet de lier des variables d'une page HTML comprise en **{{ variable }}** à un contenu défini dans une route.

Pour cela :

1. créez un dossier **templates**
2. créez un fichier nommé **t_index.html** dans le sous-dossier **templates**
3. tapez dans le fichier :
    a. **!** ou **html:5 :** l'extension **Mithril Emmet** va automatiquement vous implémenter un code HTML.

Nous allons analyser le contenu HTML ainsi généré.

Autopsie d'une page HTML

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width,
        initial-scale=1.0">
        <title>Document</title>
    </head>

    <body>
    </body>
</html>
```

Le moteur de templates : Jinja

Pour faire la liaison entre le fichier**t_index.html** et le fichier **app.py** :

- importer la méthode **rendre_template** :

```
from flask import Flask, render_template
```

- remplacez le **return** de la **route index** par le code suivant :

```
@app.route("/")
def index():
    return render_template("t_index.html")
```

Le moteur de templates : Jinja

Pour écrire un contenu, il est possible de le faire directement dans la page HTML. Nous parlons alors de **contenu statique.**

```
<body>
<h1>Bonjour !</h1>
</body>
```

Le moteur de templates : Jinja

Si nous souhaitons que le contenu soit modifié à partir du code Python (**contenu dynamique**), il faut :

- modifier le contenu de la route, par exemple :

```
@app.route("/")
def f_index():
    v_texte = "Bonjour tout le monde"
    return render_template("t_index.html", t_texte = v_texte)
```

  Pour des raisons de lisibilité, nous préfixons les variables : **v_** pour variable et **t_** pour template.

- ajouter le balisage suivant dans le fichier **index.html** :

```
<body>
<h1>{{t_texte }}</h1>
</body>
```

Le moteur de templates : Jinja

Il est aussi possible d'appliquer des filtres sur les données.

```
<h1>{{ t_texte | upper }}</h1>
```

Ce qui équivaut à :

```
<h1>{{ t_texte.upper() }}</h1>
```

Le moteur de templates : Jinja

Cela peut aussi servir pour afficher le titre d'une page :

```
@app.route("/")
def f_index():
    v_titre = "Accueil"
    v_texte = "Bonjour tout le monde"
    return render_template("t_index.html",
                           t_titre = v_titre,
                           t_texte = v_texte)
```

Dans le fichier **t_index.html** :

```
<title>{{ t_titre }}</title>
```

Type de variable : dictionnaire

Il est aussi possible de stocker les textes à ajouter sous forme de dictionnaire :

```
v_contenu = {
                    "titre" : "Accueil",
                    "texte" : "Application de démonstration"
                    }
```

De les passer dans le return :

```
return render_template("t_index.html",
                       t_contenu = v_contenu)
```

Puis d'appeler les contenus par leur clé dans le fichier HTML :

```
{{ t_contenu.titre }}
{{ t_contenu.texte }}
```

Exercice

Modifiez la route **qui-suis-je** afin que votre prénom et nom s'affichent à partir d'un code du fichier **app.py** dans un fichier HTML que vous nommerez **qui_suis_je.html**.

Questions sur les filtres :

1. Quel filtre permet de mettre en majuscule la 1ère lettre de chaque mot ?
2. A quoi sert le filtre *safe* ? Faîtes un code l'utilisant.
3. Quelle est la différence avec le filtre *striptags* ?

Exercice : correction

- Fichier *app.py*

```
@app.route("/qui-suis-je")
def qui_suis_je():
    v_prenom = "Jean"
    return render_template("index.html", t_prenom = v_prenom)
```

- Fichier *qui_suis_je.html*

```
<body>
    {{ t_prenom }}
</body>
```

Le moteur de templates : Jinja

Il est aussi possible d'utiliser les structures du langage Python pour afficher des données.

- La boucle for sur une liste :
  Fichier **app.py** :

```
@app.route("/qui-suis-je")
def f_qui_suis_je():
    v_prenom = "Jean-Lou"
    v_competences = ["Gestion de projet"  , "Python", "Data"]
    return render_template("t_qui_suis_je.html",
                           t_prenom = v_prenom,
                           t_competences = v_competences)
```

  Fichier **qui_suis_je.html** :

```
<h1> {{ t_prenom }} </h1>
    <br>
    {% for competence in t_competences %}
        <li>{{ competence }}</li>
    {% endfor %}
```

Exercice

1. Comment n'afficher que le 1er élément de la liste afin d'écrire la phrase suivante :
   a. *Il connaît la gestion de projet.*
2. Affichez la phrase suivante si et seulement si le terme Python apparaît dans la liste des compétences :
   a. Il connaît très bien le langage **Python**.

Le moteur de templates : Jinja

Réponse :

1. Comment n'afficher que le 1er élément de la liste afin d'écrire la phrase suivante : *Son rôle principal est chef de projet*.

```
Son langage préféré est le {{ t_competences[0] }}
```

2. Affichez seulement le terme Python en gras, si ce dernier existe dans la liste.

```
{% for competence in t_competences %}
        {% if competence == "Python" %}
        Il connaît très bien le langage<strong>{{competence | upper
}}</strong>.
        {% endif %}
{% endfor %}
```

Les pages d'erreur

Le code suivant permet de gérer un code erreur 404 :

```
@app.errorhandler(404)
def page_introuvable(e):
    return render_template("t_404.html"), 404
```

Exercice :

- Quels sont les codes erreurs qui peuvent souvent apparaître pour une application web ?
- Complétez votre code afin de les résoudre.

Exemple : GIE

Architecture de l'application

Avant de commencer à coder, il faut bien paramétrer son environnement de travail :

- Flask et son éco-système : il convient de réaliser une veille pour identifier en amont les bibliothèques nécessaires et/ou utiles à la réalisation du projet.
- Dossier de l'application : **Application_GIE**
- Nom de l'environnement virtuel : **venv**
- Structuration de l'arborescence :
    - dossier **application** qui contiendra tous les fichiers relatifs à l'application (fichiers Python, HTML, CSS, Js, les images, ….). Ces fichiers seront classés dans des dossiers spécifiques à Flask ou propres à notre organisation.
    - un fichier **run.py** pour lancer l'application.
- Nom de l'application : **gie** (**G**estion des **I**ntervenants **E**xternes)
- Un fichier d'initiation (**__init__.py**) pour créer les instances des composants nécessaires (base de données par exemple).

Architecture de l'application

Il convient aussi de définir une nomenclature pour le nom des éléments qui seront créés :

- Nommage des fichiers :
  - t_ : pour les templates
  - r_ : pour les routes
  - c_ : pour les classes
- Nommage des méthodes :
  - f_ : devant le nom de la méthode
- Nommage pour les variables :
  - b_ : champ de la base de données
  - t_ : variable envoyée vers le template
  - v_ : variable interne à la méthode
  - form_ : variable du formulaire

Tous ces éléments sont à indiquer dans la documentation rédigée pour ce projet.

Mise en place

1. Créez votre environnement virtuel
2. Installez les bibliothèques
3. Créez l'arborescence
4. Créez un fichier **gie.app**. Il contiendra toutes les routes.
5. Créez les fichiers des pages HTML telles qu'indiquées dans l'arborescence.

Remarque : dans un premier temps

- ne vous souciez pas du fichier __init__.py
- ne vous souciez pas du fichier run.py

Structuration des templates

Le moteur de template : hiérarchie

Sous Jinja, il est possible de créer une hiérarchie entre les pages HTML afin de distinguer celle(s) qui contiennent des composants pérennes (en-tête et pied de page par exemple), et celles qui contiennent des informations dynamiques.

Dans ce cadre, nous allons créer une page **base.html** qui contiendra le code HTML de base et l'appellerons dans les autres pages.

Cette page contiendra aussi une partie qui correspondra au bloc de contenu des autres pages.

Remarque : pour connaître et utiliser les mots-clés en HTML, CSS et Js, il existe des cheatsheets, notamment une en ligne.

Le moteur de template : hiérarchie

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>

        EN-TETE<br/>
        {% block content %}

        {% endblock %}
        <p></p>
        PIED DE PAGE<br/>

</body>
</html>
```

Le moteur de template : hiérarchie

Nous allons copier / coller le code HTML précédemment généré dans le fichier *base.html* et le retirer des autres pages.

Ainsi, notre page **t_index.html** ressemble à :

```
{% extends "base.html" %}
{% block content %}
    <h1>{{ t_texte.upper() }}</h1>
{% endblock %}
```

Le moteur de template : les titres

Il est possible de passer des variables dans l'appel à une page HTML :

- Pour modifier le titre d'une page.
  - Dans le fichier **base.html**, entre les balises <HEAD> :

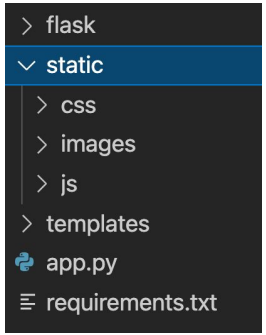
```
{% if t_titre %}
        {{ t_titre }}
    {% else %}
        Application GIE
    {% endif %}
```

  - Dans le fichier de la route :

```
return render_template("index.html",
                       title="Accueil")
```

Eléments de présentation CSS / Bootstrap / Images

CSS



Pour modifier la présentation, il est possible d'utiliser un/des fichiers CSS. Ils doivent être placés dans le dossier **static**.

Pour des raisons de lisibilité, nous allons aussi créer les sous-dossiers **css**, **images** et **js**.

Dans le fichier **base.html**, il faut ajouter le code suivant dans le **HEAD :**

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

CSS

Quelques éléments de syntaxe :

- **balise { propriété; }** : exemple
  - h1
  - { color: red;}
- **id : #id { propriété; }** - exemple
  - #important
  - { font-size: 14px;
  - text-decoration: underline dotted red; }

  Pour l'insérer dans le code HTML :

```
Il connaît très bien le langage<strong id="important">{{competence | upper }}</strong>.
```

CSS

Quelques éléments de syntaxe :

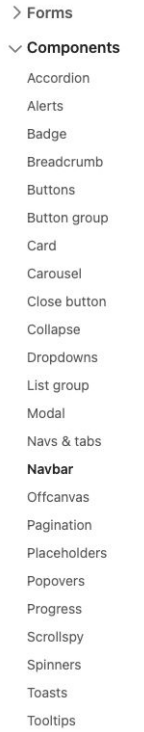
- **class : element.classe { propriétés; }** Une classe cible un élément particulier
    - ```
      h2.marge
      { margin: 2em; }
      ```

    Pour l'insérer dans le code HTML :

    ```
    <h2 class="marge">Erreur 404</h2>
    ```

Pratique : le site codepen vous permet d'essayer des balises HTML / CSS et du Js et de voir directement ce que cela donne.
    ```

Bootstrap



Bootstrap est un framework qui contient des éléments de présentation pré-définis. Il permet ainsi rapidement de pouvoir créer des effets visuels.

1. Allez sur le site de Bootstrap
2. Cliquez sur la partie **Get started**
3. Copiez le code **Include via CDN**. Copier le code css et le code js. CDN (Content Delivery Network) est un système qui stocke un fichier sur un serveur, et nous appelons ce fichier. Il n'y a ainsi pas de stockage en local.
4. Collez ce code dans la partie <head> du fichier **base.html**
5. Encapsulez le code Jinja entre les balises suivantes :

```
<div class="container">
    EN-TETE <br/>
        {% block content %}
        {% endblock %}
        <p></p>
    PIED DE PAGE <br/>
<div>
```

Bootstrap

Nous allons ajouter un menu de navigation afin de pouvoir aller de page en page. Ce menu contiendra ainsi les intitulés :

Accueil **Qui suis-je**

Pour créer les liens afin qu'**Accueil** mène vers la page d'accueil, il faut utiliser la syntaxe suivante :

```
<a class="navbar-brand"
 href="{{url_for('f_index')}}">Accueil<
/a>
```

où **url_for** indique vers quelle route mène le lien.

Bootstrap

Exercice : ajoutez le lien vers la page *Qui suis-je*.

Réponse :

```
<a class="navbar-brand"
 href="{{url_for('qui_suis_je')}}">Qui suis-je
</a>
```

Les images

Exercice :

1. en vous basant sur les manipulations réalisées précédemment pour le CSS, trouvez comment insérer une image sur la page principale.
2. faîtes en sorte que cette image soit dans la barre de navigation et qu'en cliquant dessus on accède à la page d'accueil.
3. créez un pied de page en utilisant Bootstrap.

Les images

Correction :

1. `<img src="{{ url_for('static', filename='images/logo.png') }}">`
2. `<a class="navbar-brand" href="{{url_for('f_index')}}"><img src="{{ url_for('static', filename='images/Logo_Cefim.png') }} " width=50 heigth=50></a>`

Flask-Bootstrap

Il est aussi possible d'installer la bibliothèque Flask-Boostrap.

Pour utiliser les méthodes/fonctions de ce package, par exemple pour un formulaire avec le package WTF :

1. Insérer la ligne suivante dans le fichier HTML du formulaire :

```
{% import "bootstrap/wtf.html" as wtf %}
```

2. Appeler la méthode quick_form avec comme argument le nom de votre formulaire :

```
{{ wtf.quick_form(html_formulaire) }}
```

Structuration des pages

Il est possible de structurer un peu plus la page **base.html**. Ainsi, la partie Navigation peut être écrite dans un fichier **navbar.html** et inclure ce fichier dans le fichier **base.html** :

Le fichier **navbar.html** contient :

```
<nav class="navbar navbar-expand-lg navbar-light"  style="background-color:
#f8d280;">
   <a class="navbar-brand"  href="{{url_for('f_index')}}" >
   <img src="{{ url_for('static', filename='images/Logo_Cefim.png') }} "
        alt="Retour à la page d'accueil"
        title="Logo de CEFIM"
        aria-label="Retour vers la page d'accueil"
        width=70 heigth=70>
   </a>
   <a class="navbar-brand"  href="{{url_for('f_qui_suis_je')}}" >Qui suis-je ?</a>
</nav>
```

Dans le fichier **base.html,** on y insère le code suivant en lieu et place du code précédent :

```
{% include "navbar.html" %}
```

L’accessibilité

L’accessibilité est le fait de rendre accessible les contenus web à des personnes ayant un handicap.Dans ce cadre, il convient de compléter le balisage avec certains attributs :

```
<img src="{{ url_for('static', filename='images/Logo_Cefim.png') }} "
        alt="Image de retour vers la page d'accueil"
        title="Logo de CEFIM"
        aria-label="Retour vers la page d'accueil"
        width=70 heigth=70>
```

Un référentiel a été mis en place pour expliquer les règles à suivre : le référentiel général d’amélioration d’accessibilité.

Projet GIE

Mise en place

- Modifiez votre code afin de bien structurer vos templates.
- Ajoutez un menu en vous servant de Bootstrap.
- Ajoutez un pied de page en utilisant Bootstrap.
- Ajoutez les balises nécessaires pour rendre vos composants accessibles.

Les graphiques

La conception

Nous allons organiser notre code de la façon suivante :

- Création d'un dossier **Analyses**
- Création d'un fichier **analyses.py** : nous y développerons les fonctions d'analyses (avec Numpy et Pandas) et de représentations (avec PlotLy)
- Importation de **Analyses.analyses** dans le fichier **app.py**
- Appel de la/des fonctions de création de graphiques dans une route
- Passage de la variable contenant le graphique au template
- Insertion du graphique dans une balise HTML et Script

Exemple

```
def analyse_kpi():
    donnees = go.Bar(x = ["Tom", "Albert", "Georges"], y = [10, 12, 15])
    graph = go.Figure(data = donnees)
    graph_JSON = json.dumps(graph, cls=plotly.utils.PlotlyJSONEncoder)
    return graph_JSON
```

1. Nous définissons les données et leur mode de représentation.
2. Nous créons le graphe.
3. Nous convertissons ce graphique dans un format compréhensible pour un page HTML : json.
4. Nous retournons le graphique.

Exemple

Nous appelons la fonction dans la route :

```
@app.route("/graphique")
def f_graphiques():
    f_analyse_graphique = analyse_kpi()
    return render_template('t_graphiques.html', t_graphique=f_analyse_graphique)
```

Exemple

Nous intégrons le graphique dans le gabarit HTML :

```
<script src='https://cdn.plot.ly/plotly-latest.min.js'></script>

<div id ="chart">
  <script type='text/javascript'>
    var graphs = {{ t_graphique | safe }};
    Plotly.plot("chart",graphs,{})
  </script>
</div>
```

1. Nous appelons un fichier js en ligne pour afficher le graphique.
2. La balise **div** contient un **id** qui est le nom du graphique de la fonction **Plotly.plot : chart**
3. Nous chargeons la variable puis l'interprétons avec la méthode de PlotLy

Les formulaires

<form>

Les formulaires classiques

```
@app.route('/formulaire', methods=['GET', 'POST'])
def formulaire():
    if request.method == 'POST':
        f_nom = request.form.get('nom')
        f_prenom = request.form.get('prenom')
        return f'''
                <h1>Votre nom : {f_nom}</h1>
                <h2>Votre prénom : {f_prenom}</h2>
                '''
    return '''
            <form method="POST">
                <div><label>Votre nom : <input type="text" name="nom"></label></div>
                <div><label>Votre prénom : <input type="text"
name="prenom"></label></div>
                <input type="submit" value="Envoyer">
            </form>'''
```

Les formulaires classiques

Analyse du code :

- `methods=['GET', 'POST'])`: nous complétons la route afin d'indiquer que les traitements à réaliser concernent ou l'envoi de donnée (GET) ou la réception de données (POST).
- `if request.method == 'POST'` : le programme détecte si il reçoit une information (POST). Le cas échéant, il affiche alors les données saisies dans le champ *input* du formulaire.
- `f_nom = request.form.get('nom')` : nous stockons dans la variable python *f_nom* le contenu du champ HTML dont le nom (*name*) est *nom* (cf. ci-dessous).

Les formulaires classiques

Analyse du code :

- `<form method="POST">` : balise HTML pour afficher un formulaire.
- `<label>Votre nom :` : balise pour afficher l'intitulé du champ du formulaire.
- `<input type="text" name="nom">` : champ à compléter par l'utilisateur.
  - **type** : indication sur le type de données que que contenir le champ. Il existe plusieurs types permettant d'afficher ou de traiter certaines données.
  - **name** : la variable qui contient le nom du champ. Ce nom est communiqué ensuite au programme pour pouvoir traiter la donnée.

Exercice

Modifiez le code précédent en séparant bien le code :

- le code Flask dans le fichier app.py
- le code HTML dans 1 fichier appelé t_formulaire.html
- ajoutez un champ “age” de type nombre.

Correction

Fichier t_formulaire.html

```
{% extends "base.html" %}
{% block content %}
{% if t_nom %}
    <h1>Votre nom : {{ t_nom }}</h1>
    <h2>Votre prénom : {{ t_prenom }}</h2>
    <h3>Votre âge : {{ t_age }}</h3>
{% else %}
    <form method="POST">
        <div><label>Votre nom : <input type="text" name="nom"></label></div>
        <div><label>Votre prénom : <input type="text" name="prenom"></label></div>
        <div><label>Votre âge : <input type="number" name="age"></label></div>
        <input type="submit" value="Envoyer">
    </form>
{% endif %}
{% endblock %}
```

Correction

Fichier app.py

```
@app.route('/formulaire', methods=['GET', 'POST'])
def f_formulaire():
    if request.method == 'POST':
        f_nom = request.form.get('nom')
        f_prenom = request.form.get('prenom')
        f_age = request.form.get('age')
        return render_template("t_formulaire.html",
                               t_nom = f_nom,
                               t_prenom = f_prenom,
                               t_age = f_age)
    return render_template("t_formulaire.html")
```

Exercice : créer un formulaire de contact

Il contiendra les champs suivants :

- Nom : la saisie se fait avec la 1ère lettre en majuscule et le reste en minuscule. Champ obligatoire.
- Prénom : la saisie se fait avec la 1ère lettre en majuscule et le reste en minuscule. Champ obligatoire.
- Mail : champ facultatif.
- Champ Commentaire : champ obligatoire.

Le nom des variables : f_nom, f_prenom, f_mail, f_telephone, f_remarque

- t_nom, t_prenom, t_mail, f_telephone, t_remarque

Le fichier : t_contact.html

Comportement de votre page :

Le formulaire s'affiche si aucune donnée n'a encore été saisie. Utilisez un modèle de formulaire Bootstrap.

- Sinon les informations suivantes s'affiches, formatées de cette manière :

Bienvenue **PRENOM NOM**, voici le commentaire que vous avez laissé : **COMMENTAIRE**

Nous vous rappel, contacterons beaucoup le prochainement au **MAIL** ou par **TELEPHONE**.

*Si le champ mail ou téléphone n'a pas été rempli, affichez : Les champs mails et téléphone sont vides.

WTF !

What the Form

What the Form est une bibliothèque de Flask qui permet de créer et utiliser des formulaires. Elle est à considérer comme un framework dédié à la création de formulaires.

Au niveau de la conception de notre application, nous allons répartir le code de la façon suivante :

1. Un fichier **t_formulaire.html** dans lequel nous allons créer le gabarit du formulaire et y insérer les balises relatives au formulaire créé dans le fichier *formulaire.py*. C'est **la vue**.
2. Une route, dans le fichier **app.py** pour instancier le formulaire du fichier formulaire.py et transmettre les données vers le fichier t_formulaire.html. Nous y développerons aussi les traitements à réaliser. C'est **le contrôleur**.

What the Form

Le **pip install flask-wtf** installe 2 bibliothèques :

- flask-wtf
- WTForms

Dans le fichier **app.py**, nous allons importer :

- **flask-wtf** la méthode **FlaskForm**
- **wtforms** les méthodes **StringField** et **SubmitField** : ces méthodes permettent de créer des types de champs de saisie.

```
from flask_wtf import FlaskForm
from wtforms import StringField, SubmitFiel
```

Remarque : n'oubliez pas de mettre à jour votre fichier**requirements.txt**. Pour ajouter un contenu à un fichier déjà existant :**pip freeze >> requirements.txt**.

What the Form

Nous allons installer une dernière méthode liée au fait que des champs doivent être remplis :

```
from wtforms.validators import DataRequired
```

Il existe plusieurs méthodes de validation des données. Certaines nécessitent d'installer des packages complémentaires, par exemple pour les mails :

pip install wtforms[email]

What the Form

Un formulaire doit être protégé afin que les données saisies dans les différents champs ne nuisent pas au programme.
Parmi les failles qu'il faut prendre en compte : CSRF.

Pour cela, il faut créer une clé secrète :

```
app.config['SECRET_KEY'] = "Ma super clé !"
```

Cette clé secrète permet de s'assurer que les données envoyées vers Flask viennent bien de ce formulaire, et non d'une autre source.

What the Form

Remarque :

- la méthode .config permet de paramétrer certaines propriétés de l'application. Ces paramètres peuvent être stockés dans un fichier de configuration séparé.
- Le package Flask-Security permet de protéger son application Flask.

What the Form

Nous allons ensuite créer une classe de type FlaskForm afin d'indiquer la composition et quelques traitements que le formulaire doit réaliser :

```
class c_Formulaire_enregistrement_informations (FlaskForm):
  wtf_nom = StringField("Nom", validators=[DataRequired()])
  wtf_envoyer = SubmitField("Envoyer")
```

Notre formulaire est ainsi constitué de 2 éléments :

1. un champ de type texte (StringField). Le 1er argument est le label du champ.
2. un champ de type bouton d'envoi (SubmitField)

wtforms contient toute une série de type de champs.

What the Form

Nous créons une route et testons afin de voir si en cliquant sur le bouton envoyer tout se passe bien :

```
@app.route("/formulaire")
def f_formulaire():
    f_formulaire = c_Formulaire_enregistrement_informations()
    print(f"Formulaire créé ! {f_formulaire}.")
    return "Formulaire bien créé !"
```

What the Form

Nous allons créer une page **t_formulaire.html** qui servira pour la route. Nous 'créons' une variable **html_formulaire** qui correspondra à l'instanciation de la classe **c_Formulaire_enregistrement_informations.** Cette variable a comme méthode celles de la classe (wtf_nom et wtf_envoyer).

Entre le `{% block content %}` et le `{% endblock %}` :

```
<H1>Formulaire d'enregistrement d'un intervenant</H1>
      <form method="POST">
          {{ html_formulaire.hidden_tag()}}
          {{ html_formulaire.wtf_nom.label }}
          {{ html_formulaire.wtf_nom() }}
          <br/>
          {{ html_formulaire.wtf_envoyer()}}
      </form>
```

What the Form

Pour transmettre les données vers le formulaire :

```
@app.route("/formulaire_inscription", methods=["GET", "POST"])
def f_enregistrer_informations():
 f_formulaire = c_Formulaire_enregistrement_informations()

 if f_formulaire.validate_on_submit():
    f_nom = v_formulaire.wtf_nom.data
    f_formulaire.wtf_nom.data = ""

 return render_template("t_formulaire_inscription.html" ,
                        titre = "Formulaire d'inscription",
                        html_formulaire = f_formulaire)
```

Remarque : je peux rajouter un **print(f_nom)** afin de vérifier que la valeur saisie dans e formulaire est bien stockée dans la variable f_nom.

What the Form

- `@app.route("/formulaire_inscription", methods=["GET", "POST"])` : GET est une méthode pour recevoir les données et POST pour envoyer des données.
  - `f_formulaire = c_Formulaire_enregistrement_informations()` : nous instancions un formulaire dans une variable appelée f_formulaire. Cette variable est propre à cette fonction. Mais les champs sont ceux de la classe (wtf_nom et wtf_envoyer).
  -
```
if f_formulaire.validate_on_submit():
    f_nom = v_formulaire.wtf_nom.data
    f_formulaire.wtf_nom.data = ""
```
    Lorsque l'utilisateur appuie sur le bouton (SubmitField), f_nom prend la valeur de la variable **wft_nom**, puis nous remettons cette valeur à vide.

Remarque : une autre possibilité pour vider le formulaire des valeurs saisies, juste avant le *return*, écrire :

```
f_formulaire = c_Formulaire_enregistrement_informations(formdata=None)
```

What the Form

- `html_formulaire = f_formulaire)` : la variable html_formulaire est celle qui est utilisée dans le formulaire. Nous indiquons qu'elle prend comme valeur celle de l'objet f_formulaire. Cela permet de lier les champs du formulaire HTML avec ceux de l'objet f_formulaire (wtf_nom et wtf_envoyer).

What the Form

Pour ajouter des classes Bootstrap relatives aux formulaires, il faut les mettre en paramètres :

```
<form method="POST">
            {{ html_formulaire.hidden_tag()}}
            {{ html_formulaire.wtf_nom.label(class="form-label") }}
            {{ html_formulaire.wtf_nom(class="form-control") }}
            <br/>
            {{ html_formulaire.wtf_envoyer(class="btn btn-primary")}}
        </form>
```

Exercice

Reprenez le formulaire et modifiez-le en utilisant WTF. Ce formulaire doit contenir :

- 1 champ Nom
- 1 champ Prénom
- 1 champ Age
- 1 champ Email

Les champs *input* doivent contenir un message indiquant ce qu'il faut remplir.
Les champs Nom, Prénom, Age sont obligatoires.

Une fois le bouton *Envoyer* appuyé, affichez les données sur une page.

Correction

```
from wtforms import StringField, IntegerField, SubmitField, EmailField
from wtforms.validators import DataRequired, Email

class c_Formulaire_enregistrement_informations(FlaskForm):
    wtf_nom = StringField("Nom", validators=[DataRequired()])
    wtf_prenom = StringField("Prénom", validators=[DataRequired()])
    wtf_age = IntegerField("Age", validators=[DataRequired()])
    wtf_email = EmailField('Email', [DataRequired(), Email()])
    wtf_envoyer = SubmitField("Envoyer")
```

Correction

```
if f_formulaire.validate_on_submit():
    f_nom = v_formulaire.wtf_nom.data
    f_prenom = html_formulaire.wtf_prenom.data
    f_age = html_formulaire.wtf_age.data
    f_email = html_formulaire.wtf_email.data
    f_formulaire.wtf_nom.data = ""
    return render_template("t_formulaire.html",
                           t_nom = v_nom,
                           t_prenom = v_prenom,
                           t_age = v_age,
                           t_email = v_email)
```

Correction

```
{% if t_nom %}
    Votre nom : {{ t_nom }}<br>
    Votre prénom : {{ t_prenom }}<br>
    Votre age : {{ t_age }}<br>
    Votre email : {{ t_email }}<br>
{% else %}
<H1>Formulaire d'enregistrement d'un intervenant</H1>
     <form method="POST">
          {{ html_formulaire.hidden_tag()}}
          {{ html_formulaire.wtf_nom.label }}
          {{ html_formulaire.wtf_nom() }}
          {{ html_formulaire.wtf_prenom.label }}
          {{ html_formulaire.wtf_prenom() }}
          {{ html_formulaire.wtf_age.label }}
          {{ html_formulaire.wtf_age() }}
          {{ html_formulaire.wtf_email.label }}
          {{ html_formulaire.wtf_email() }}
```

Un peu de sécurité

Pour se prémunir de certaines failles de sécurité, voici quelques méthodes à ajouter dans le template du formulaire :

- Faille CSRF : `{{ html_formulaire.csrf_token() }}`

Exercice

COncevez puis réalisez une page constituée des champs suivants :

- Nom : obligatoire
- Prénom : obligatoire
- Mail : optionnel
- Téléphone : optionnel
- Raison : liste déroulante contenant les données suivantes Questions, Demandes, Commentaires
- Remarques : obligatoire

Contraintes :

- si les champs Mail et Téléphone n’ont pas été renseignés, indiquez-le dans la page de restitution des informations.

Projet GIE

Mise en place

- Développez une page d'inscription pour un intervenant. Le formulaire doit contenir les champs définis lors de la phase de conception.
- Ajoutez des contraintes au niveau des valeurs saisies :
    - Le nom et prénom doivent être corrects (ils ne peuvent être composés que de lettre de l'alphabet complétées par les caractères ' et -).
    - L'adresse mail doit être valide.
    - La matière enseignée doit être dans une liste.
    - La promotion où la matière est enseignée doit être dans une liste.
    - Le nombre de jours doit être positif et peut être décimal.
    - Le tarif doit être positif et peut être décimal.
- Vérifiez dans la console que les valeurs saisies sont bien prises en compte.

Base de données

SQLite + Flask

SQLite + Flask

Nous allons compléter l'application :

1. transmettre les données du formulaire vers la base de données ;
2. afficher un message indiquant si les données ont bien été enregistrées ou pas ;
3. lister les enregistrements de la base de données.

Créer une base de données

Nous allons créer la base de données grâce

- à une fonction **f_creerLaBaseDeDonnees()**
- qui sera contenu dans un fichier **f_baseDeDonnees.py**
- qui sera stocké dans un dossier **utils**

Pour rappel : le formulaire d'inscription

```
@app.route("/formulaire_inscription", methods=["GET", "POST"])
def f_formulaire():
v_formulaire = c_Formulaire_enregistrement_informations()
if v_formulaire.validate_on_submit():
    v_nom = v_formulaire.wtf_nom.data
    v_prenom = v_formulaire.wtf_prenom.data
    v_formulaire.wtf_nom.data = ""
    flash("Les données ont bien été saisies.")
    return render_template("t_formulaire_enregistrement.html",
                           t_nom = v_nom,
                           t_prenom = v_prenom)
return render_template("t_formulaire_enregistrement.html" ,
                       titre = "Formulaire d'inscription",
                       html_formulaire = v_formulaire)
```

Création de la base

1. nous allons créer la base suivante :

```
import os
import sqlite3
def f_creerLaBaseDeDonnees():
    if os.path.isfile('base_intervenants.db'):
        print("la base existe déjà.")
    else :
        connexion = sqlite3.connect("base_intervenants.db")
        curseur = connexion.cursor()
        curseur.execute("""
                CREATE TABLE intervenants (
                    nom_bdd varchar(100),
                    prenom_bdd varchar(100)
                    )
                """)
        connexion.commit()
        connexion.close()
```

Création d'une route

2. nous insérons la fonction dans le code du fichier principal app.py :

```
f_creerLaBaseDeDonnees()
```

3. nous allons créer une route qui mènera vers une page qui affichera un message indiquant si l'enregistrement s'est bien passé ou non.

```
@app.route('/ajouter_intervenant',methods = ['POST',
'GET'])
def ajouter_intervenant():
```

Enregistrement des données

3. nous essayons de stocker dans 2 variables les données saisies dans les champs du formulaire, puis nous les envoyons vers la base de données :

```
if request.method == 'POST':
    try:
        v_nom = request.form['wtf_nom']
        v_prenom = request.form['wtf_prenom']
        connexion = sqlite3.connect("base_intervenants.db")
        curseur = connexion.cursor()
        curseur.executemany("INSERT INTO intervenants VALUES
(?,?)", [(v_nom, v_prenom)])
        connexion.commit()
        f_message = "Enregistrement inscrit dans la base."
```

Gestion des erreurs

4. En cas de souci :

```
except:
    connexion.rollback()
    f_message = "Un problème est apparu pendant l'enregistrement."
```

5. Au final :

```
finally:
    connexion.close()
    return render_template("t_resultat_inscription.html",t_message = f_message)
```

Modification du template

6. Dans la page du formulaire d'enregistrement, je rajoute une **action**, le code qui sera exécuté une fois le bouton Envoyer appuyé.

```
<form action = "/ajouter_intervenant" method="POST">
            {{ html_formulaire.hidden_tag()}}
            {{ html_formulaire.csrf_token() }}
            {{ html_formulaire.wtf_nom.label(class="form-label") }}
            {{ html_formulaire.wtf_nom(class="form-control") }}
            {{ html_formulaire.wtf_prenom.label(class="form-label") }}
            {{ html_formulaire.wtf_prenom(class="form-control") }}
            <br/>
            {{ html_formulaire.wtf_envoyer(class="btn btn-primary")}}
        </form>
```

Affichage du message

7. La page indiquant le résultat de l'enregistrement :

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

{% block content %}

    <h3>{{ t_message }}</h3>

{% endblock %}
```

Affichage des enregistrements

Exercice :

- affichez les données enregistrées dans la base sur une nouvelle page.
- utilisez la méthode **Flash**.

Correction : Flash

1. Importer la méthode :

```
from flask import Flask, render_template, request, flash
```

2. Ecrire le message dans la route :

```
        flash("Intervenant enregistrés. ")
```

3. Intégrer le message dans la page HTML : nous faisons une boucle car il peut y avoir plusieurs messages selon le résultat du traitement.

```
{% for message in get_flashed_messages() %}
            {{ message }}
{% endfor %}
```

Rappel global

Conception d'un formulaire de contact

Méthode :

1. **Je crée une classe formulaire avec Flask-WTF**
    a. Je donne un nom parlant à la classe :
        i. class **Contact_form**(FlaskForm) :
    b. Je définis des noms de variables parlant :
        i. **fc_last_name** = StringField(''Nom'', validators=[DataRequired()])
        ii. **fc_first_name** …
        iii. ….
        iv. fc_button_send …

Conception d’un formulaire de contact

Les prochaines diapos vont présenter une méthode de conception, avec quelques éléments techniques, pour un formulaire de contact.

Objectif : développer un formulaire de contact et stocker les informations dans une base de données.

Composants du formulaire : 4 champs

- Nom : obligatoire
- Prénom : facultatif
- Mail : obligatoire
- Commentaire : obligatoire

Conception d'un formulaire de contact

2. **Je crée la base de données dans un fichier *bdd_contact_form.py*.**

   La base de données s'appelle : *bdd_contact_form.db*

   a. **Je teste si la base existe déjà :**

      i. => librairie os et méthode isfile

   b. **Si elle n'existe pas, je la crée :**

      i. connexion = sqlite3.connect.("bdd_contact_form.py")

      ii. Je crée les champs pour stocker les données :

         1. bdd_fc_last_name varchar(40)
         2. …..

Conception d'un formulaire de contact

3. **Je crée les 2 pages HTML :**
    a. **La page du formulaire :**
        i. **Je crée une route : *<u>/contact_form</u>***
            1. La route doit réaliser 1 traitement : afficher le formulaire
                a. J'instancie le formulaire défini précédemment (Contact_form)
                    i. f_contact_form = Contact_form()
                b. J'envoie ce formulaire vers la page HTML avec le formulaire
                    i. render_template
                    ii. html_contact_form = f_contact_form
        ii. **Je crée un fichier html : *<u>t_contact_form.html</u>***
            1. J'affiche les labels des champs :
                a. {{ html_contact_form.fc_last_name.label() }}
            2. J'affiche les champs du formulaire :
                a. {{ html_contact_form.fc_last_name() }}

Conception d'un formulaire de contact

4. **Je crée les 2 pages HTML :**
    a. **La page qui affiche si tout s'est bien passé, ou pas !**
        i. **Je crée une route : *<u>/resultat_contact_form</u>***
            1. La route doit réaliser 4 traitements :
                a. Vérifier si les données ont bien été envoyées :
                    i. **if et POST**
                    ii. **try … except … finally**
                b. Le cas échéant, stocker les données dans des variables
                    i. f_last_name = request.form['c_last_name']
                    ii. ….

Conception d'un formulaire de contact

4. **Je crée les 2 pages HTML :**
    a. **La page qui affiche si tout s'est bien passé, ou pas !**
        i. **Je crée une route : *//resultat_contact_form* (suite)**
            1. La route doit réaliser 4 traitements (suite) :
                c. Enregistrer les données dans la base de données
                    i. Je me connecte à la base de données
                    ii. Je crée le curseur pour travailler sur la base
                    iii. J'insère les données : INSERT … f_last_name …
                    iv. J'envoie les données
                    v. Je ferme la base de données
                d. Envoyer un message indiquant si tout s'est bien passé (ou pas)

Conception d’un formulaire de contact

4. **Je crée les 2 pages HTML (suite) :**
    a. **La page qui affiche si tout s’est bien passé, ou pas !**
        ii. **Je crée un fichier html : *t_resultat_contact_form.html***
            1. Cette page affiche juste le message indiquant si les données sont bien inscrite, ou si il y a eu un problème.

Conception d’un formulaire de contact

Une variable demande ainsi 4 nommages :

1. fc_last_name => flask-wtf - conception du formulaire
2. bdd_fc_last_name => sqlite - champ de la base de données
3. html_contact_form.fc_last_name => nom du formulaire HTML
4. f_last_name => variable Python pour traiter la donnée

Projet GIE

Exercice

- Complétez votre programme afin de relier les champs du formulaire à la base de données.
- Afficher toutes les données relatives aux intervenants sur une page.

Javascript

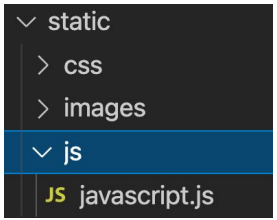
Charts.js : intégration

Nous allons intégrer un graphique de la librairie Charts.js. Cette librairie offre toutes une série d'exemple de graphiques.

A travers son compte GitHub, nous pouvons voir qu'elle est populaire et mise à jour assez souvent.

Javascript : intégration

L'intégration de code Javascript se fait de la même manière que pour le CSS ou une image :



L'appel se fait de la même façon :

```
<script
        src="{{ url_for('static', filename='js/javascript.js') }}">
</script>
```

Javascript : intégration

Pour intégrer un graphique réalisé avec **Charts.js** :

1. Dans le dossier **static/js** : créez un fichier **graphique.js**. Ce fichier contiendra le code javascript de la page : https://www.chartjs.org/docs/latest/ (ne prenez que le code compris entre les balises script **sans les balises**).
2. Dans le template, par exemple **t_index.html**, copiez le code suivant :

```
<div>
    <canvas id="myChart"></canvas>
    <script type="text/javascript"
        src="{{ url_for('static', filename='js/graphique.js')
            }}"></script>
</div>
```

- La balise <canvas> est utilisée. Elle sert à intégrer des graphiques ou des animations dans une page HTML.
- url_for indique où se situe le fichier js à intégrer.

Javascript : intégration

3. Ajouter le script d'exécution dans la fichier **base.html** :
    a. `<script src="https://cdn.jsdelivr.net/npm/chart.js"></script>`

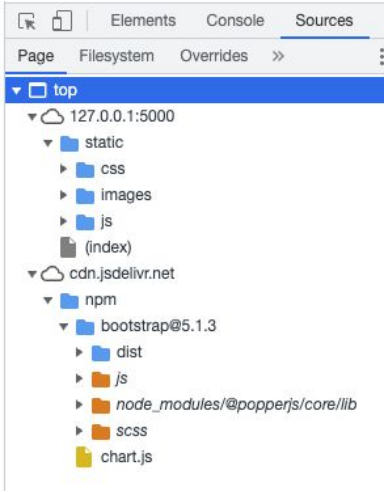
Javascript : intégration

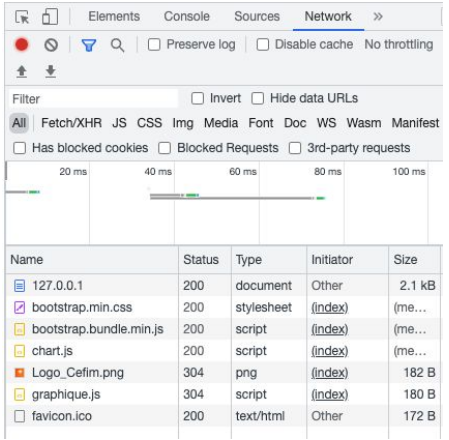
Une petit analyse du code nous permet d'identifier les instructions qui peuvent être modifiée :

- `ctx = document.getElementById('mon_graphique').getContext('2d');`
- `type: 'doughnut',` : les types de graphiques sont indiqués dans la partie Chart Types du site.
- `labels: ["Nombre de jours", "Tarifs"],`
- `label: 'Données sur les intervenants externes' ,`
- `data: [100, 500],`

Analyse des composants d'une page

Il est possible de voir les composants qui se chargent dans une page grâce aux utilitaires des navigateurs. Par exemple, sous Chrome :





Je dé-Pense donc je suis

Nous allons compléter notre application en ajoutant 1 nouveau champ à la base de données : coût.

Exercice : modifier votre programme afin de rajouter ce nouveau champ dans toutes les pages nécessaires.

Flask et Chart.js

Passer des variables vers Charts.js

Pour passer une variable de Flask vers Charts.js, il suffit :

- Créer une fonction avec notre code Js du fichier graphique.js. Cette fonction prend 2 arguments : le label et les valeurs. Pour cela, encapsuler le code Js entre :

```
function graphique(label, valeur) {}
```

- Ecrivez les arguments de la fonction à leur emplacement dans le code :

```
labels: label,
        datasets: [{
            label: 'Données sur les intervenants externes' ,
            data: valeur,
```

Passer des variables vers Charts.js

- Dans le template, ajoutez le code suivant :

```
<script type="text/javascript"
        src="{{ url_for('static', filename='js/graphique.js') }}" >
    </script>
    <script type="text/javascript">
      monGraphique =
              monTresBeauGraphique(
                    {{t_label|tojson}},
                    {{t_cout|tojson}})
    </script>
```

Passer des variables vers Charts.js

Nous allons stocker les coûts de la base de données dans une variable de type tableau (n'oubliez pas que par défaut, les requêtes SQLite renvoie un tuple !) :

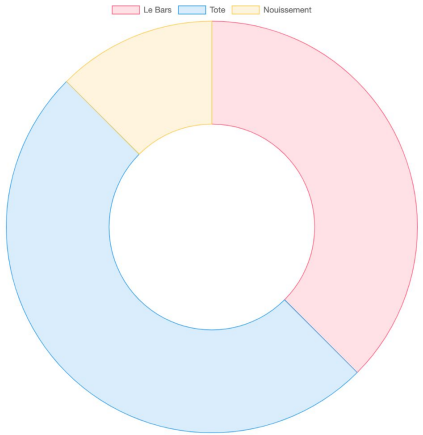
```
couts = curseur.execute("SELECT cout_bdd FROM
intervenants")
  v_tableau_cout = []
  for cout in couts:
      v_tableau_cout.append(cout[0])
```

Transmettre cette variable au template :

```
return render_template("t_tableauDeBord.html",
                       t_label = v_liste_intervenants,
                       t_cout = v_tableau_cout)
```

Exercice

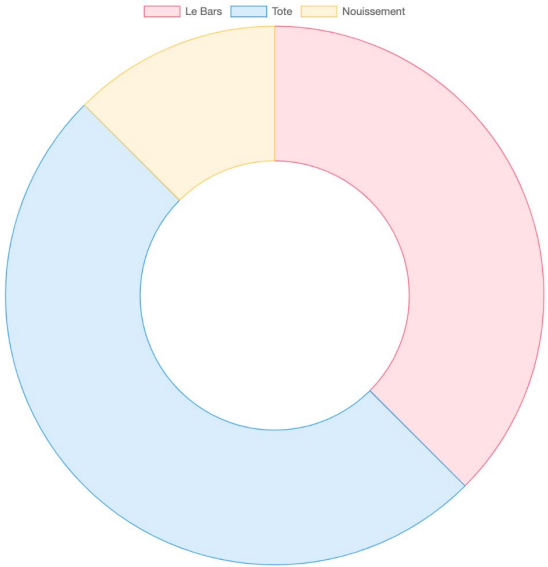
- Affichez la somme des coûts saisis. Réalisez ce calcul en utilisant SQLite. Formatez l'affichage pour qu'il ressemble à :



Au regard du nombre d'intervenants actuellement saisis dans la base de données, si nous réalisons la somme des différents coûts indiqués sous arrivons à **1200 €**.

Exercice

- Affichez la liste des intervenants avec leur coût associé. N'hésitez pas à modifier votre fichier **style.css**.



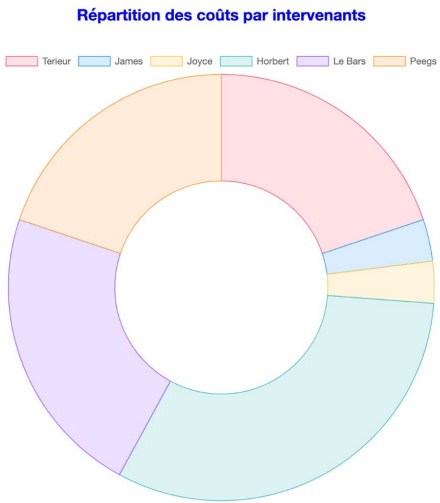
Au regard du nombre d'intervenants actuellement saisis dans la base de données, si nous réalisons la somme des différents coûts indiqués sous arrivons à **1200 €**.

Le tableau suivant liste le cout par intervenant.

Nom	Coût
Le Bars	450
Tote	600
Nouissement	150

Exercice

- Indiquez le nombre d'intervenants.
- Classez les intervenants par coût du plus cher au moins cher.



Il y a actuellement **6 intervenants**.
Au regard du nombre d'intervenants actuellement saisis dans la base de données, si nous réalisons la somme des différents coûts indiqués sous arrivons à **2520 €**.

Le tableau suivant liste le cout par intervenant.

Nom	Coût
Horbert	800
Le Bars	560
Terieur	500
Peegs	500
James	80
Joyce	80

Flask : le retour

Exercice

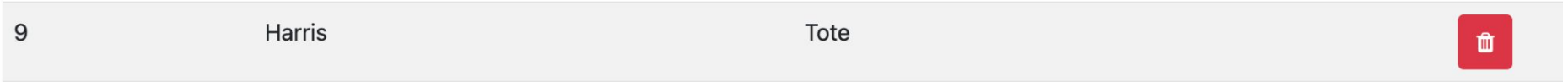
Complétez l'application afin :

- de pouvoir supprimer les enregistrements en passant par une interface en ligne (par exemple, un bouton supprimer).

Exercice

Suppression des enregistrements.

Objectif : cliquer sur un bouton pour supprimer un enregistrement.



Méthode : se servir de l'ID de l'enregistrement pour supprimer la données.

User storie :

- En tant qu'utilisateur
- Je peux pouvoir supprimer un enregistrement en cliquant sur un bouton
- afin d'effacer un enregistrement de la base de données

Exercice

1. Je vais définir une route pour supprimer un enregistrement :
    a. dans l'url, je prends comme argument l'ID de la données à supprimer :

```
@app.route('/supprimer_intervenant/<id>',methods = ['POST', 'GET'])
```

    b. je passe cette ID comme argument de la fonction :

```
def f_supprimer_intervenant(id):
```

    c. je me connecte à la base et exécute une requête pour supprimer la donnée :

```
 if request.method == 'POST':
  connexion = sqlite3.connect("base_intervenants.db")
  curseur = connexion.cursor()
  curseur.execute("DELETE FROM intervenants WHERE rowid=?", id)
  connexion.commit()
 liste_intervenants = curseur.execute("SELECT rowid, * FROM
intervenants")
```

    d. j'affiche de nouveau la liste :

```
 return render_template("t_lister_intervenants.html",
                        t_intervenants = liste_intervenants)
```

Exercice

2. Je vais rajouter un bouton avec l'icone Poubelle sur le tableau qui liste les intervenants. {{ intervenant[0] }} correspond au n° de l'ID.

```
<td>{{ intervenant[2] }}</td>
<td>
<form action="/supprimer_intervenant/{{ intervenant[0] }}" method="POST">
<input type="hidden" value="{{ intervenant[0] }}"/>
<button class="btn btn-danger fa fa-trash"></button>
</form>
</td>
```

Remarque : la classe fa fa-trash permet d'ajouter l'icône poubelle. Pour cela, il faut aussi ajouter dans le fichier base.html le lien suivant :

```
<link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/4.7.0/css/font-awesome.min.css"
```

SQLAlchemy

Principes

- Nous allons utiliser l'ORM SQLAlchemy pour créer et gérer la base de données.
- L'avantage d'un ORM est de rendre l'application indépendante d'un SGBD. Il est ainsi plus aisé de connecter un nouveau système sur l'application.
- Nous sommes sur une architecture web de type n-tier.
- Cette architecture est complétée par un design pattern MVC :
    - **M**odel : la base de données
    - **V**iew : la page HTML
    - **C**ontrol : les traitements réalisés

Dans ce cadre, nous allons refaire la partie base de données.

Le modèle : SQLAlchemy

1. Installer SQLAlchemy : **pip install flask-sqlalchemy**
2. Importer cette bibliothèque dans dans **app.py** :

```
from flask_sqlalchemy import SQLAlchemy
```

3. Indiquer l'emplacement où la base de données est située :

```
app.config["SQLALCHEMY_DATABASE_URI"] =
"sqlite:///maBaseDeDonneesIntervenants.db"
```

Le modèle : SQLAlchemy

Remarque : si vous avez le message d'avertissement suivant

FSADeprecationWarning: SQLALCHEMY_TRACK_MODIFICATIONS adds significant overhead and will be disabled by default in the future. Set it to True or False to suppress this warning.

il faut ajouter la ligne suivante de configuration :

```
app.config["SQLALCHEMY_TRACK_MODIFICATIONS" ] = False
```

Le modèle : SQLAlchemy

5. Création d'une base de données appelée **db** liée à l'application :

```
sqla_baseDeDonnees = SQLAlchemy(app)
```

6. Création du modèle de la base de données : il faut indiquer une clé primaire.

```
class bdd_intervenants(sqla_baseDeDonnees.Model):
sqla_id = sqla_baseDeDonnees.Column(sqla_baseDeDonnees.Integer, primary_key
= True)
sqla_nom = sqla_baseDeDonnees.Column(sqla_baseDeDonnees.String(50), nullable
= False)
sqla_prenom = sqla_baseDeDonnees.Column(sqla_baseDeDonnees.String(50),
nullable = False)
sqla_cout = sqla_baseDeDonnees.Column(sqla_baseDeDonnees.Integer, nullable =
False)
```

Le modèle : SQLAlchemy

7. Pour créer la base de données : insérez

```
sqla_baseDeDonnees.create_all()
```

suite à la classe.

Le modèle : SQLAlchemy

8. Création du formulaire :

```
class c_Formulaire_enregistrement_informations(FlaskForm):
 wtf_nom = StringField("Nom", validators=[DataRequired()])
 wtf_prenom = StringField("Prenom", validators=[DataRequired()])
 wtf_envoyer = SubmitField("Envoyer")
```

Le contrôle :

9. Nous créons la route, la fonction et initialisons les variables :

```
@app.route("/formulaire", methods=["GET", "POST"])
def enregistrer_informations():
    formulaire = Formulaire_enregistrement_informations()
```

Le contrôle :

10. Nous peuplons la base de données. Nous mettons comme condition qu'une même adresse mail ne peut apparaître 2 fois

```
if v_formulaire.validate_on_submit():
    intervenant = baseDeDonnees_sqlalchemy(nom_base =
v_formulaire.wtf_nom.data,
                                           prenom_base =
v_formulaire.wtf_prenom.data)
    baseDeDonnees_sqlalchemy.session.add(intervenant)
    baseDeDonnees_sqlalchemy.session.commit()
```

Le contrôle :

11. Nous instancions les variables qui seront communiquées à la page HTML pour être affichées :

```
prenom_form = formulaire.prenom_form.data
nom_form = formulaire.nom_form.data
age_form = formulaire.age_form.data
mail_form = formulaire.mail_form.data
flash("Utilisateur ajouté !")
```

Le contrôle :

12. Nous réinitialisons les variables.

```
formulaire.prenom_form.data = ''
formulaire.nom_form.data = ''
formulaire.mail_form.data = ''
formulaire.age_form.data = ''
```

Le contrôle :

13. Nous créons une variables qui contiendra tous les utilisateurs inscrits classés par ordre chronologique. L'objectifs est ensuite de l'appeler pour afficher chacun des utilisateurs dans un tableau HTML.

```
utilisateurs_inscrits =
mabase.query.order_by(mabase.date_ajout_base)
```

Le contrôle :

14. Nous transmettons l'ensemble des variables à la page HTML :

```
return render_template("formulaire.html" ,
                       prenom_html = prenom_form,
                       nom_html = nom_form,
                       age_html = age_form,
                       mail_html = mail_form,
                       formulaire_html = formulaire,
                       utilisateurs_inscrits_html =
                                 utilisateurs_inscrits)
```

Le vue : le fichier HTML

15. Nous mettons en place les champs du formulaire :

```
<H2>Merci de compléter le formulaire suivant :  </H2>
        <br/>
        <div class="shadow p-3 mb-5 bg-body rounded" >
        <form method="POST">
            {{ formulaire_html.hidden_tag()}}
            {{ formulaire_html.prenom_form.label(class="form-label") }}
            {{ formulaire_html.prenom_form(class="form-control") }}
            {{ formulaire_html.nom_form.label(class="form-label") }}
            {{ formulaire_html.nom_form(class="form-control") }}
            {{ formulaire_html.age_form.label(class="form-label") }}
            {{ formulaire_html.age_form(class="form-control") }}
            {{ formulaire_html.mail_form.label(class="form-label") }}
            {{ formulaire_html.mail_form(class="form-control") }}
            <br/>
            {{ formulaire_html.envoyer_form(class="btn btn-primary")}}
        </form>
        </div>
```

Le vue : le fichier HTML

16. Nous affichons les utilisateurs inscrits :

```
<table class="table table-striped" >
        <thead>
          <tr>
            <th scope="col">ID</th>
            <th scope="col">NOM</th>
            <th scope="col">PRENOM</th>
            <th scope="col">AGE</th>
            <th scope="col">MAIL</th>
          </tr>
        </thead>
        <tbody>
    {% for utilisateur_inscrit in utilisateurs_inscrits_html %}
            <tr>
                <th scope="row">{{ utilisateur_inscrit.id_base }}  </th>
                <td>{{ utilisateur_inscrit.nom_base }} </td>
                <td> {{ utilisateur_inscrit.prenom_base }} </td>
                <td>{{ utilisateur_inscrit.age_base }} </td>
                <td><a href="mailto:{{ utilisateur_inscrit.mail_base }}" >{{ utilisateur_inscrit.mail_base }} </a></td>
            </tr>
    {% endfor %}
        </tbody>
    </table>
```

Exercice

- Ajoutez une fonctionnalité pour modifier les données.
  - Indices :
    - Servez-vous de l'identifiant pour accéder au bon enregistrement. Votre fonction aura ainsi la forme suivante : **def modifier_utilisateur(id):**
    - Utilisez la méthode **request** de Flask.
- Ajoutez une fonctionnalité pour supprimer les données.
  - Indice :
    - Agissez de la même manière que pour la modification.
    - Utilisez la méthode **delete** liée à la session de la base de données.

Organisation du code

Bibliothèque et configuration

Afin de simplifier le code du programme principal, nous allons le découper et le copier dans d'autres fichiers qui seront appelés :

- mettre les bibliothèques importées dans un fichier bibliotheques.py
- appeler ce fichier en écrivant dans app.py :

```
from bibliotheques import *
```

- compléter le fichier de configuration **config.py** :

```
class Config(object):
    DEBUG = False
    TESTING = False
    SECRET_KEY = "OK"
    SQLALCHEMY_DATABASE_URI = "sqlite:///mabase.db"
    SQLALCHEMY_TRACK_MODIFICATIONS = False
```

- pour appeler ces configurations, dans **app.py** , après la création de l'app :

```
app.config.from_object("config.Config")
```

où **config** est le nom du fichier et **Config** le nom de la classe.

Les classes

- Nous allons regrouper les classes dans un fichier **formulaires.py.**
- Dans le fichier **bibliotheques.py**, nous ajoutons l'import suivant :

```
from formulaire import *
```

Les routes

4. Complétez et modifiez le code comme suit :

```
from bibliotheques import *

page_erreur = Blueprint('page_erreur', __name__)

@page_erreur.app_errorhandler(404)
def page_introuvable(e):
  return render_template("404.html"), 404
```

Remarque : la méthode à utiliser change. Ce n'est plus **errorhandler** mais **app_errorhandler**.

Les routes

Nous allons séparer nos pages dans différents fichiers. Ainsi, le fichier app.py contiendra l'accès à l'index, les autres pages seront dans des fichiers relatifs à leur thématiques (fichier pour les pages d'erreur, fichier pour les pages du formulaires, ….).

Nous allons faire cela pour les pages d'erreur :

1. Nous allons utiliser la méthode **Blueprint** de Flask. dans le fichier bibliotheques.py, ajoutez la méthode **Blueprint** :

```
from flask import Flask, render_template, flash, Blueprint
```

2. Créez un fichier **erreurs.py**
3. Copiez-collez le code des routes d'erreur dedans.

Les routes

5. Dans le fichier app.py, ajoutez les lignes suivantes :

```
from erreurs import page_erreur
…..
app.register_blueprint(page_erreur)
```

Les routes : exercice

Réalisez les mêmes actions pour la page **qui suis-je**.

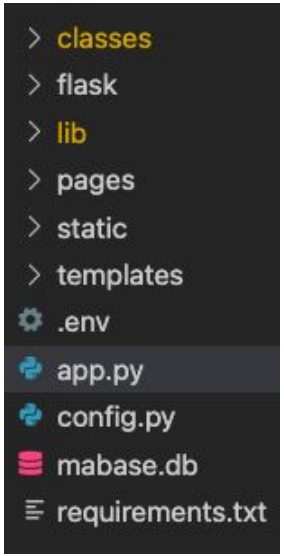
Correction

Une fois les modifications apportées, n'oubliez pas de modifier l'accès à la page dans le fichier **base.html** :

```
<a class="nav-link" href="{{url_for('page_qui.qui_suis_je')}}">Qui suis-je</a>
```

Classement des fichiers

Nous allons créer de nouveaux dossiers pour classer les pages par thèmes :



Gestion du code

Il est aussi pertinent :

- d’identifier les constantes et de les mettre dans un fichier à part.
- de stocker les requêtes SQL dans un fichier à part, soit sous forme de variable, soit sous forme de fonction.

Git et GitHub

Git ?

Git est un **logiciel** qui permet de gérer les **versions** des fichiers d'un projet.

Il convient de bien distinguer Git, le système de versionning, de dépôts en ligne :

- GitHub
- GitLab
- Bitbucket

Ces solutions permettent en effet de gérer des fichiers en ligne.

Git et GitHub

Partir de l'existant

Nous avons déjà toute une arborescence qui existe. Dans ce cadre, il va falloir :

- initialiser Git dans le dossier de ce projet
- faire des add et commit des fichiers existants
- les envoyer vers GitHub

Pour cela, sur le terminal à la racine du dossier du projet :

- **git init** => cela créé un dossier .git dans le dossier.
- **git add .** => j'ajoute tous les dossiers et fichiers présents.
- **git commit -m "Initialisation du projet"** => les dossiers/fichiers sont prêts à être transmis.
    - En cas de modification d'un fichier :
        - **git status** => indique les modifications apportées
        - **git diff** => montre les modifications
    - Pour connaître les actions réalisées : **git log**
- **git config --global user.name "votre nom"**
- **git config --global user.email "votre mail"**

et le transmettre au monde !

Sous GitHub :

1. Créez un dossier. **Ne demandez la création d'aucun fichier.** Si vous demandez la création sur GitHub, il vous faudra d'abord les envoyer vers le dépôt sur votre ordinateur, puis réaliser les instructions suivantes.

Sous votre terminal :

1. Pour lier votre dépôt local à votre dépôt GitHub :
    a. **git remote add origin nomDeVotreDepot**
2. Pour transmettre les dossiers / fichiers :
    a. **git push -u origin master**

.gitignore

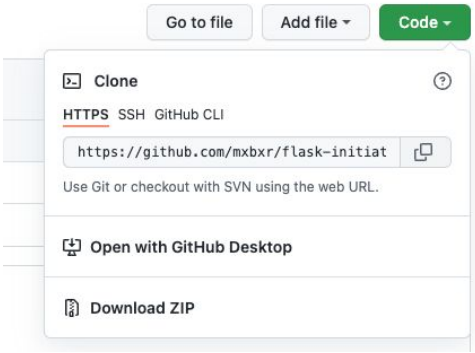
Tout ne doit pas être versionné. Ainsi, l'environnement virtuel ne doit pas être envoyé vers GitHub, de même que pour les bibliothèques installées.
Il convient alors de modifier/compléter le fichier .gitignore pour lui indiquer les dossiers et fichiers à ne pas prendre en compte.

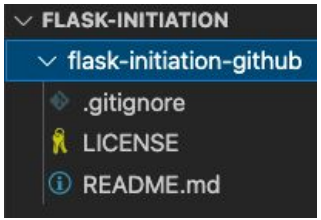
- Ignorer un fichier : nomDuFichier.ext
- Ignorer un dossier : nomDuDossier/

Pour ignorer le dossier de l'environnement virtuel, environnement virtuel appelé **venv** :

- venv/

En avant !





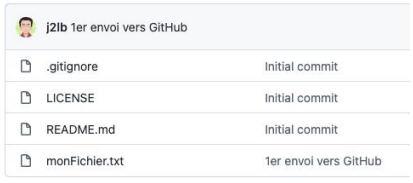
1. Installer Git
    a. Téléchargez et installez Git sur votre ordinateur.
2. Créer un compte GitHub
    a. Créez-vous un compte sur GitHub.
    b. Sous GitHub, créez un **nouveau dépôt / repository : *flask-initiation-github***
    c. Cliquez sur le **bouton vert Code** et copiez le contenu du champ HTTPS
3. Lier votre compte GitHub avec votre projet
    a. Créez un dossier sur votre ordinateur : Flask-initiation
    b. Ouvrez ce dossier sous VS Code
    c. Créez un fichier : monFichier.txt
    d. Ouvrez le Terminal
    e. Tapez **git clone + contenu du champ HTTPS**
        i. Exemple : **git clone https://github.com/mxbxr/flask-initiation-github.git**
        ii. Un dossier *flask-initiation* apparaît avec les fichiers de GitHub. Ce dossier est une copie locale.

En avant !

4. Envoyer un 1er fichier
    a. Dans le dossier *flask-initiation-github*, créez un fichier : monFichier.txt
    b. Dans le terminal, tapez : **git add monFichier.txt**
        i. Cela envoie le fichier vers un espace de transition, le **staging**.
    c. Puis tapez : **git commit -m “1er envoi vers GitHub”**
        i. Un commit permet de joindre un message indiquant par exemple les modifications apportées.

```
[main f77541c] 1er envoi vers GitHub
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 monFichier.txt
```

    d. Puis tapez : **git push**
        a. Envoi le fichier vers GitHub
        b. Si vous allez sur GitHub, le fichier doit apparaître.



En avant !

- Pour retirer un fichier d’un **git add** :
  - **git reset nomDuFichier**
- Pour annuler un **git add** :
  - **git reset**

Notre projet

Gestion de code et déploiement

Pour déployer notre application, nous allons utiliser le service d'hébergement en ligne Heroku.
Pour cela, il nous faut d'abord avoir un dépôt sous GitHub.
Une fois le dépôt créé, nous pourrons le lier à Heroku.

Pour que l'application puisse être déployée :

- ajoutez le code suivant dans votre fichier app.py :

```
if __name__ == '__main__':
    app.run()
```

- installer gunicorn : gunicorn est une bibliothèque qui a pour rôle de permettre à votre application de dialoguer avec un serveur web.

  pip install gunicorn
- mettez à jour votre fichier **requirements.txt**.

Heroku

Présentation

Heroku est un service en ligne pour déployer, et ainsi mettre en ligne, une application de type web.

1. Créez une nouvelle application en cliquant sur **Create new app**.
2. Donnez un nom à votre application. Ce nom pourra être modifié dans les settings.
3. Choisissez de vous **connecter à GitHub** et connectez-vous au **repo-name** identifiée.
4. Allez tout en bas de la page et faîtes **Deploy branch** (restez sur la branche master).

Après un certain temps, le temps d'installer l'environnement puis d'exécuter le fichier **requirements.txt**, un bouton view vous permettra d'accéder à votre application en ligne.

Git et le travail en équipe

Les branches (1/3)

Git permet de travailler à plusieurs sur un même projet. Cependant, il faut faire attention à ne pas travailler en même temps sur un même fichier. En cas contraire, il y aura des problèmes de versions. Dans ce cadre, il est important de paramétrer son Git avec des branches.
Par défaut, Git met en place une branche principale : **main**.

Il est possible de créer d'autres branches : 1 par fonctionnalité par exemples (branches souvent appelée **features/nomDeLaFonctionnalité**).

En termes d'organisation nous allons définir les branches suivantes :

- **dev** : branche où sera déposée les fichiers une fois terminée. Ils y seront vérifiés avant d' être envoyés sur la branche main.
- **features** : branche qui aura comme sous-branches les fonctionnalités de l'application.
- sous features : les branches relatives aux fonctionnalités/codes réalisés (pages, templates, classes, …).

Les branches (2/3)

Dans le terminal :

1. Créer une branche : **git branch dev**
2. La branche devrait s'actualiser sur GitHub. En cas contraire : **git push --set-upstream origin dev**
3. Aller en local sur la branche : **git checkout dev**

Les habitudes à avoir :

1. regardez toujours sur quelle branche sous êtes avant de faire un commit et/ou un push :
   => **git status**
2. connaître l'état des actions en cours (add, commit, …) :
   => **git status**
3. connaître les branches :
   => **git branch -a**

Les branches (3/3)

Une fois les fichiers validés, pour les envoyer vers la branche principale (main) :

1. je me positionne sur la branche principale : **git checkout main**
2. je fusionne la branche où sont les fichiers avec cette branche principale : **git merge dev**
3. j’envoie les dossiers/fichiers vers la branche principale : **git push origin main**