Programmation Web coté serveur en Python

1. Préambule

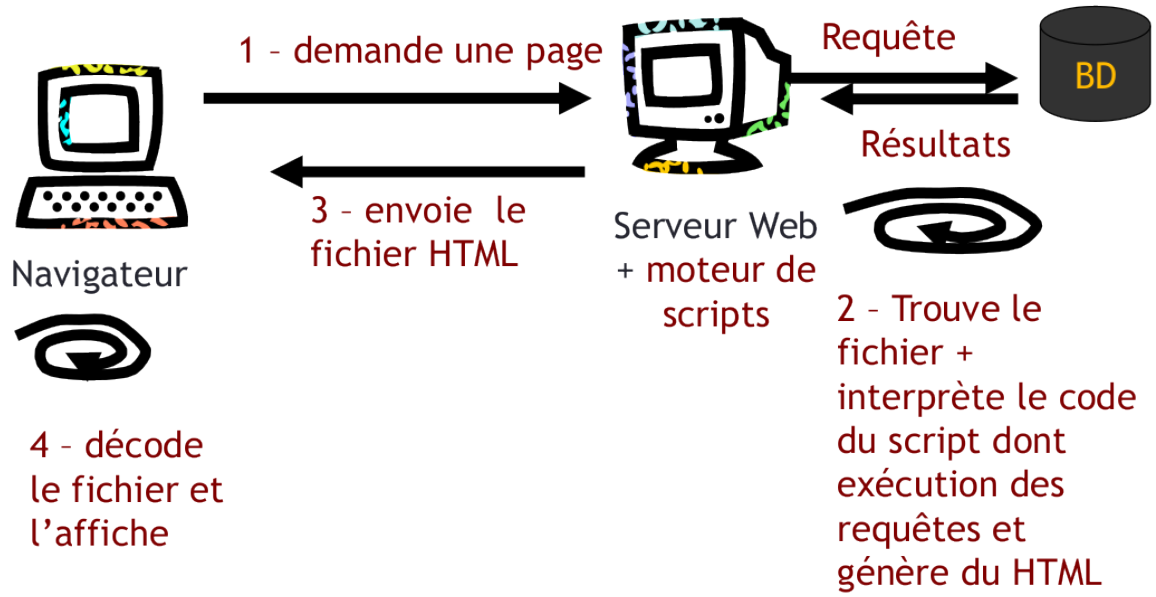
1.1 Langages coté client et langages coté serveur ...

HTML *(HyperText Markup Language)* et CSS *(Cascading Style Sheet)*, sont deux langages de simple affichage statique de contenus (statique car rien ne change, le contenu est toujours le même).

Les langages de script permettent de produire des pages web non statiques, c'est-à-dire dont le contenu peut varier en fonction de différentes conditions (et notamment des actions de l'utilisateur).

Il existe des langages côté client (le principal actuellement est Javascript) et des langages côté serveur.

PHP est un célèbre exemple de langage de programmation qui s'exécute côté serveur. Ce langage permet de produire du code, visualisable par le client, qui peut différer en fonction de circonstances définies, ce qui introduit donc un certain dynamisme. Ce dynamisme peut être accru grâce à une connexion avec une base de données (PostGreSQL, MySQL, SQLite, etc.).



La figure ci-dessus (également présentée dans les slides du 1er cours) présente l'architecture couplant un langage de programmation et une base de données afin de générer, sur le serveur, du contenu dynamique lors d'une requête en provenance d'un client (navigateur Web) : c'est ce que nous allons implémenter dans les 2 prochains TP.
Nous n'utiliserons pas le langage PHP mais le langage Python avec le *micro-framework* `Flask`.

Il existe de nombreux *frameworks* de programmation de **serveur** Web dans un multitude de langages, par exemple :

- en PHP : `laravel`, `Symphony`,
- en Python : `Django`, `Flask`, `cherry.py`
- en JavaScript avec node.js : `Express.js`,
- en Java : `Spring`, ...
- en Ruby : `Ruby on Rails`, ...

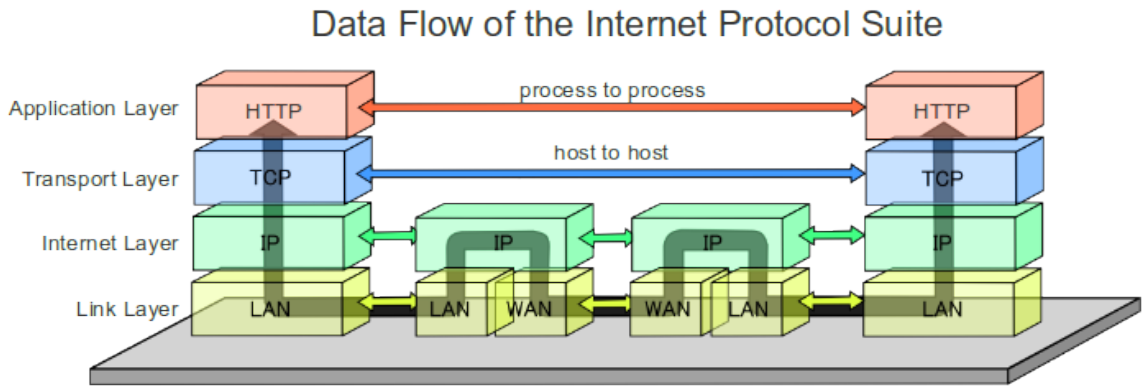
Les logiciels `Apache HTTP server` et `NGINX` sont également des serveurs HTTP.

Au même titre, il existe différents **clients**, que vous utilisez ou connaissez déjà pour certains puisqu'il s'agit notamment des navigateurs Web :

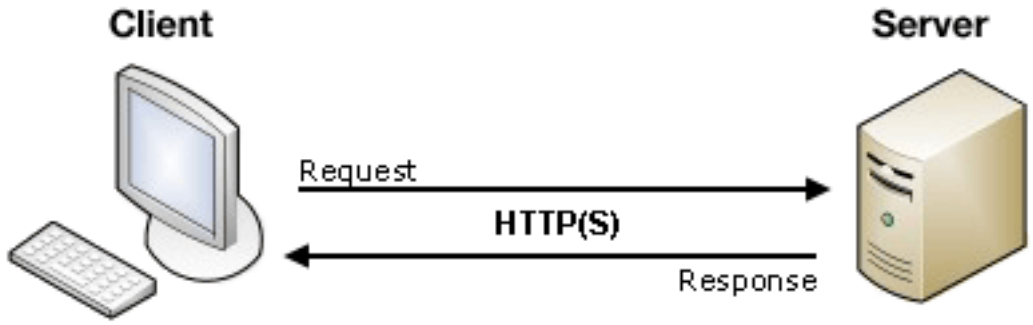
- `Chrome`, `Firefox`, `Microsoft Edge`, etc.
- En ligne de commande : `cURL`, `Wget`
- Des bibliothèques proposent d'effectuer des requêtes HTTP dans la plupart des langages de programmation (`requests` et `httpx` en Python par exemple)

1.2 Le protocole HTTP

Dans le 1er cours, nous avons évoqué le protocole HTTP (pour *Hypertext Transfer Protocol*, littéralement "protocole de transfert hypertexte"). Il s'agit d'un **protocole de communication client-serveur** (se situant au niveau "application" du modèle en couches présenté à cette occasion et dont l'illustration est rappelée ci-dessous).



Source: Wikimedia Commons - user: renepick / CC-BY-SA-3.0



Source: Mozilla Developpers Network

Le protocole HTTP définie plusieurs **méthodes**. Une méthode est une commande spécifiant au serveur un **type de requête**, lui demandant d'effectuer une action particulière sur la ressource désignée par l'URL qui accompagne cette méthode.

Les méthodes qui seront utilisées dans cette série de TP sont les suivantes (il s'agit également des plus utilisées):

- **GET** : méthode pour demander une ressource (une requête GET est sans effet sur la ressource, on peut la répéter et obtenir le même résultat), c'est la méthode utilisée lors de la récupération d'une page Web par exemple.
- **POST** : méthode pour transmettre des données en vue de leur traitement (par exemple depuis un formulaire HTML)

→ **Concrêtement, lorsqu'une page HTML est consultée, la première étape consiste à la récupérer : une requête HTTP de type GET sur l'url de la page est effectuée**.

→ **De même, lorsque vous validez un formulaire, une requête HTTP de type POST est effectuée sur une url du serveur permettant de valider et de traiter le formulaire**.

Ces autres méthodes sont parfois utilisées (notamment lors de l'utilisation d'une API de type REST pour les 3 dernières), gardez leur nom en tête pour savoir qu'il s'agit de méthodes HTTP si vous les rencontrez :

- HEAD : méthode pour demander des informations sur la ressource, sans récupérer la ressource.
- PUT : méthode pour remplacer ou ajouter une ressource.
- DELETE : méthode pour supprimer une ressource.
- PATCH : méthode pour faire une modification partielle (contrairement à PUT qui la remplacerait) d'une ressource.

1.3 Le fil rouge de ce TP et des suivants ...

Nous souhaitons construire **une plate-forme collaborative permettant de noter et de donner un avis sur les bâtiments du campus**. Cette tâche complexe peut-être décomposée en plusieurs sous-tâches:

- créer une base de données permettant de stocker cette information
- donner à l'utilisateur la possibilité de consulter l'information (localisation des bâtiments notés et détails des notes/avis)
- donner à l'utilisateur la possibilité de saisir une nouvelle information (ajout d'une note/avis à une localisation en ayant déjà où non)

Aujourd'hui (**TP5**), nous verrons comment organiser le code coté serveur pour que différentes actions puissent être effectuées en fonction du type de requête ou du chemin de la requête *(i.e. comment passer d'un contenu statique à un contenu dynamique)* et nous verrons comment utiliser des *templates* HTML.

Le **TP6** servira à créer un formulaire dans la page HTML et à le traiter coté serveur pour ajouter son contenu dans une base de données SQLite. Les informations présentes dans la base de données seront mobilisées pour être affichées coté client ; elles pourront également être mises à jour.

Le **TP7** servira à ajouter une carte interactive dans la page HTML. C'est cette carte qui sera utilisée comme interface pour l'ajout d'avis (lors du clic de l'utilisateur par exemple) ainsi que pour l'affichage de la localisation des avis présents dans la base de données (sous forme de *markers*). Cette carte fera appel à du code Javascript, exécuté directement dans le navigateur web du client. L'application construite pourra être déployée individuellement sur la plate-forme PythonAnywhere.

Le **TP8**, sera consacré au langage Javascript (aspects théoriques et pratiques).

2. La bibliothèque `Flask`

Il s'agit d'un *micro-framework* conçu pour développer une application Web coté serveur. À ce titre il va fournir des outils génériques pour créer ce type d'application tout en favorisant la mise en oeuvre de bonnes pratiques et en limitant l'écriture de code inutile. Le qualificatif *micro* décrit ici plusieurs réalités :

- vise à garder le code de base simple mais extensible,
- n'est pas opiniâtre dans le choix d'une base de données,
- au final, `Flask` s'occupe de l'essentiel (gérer les routes et les requêtes HTTP et permettre l'utilisation de *templates*) mais les autres tâches "optionnelles" (connexion à une DB, authentification, formulaires, etc.) sont laissées à des extensions.

`Flask` repose notamment sur deux bibliothèques dont il est utile d'avoir connaissance : `Werkzeug` (une boite à outils WSGI) et `Jinja2` (un moteur de *template*).

Liens utiles:

- Documentation officielle de `Flask`: https://flask.palletsprojects.com/en/1.1.x/
- Documentation officielle de `Jinja2`: https://jinja.palletsprojects.com/en/2.11.x/
- Documentation officielle de `Werkzeug` : https://werkzeug.palletsprojects.com/en/0.16.x/
- Liste de *plugins* et de ressources pour `Flask`: https://github.com/humiaozuzu/awesome-flask

2.1 Premier pas avec Flask - Application et routes

Application Code :

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def root():
    return "Hello from Flask!"
```

Pour tester : http://mthh.pythonanywhere.com/

Explications :

- app désigne une application Flask;
- la fonction `root` est appelée une **vue**. Elle retourne une chaîne de caractères, qui sera le contenu de la réponse. Par défaut, le statut de la réponse est 200, et le type de contenu est HTML, encodé en UTF-8;

- la ligne qui précède la fonction `root` est un **décorateur**, il sert ici à spécifier l'URL pour laquelle cette vue doit être utilisée (sa ***route***). Ce décorateur est une méthode de l'objet `app`, l'application Flask, créée au-dessus, dans laquelle s'inscrit cette route.

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

Dans Flask, les routes sont déclarées via le décorateur `@app.route`, comme dans l'exemple ci-dessus. Une route peut être paramétrée, auquel cas le paramêtre sera passé à la fonction vue :

```
@app.route("/hello/<name>")
def hello(name):
    return "Hello {}".format(name)
```

Pour tester : http://mthh.pythonanywhere.com/hello/John

Les exemples ci-dessus vous permettent de personnaliser la réponse en fonction de la demande de l'utilisateur ou de faire effectuer des calculs coté serveur en bénéficiant de la puissance et de l'étendue des fonctionnalités offertes par python et son écosystème.

- Il est également possible de spécifier plusieurs paramètres lors de la définition d'une route en utilisant la forme `/hello/<a>/<b>` :

```
@app.route("/add/<a>/<b>")
def add(a, b):
    return "Result: {}".format(int(a) + int(b))
```

Pour tester : http://mthh.pythonanywhere.com/add/3/12

- Il est possible de spécifier le type attendu pour les paramètres de la route ; s'ils ne sont pas satisfaits, une réponse de type `404 - NOT FOUND` sera renvoyé par défaut :

```
@app.route("/add/<int:a>/<int:b>")
def add(a, b):
    return "Result: {}".format(a + b)
```

Pour tester : http://mthh.pythonanywhere.com/add/12/abc

Un réponse `404 - NOT FOUND` sera également renvoyée si vous essayer d'accéder à une route qui n'est pas définie (ici `abdef`) :

Pour tester : http://mthh.pythonanywhere.com/abdef

- Par commoditée il est également possible de spécifier plusieurs routes conduisant à l'exécution d'une fonction :

```
@app.route("/greetings")
@app.route("/greetings/<name>")
def greetings(name=None):
    return "Hello {} !".format(name or 'world')
```

Pour tester : http://mthh.pythonanywhere.com/greetings Pour tester : http://mthh.pythonanywhere.com/greetings/Paul

Méthodes HTTP Lors de la création d'une route, il est possible de spécifier la ou les méthodes par laquelle elle est accessible. Dans les exemples précédents, comme aucune méthode n'était spécifiée explicitement, il s'agissait de la méthode **GET**. Ainsi notre première fonction aurait pu être écrite de la manière qui suit :

```
@app.route("/", methods=['GET'])
def root():
    return "Hello from Flask!"
```

Une route qui accepte seulement une méthode **POST** sera donc déclarée de la manière suivante :

```
@app.route('/valid-form', methods=['POST'])
```

Il pourra parfois être nécessaire de créer des routes qui acceptent les deux méthodes. Prenons l'exemple du code ci-dessous. La route `/update` est créée pour accepter les méthodes **GET** et **POST** :

```
from flask import request

@app.route('/update', methods=['GET', 'POST'])
def example():
    if request.method == 'GET':
        return render_template('update.html')
    elif request.method == 'POST':
        # traitement des données envoyées ...
        return 'Entry updated' !
```

- S'il s'agit d'une requête de type **GET**, la page qui correspond au *template* `update.html` sera retournée au client *(on s'imagine qu'elle contient notamment un formulaire permettant de mettre à jour une entrée dans la BD)*.
- S'il s'agit d'une requête de type **POST**, les données envoyées sont traitées pour mettre à jour l'entrée *(cette action correspond à la validation et à l'envoi, par le client, du formulaire contenu dans la page auquel il a accedé précédement par la route `/update`)* :

2.2 *Templates*

`Flask` dispose d'un mécanisme qui permet d'utiliser un **modèle** *(template)* de document dont certaines parties seront remplacées dynamiquement à l'exécution, avant d'être envoyé au client.

Les exemples présentés jusqu'ici ne renvoient que des chaînes de caractères. Comme indiqué plus haut, il s'agit en réalité de contenu HTML ; vous pouvez ainsi écrire des routes renvoyant du contenu HTML correctement formatté et généré de manière dynamique:

```
from datetime import date

@app.route("/pretty-add/<int:a>/<int:b>")
def pretty_add(a, b):
    current_date = date.today().isoformat()
    result = a + b
    return """
    <!DOCTYPE html>
    <html>
      <head>
        <meta charset="utf-8" />
        <title>Addition result page</title>
      </head>
      <body style="text-align:center;">
        <h1>Result page</h1>
        <p>{} + {} = <strong>{}</strong></p>
        <hr/>
        <p><em>Computed on {}</em></p>
      </body>
    </html>""".format(a, b, result, current_date)
```

Pour tester : http://mthh.pythonanywhere.com/pretty-add/12/20

Si cette méthode est utilisable, elle va toutefois vite rendre votre code difficilement compréhensible et elle ne facilite pas la réutilisation de morceau de code qui pourraient l'être.

C'est ici que l'intérêt du **moteur de *templates*** rentre en jeu. Ce dernier va permettre de stocker dans des fichiers séparés le **modèle** de nos pages HTML. Lors de l'exécution (avec la fonction `render_template` présentée ensuite), le modèle va être complété avec les paramètres fournis pour produire un document HTML utilisable par le client.

Syntaxe utilisée par `Jinja2` Reprenons l'exemple de notre route `pretty-add` définit ci-dessus ; il est possible de définir (dans un fichier `template/pretty-add.html` par exemple) le *template* suivant:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Addition result page</title>
  </head>
  <body style="text-align:center;">
    <h1>Result page</h1>
    <p>{{ a }} + {{ b }} = <strong>{{ result }}</strong></p>
    <hr/>
    <p><em>Computed on {{ current_date }}</em></p>
  </body>
</html>
```

… et de réécrire la fonction de la manière suivante :

```
from Flask import render_template

@app.route("/pretty-add/<int:a>/<int:b>")
def pretty_add(a, b):
    date_now = date.today().isoformat()
    result = a + b
    return render_template('pretty-add.html',
                           a=a,
                           b=b,
                           result=result,
                           current_date=date_now)
```

Dans cet exemple nous avons utilisé:

- des **expressions**, délimitées par les symboles `{{ ... }}` ; elles peuvent contenir des expressions Python et ici, lorsqu'on écrit `{{ result }}` on cherche donc à utiliser le contenu de la variable `result` pour l'afficher dans la page HTML qui sera renvoyée.

D'autres constructions peuvent être utilisées:

- des **déclarations** *(statements)*, délimités avec les symboles `{% ... %}` (conditions, boucles, etc.),
- des **commentaires**, délimités avec les symboles `{# ... #}`.

> **Remarque:** La fonction `render_template` accepte les variables attendues par le *template* (son premier argument) en tant qu'arguments optionnels nommés : on utilise le nom de la variable attendue dans le *template* comme clé et le nom de la variable dans le code actuel comme

> valeur (`current_date=date_now` dans l'exemple précédent).

Exemple avec une boucle Il va ainsi être possible d'utiliser, dans les *templates* des déclarations Python complexes ainsi que différents éléments utilisés habituellement pour le contrôle du flux d'exécution du programme. Regardez le code de ce *template* :

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>My Webpage</title>
  </head>
  <body>
    <ul id="navigation">
    {% for item in navigation %}
        <li><a href="{{ item['href'] }}">{{ item['caption'] }}</a></li>
    {% endfor %}
    </ul>

    <h1>My Webpage</h1>
    <p>Hello {{ name }}</p>

    {#
      A multi
          -line
        comment
    #}
  </body>
</html>
```

Explications :

- La variable `navigation` est ici une `list` de `dict`, de la forme

  ```
  navigation = [
      {"href": "index.html", "caption": "Accueil"},
      {"href": "partenaires.html", "caption": "Partenaires"}
  ]
  ```

  il est possible d'utiliser la déclaration **`{% for item in navigation %}`** qui a le même effet qu'une boucle utilisant `for` en Python - la syntaxe utilisée ici nécessite toutefois de fermer le bloc correspondant à cette boucle avec la déclaration **`{% endfor %}`**.

- La variable `name` est de type `str` et on l'affiche telle-quelle.

- La dernier bloc `{# ... #}` correspond à un commentaire ; comme dans les autres langages, servez-vous en si nécessaire !
- Les variables attendues (ici `navigation` et `name`) dans le *template* sont à donner en arguments optionnels et nommés de la fonction `render_template`.

Réutilisation de *templates* Reprenons l'exemple des pages que nous avons créées dans les TP précédents : elles contenaient une barre de navigation et un *footer* qui avaient tous deux vocation à être les mêmes pour chacune des pages du site.

Grâce au **mécanisme permettant d'inclure** des *templates*, nous allons par exemple pouvoir définir dans des fichiers séparés les éléments réutilisables de notre site, pour les inclure depuis le *template* contenant le contenu principal de la page :

```
{% include 'header.html' %}
    Contenu principal de la page ici...
{% include 'footer.html' %}
```

Les *templates* appelés de cette manière ont accès aux mêmes variables que le *template* qui les a inclus.

3 - Mise en pratique - Exercice 1 - Démarrage d'un projet `Flask`

3.1-option 1 : Installation "globale" et création d'un dossier pour le projet

Comme pour les autres packages Python, l'installation peut se faire en ligne de commande en utilisant l'utilitaire `pip`:

```
pip install Flask Jinja2
```

Si vous souhaitez les installer dans votre espace utilisateur vous pouvez utiliser l'argument `--user`:

```
pip install --user Flask Jinja2
```

On créé ensuite le dossier (`projet_flask`) dans lequel on va placer le code de notre projet :

```
mkdir projet_flask
```

3.1-option 2 : Installation dans un environnement virtuel au sein du dossier du projet

Cette solution sera préférée si vous avez l'habitude d'utiliser des environnements virtuels Python :

```
mkdir projet_flask && cd projet_flask
python3 -m venv env
```

- `projet_flask` représente le dossier dans lequel on va placer le code du projet
- `env` représente le nom de votre environnement virtuel, vous êtes libre d'en choisir un autre

On active ensuite l'environnement virtuel et on y installe Flask et Jinja2 :

```
source env/bin/activate
pip install Flask Jinja2
```

3.2 Structure du projet

Dans le dossier `projet_flask` vous aller créer un fichier `app.py` ainsi qu'un dossier `templates`. Dans le dossier `templates` vous décompresserez les données du jour : données TP5 - il s'agit de 2 fichiers de *templates* utilisant `Jinja2`.

Vous devez donc avoir la structure de projet suivante:

```
projet_flask
├── app.py                <--- Le code de l'appli Web à réaliser
├── templates          <--- Le dossier contenant nos modèles de documents HTML
│   ├── index.html
│   └── header.html
└── env              <--- Seulement si vous avez utilisé un environnement virtuel !
```

3.3 Premier bloc de code et vérification du démarage de l'application

Utilisez le premier extrait de code de ce TP (ici) pour créer votre application Flask dans le fichier `app.py`. Ajouter le morceau de code suivant à la fin du fichier :

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

Il vous permettra de lancer l'application en utilisant la commande :

```
python3 app.py
```

Il est ensuite possible d'accéder à l'application avec un navigateur web : http://127.0.0.1:5000/.

Comme vu lors du premier cours, `127.0.0.1` désigne votre **adresse IP** et `5000` désigne le **port TCP** sur lequel votre application écoute et émet.

4. Mise en pratique - Exercice 2 - Modification de l'application

1. Ajoutez le morceau de code suivant dans le fichier `app.py`, il s'agit des premiers avis fictifs à utiliser avant que nous ne constituions la base de données :

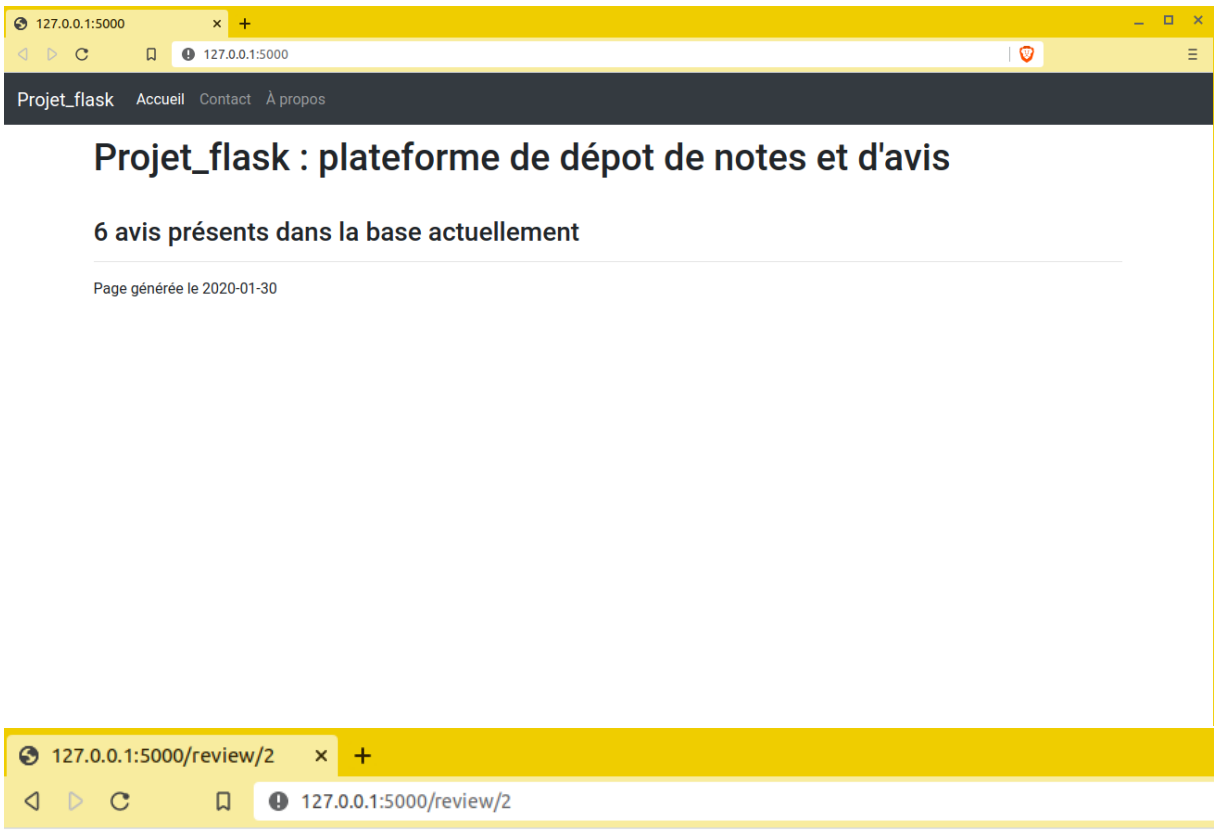
```
entries = [
 {"id_review": 1, "id_batiment": 26, "rate": 5, "comment": "Mattis molestie
 ↪  a iaculis at erat pellentesque adipiscing commodo."},
 {"id_review": 2, "id_batiment": 18, "rate": 4, "comment": "Amet massa vitae
 ↪  tortor condimentum lacinia."},
 {"id_review": 3, "id_batiment": 31, "rate": 3, "comment": "Imperdiet sed
 ↪  euismod nisi porta lorem mollis aliquam."},
 {"id_review": 4, "id_batiment": 22, "rate": 1, "comment": "Dignissim enim
 ↪  sit amet venenatis. Urna cursus eget nunc scelerisque."},
 {"id_review": 5, "id_batiment": 26, "rate": 4, "comment": "A pellentesque
 ↪  sit amet porttitor eget dolor morbi."},
 {"id_review": 6, "id_batiment": 18, "rate": 3, "comment": "Consequat nisl
 ↪  vel pretium lectus quam id leo in vitae."},
]
```

2. Vous devez modifier la route existante afin que la route `/` et la route `/index` appellent toutes deux la fonction `root`. Cette fonction devra utiliser le *template* `index.html`. Vous devrez également ouvrir ce fichier de *template* pour voir les variables qui sont attendues et que vous devrez donner en argument à la fonction `render_template`.

3. Vous devez créer une nouvelle route du nom de `/review` doit accepter un argument de type `int`, l'identifiant de l'avis que l'utilisateur souhaite obtenir. Vous devez également écrire la fonction correspondante : pour un identifiant donné, elle doit renvoyer la note et l'avis sous forme d'un bloc de texte formatté de manière simple *(par exemple: "Note X - Commentaire : xxxxx")*. Si aucun avis ne correspond à l'identifiant, elle retourne une chaine de caractères vide.

4. Dans votre navigateur, vérifiez les points suivants avant de passer à l'exercice qui suit:

- les routes `/` et `/index` fonctionnent et renvoient le même contenu (cf. image).
- la route `/review` retourne une erreur `404 - NOT FOUND` alors que la route `/review/2` retourne bien le contenu de l'avis correspondant.



Rate: 4 - Comment: Amet massa vitae tortor condimentum lacinia.

5. Mise en pratique - Exercice 3 - Utilisation des *templates*

1. En plus d'afficher le nombre d'avis présents dans la base, nous souhaitons afficher la totalité de la liste des avis sur la page d'accueil. Vous devez modifier le *template* `index.html` pour créer dynamiquement un élément liste HTML où chaque item correspondra à un avis. Un exemple similaire a été donné ici.