TD 1 *API REST Flask*

Dans ce TD, nous allons débuter la conception d'une architecture logicielle simple avec un serveur fondé sur une API *REST* en *Flask*. Dans un deuxième temps, nous compléterons ce travail avec un front en JavaScript standard (Vanilla JS). Parallèlement, nous commencerons l'étude de la bibliothèque *VueJS* qui nous permettra de réaliser un front plus élaboré basé sur des composants.
REST (*Representational State Transform*) est une façon moderne de concevoir les services web et possède les avantages suivants :

- Bonne montée en charge du serveur
- Simplicité des serveurs (retour aux sources du protocole HTTP)
- Équilibrage de charge
- le serveur offre une API
- les services sont représentés par des URL donc simplicité et bonne gestion du cache
- Possibilité de décomposer des services complexes en de multiples services plus simples qui communiquent entre eux

Les technologies concurrentes à REST sont XML-RPC et SOAP (Microsoft)
Les principes de REST ont été théorisés par Roy Fielding dans sa thèse : `http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm` :

1. Séparation claire entre Client et Serveur
2. Le client contient la logique métier, le serveur est sans état
3. Les réponses du serveur peuvent ou non être mises en cache
4. L'interface doit être simple, bien définie, standardisée
5. Le système peut avoir plusieurs couches comme des proxys, systèmes de cache, etc
6. Éventuellement, les clients peuvent télécharger du code du serveur qui s'exécutera dans le contexte du client

Pour mémoire, une API REST peut offrir les méthodes suivantes :

Méthode	Rôle	Code retour HTTP
GET	URL pour récupérer un élement	200
GET	URL pour récupérer une collection d'élements	200
POST	URL pour poster une collection d'élements	201
DELETE	URL pour effacer un élement	200
DELETE	URL pour effacer une collection d'élements	200
PUT	URL pour modifier un élement	200

FIGURE 1 – Méthodes HTTP et REST

Mais on peut aussi avoir des erreurs :

- 400 Bad Request : requête mal formée
- 404 Not Found : la ressource demandée n'existe pas
- 401 Unauthorized : Authentification nécessaire pour accéder à la ressource.
- 405 Method Not Allowed : Cette méthode est interdite pour cette ressource

- 409 Conflict : Par exemple un PUT qui crée une ressource 2 fois
- 500 Internal Server Error : Toutes les autres erreurs du serveur

Par ailleurs, le serveur REST ne maintient pas d'état, les requêtes sont indépendantes les unes des autres. C'est un retour aux fondamentaux du protocole HTTP qui n'est pas doté de beaucoup de capacités de mémorisation

App de Quiz

Le but final est de réaliser une application qui permet de gérer des quiz dont le service d'API sera réalisé à l'aide de Flask et le client sera réalisé à l'aide de VueJS.
On va réaliser cette application en plusieurs étapes :

1. Réaliser une petite API REST de gestion de tâhes à faire (Todos) à la main avec Flask.
2. Puis réaliser une API "à la main" qui permet de gérer des quiz en mémoire.
3. Puis on ajoutera un modèle de Quiz et une base de données SQLite.
4. On va tester cette application avec Postman, puis réaliser un petit client JS Standard pour interagir avec l'API.
5. Ensuite, nous pourrons utiliser la bibliothèque Flask-RESTful qui permet de réaliser des API REST avec Flask pour amélorer le code de l'API
6. Enfin, nous améliorerons le client en utilisant des composants VueJS

Exercice 1 *API plus simple à la main avec Flask*
Commençons par un modèle simple de todos en mémoire. Le modèle est le suivant, à mettre dans un fichier `models.py` dans le répertoire `todo`

```
tasks = [
    {
        'id': 1,
        'title': 'Courses',
        'description': 'Salade , Oignons , Pommes , Clementines',
        'done': True
    },
    {
        'id': 2,
        'title': 'Apprendre REST',
        'description': 'Apprendre mon cours et comprendre les exemples',
        'done': False
    },
    {
        'id': 3,
        'title': 'Apprendre Ajax',
        'description ': 'Revoir les exemples et ecrire un client REST JS avec
           Ajax',
        'done': False
    }
]
```

Nous construisons une App Flask sur le modèle vu précédemment :

```
.
    .flaskenv
    .gitignore
    todo
        __init__.py
```

```
        app.py
        models.py
        views.py
```

Avec le fichier `.flaskenv` suivant :

```
FLASK_APP=todo
FLASK_DEBUG=1
```

et le fichier `.gitignore` suivant :

```
__pycache__
.vscode
.env
venv
```

Le fichier `__init__.py` contient juste les imports :

```
from .app import app
import todo.views
import todo.models
```

Le fichier `app.py` contient la création de l'application Flask :

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

Le fichier `views.py` contient les routes de l'API, à commencer par la route de base qui renvoie un message une tâche spécifiée par son id :

```
from flask import jsonify, abort, make_response, request
from .app import app
from .models import tasks

def make_public_task(task):
    new_task = {}
    for field in task:
        if field == 'id':
            new_task ['uri'] = url_for ('get_task', task_id = task['id'],
                _external = True)
        else:
            new_task [ field] = task[ field ]

    return new_task

@app.route ('/todo/api/v1.0/tasks', methods = ['GET'])
def get_tasks ():
    return jsonify ( tasks =[ make_public_task (t) for t in tasks ])
```

1.1 Tester votre serveur, vérifier que vous arrivez à récupérer les tâches

1.2 A quoi sert ici l'uri de la tâche ?

Exercice 2 *Routes d'envoi de données vers l'API*

Nous allons maintenant ajouter une route qui permette d'envoyer des données vers l'API.

```
@app.route ('/todo/api/v1.0/tasks', methods = ['POST'])
def create_task ():
    if not request.json or not 'title' in request.json:
        abort (400)

    task = {
        'id': tasks [ -1][ 'id '] + 1,
        'title': request.json['title'],
        'description': request .json.get('description', ""),
        'done': False
    }

    tasks.append(task)
    return jsonify ( { 'task': make_public_task (task) } ), 201
```

2.1 Modifier votre fichier `view.py` en ajoutant cette nouvelle route

Exercice 3 *Routes de gestion des erreurs*

En cas d'erreur, nous pouvons renvoyer un code d'erreur HTTP, par exemple 404 pour une tâche non trouvée, ou 400 pour une requête mal formée. Par exemple, voici une route qui renvoie une erreur 404 si la tâche n'est pas trouvée :

```
@app.errorhandler(404)
def not_found ( error ):
    return make_response ( jsonify ( { 'error': 'Not found' } ), 404)
```

Et voici une route qui renvoie une erreur 400 si la requête est mal formée :

```
@app.errorhandler (400)
def not_found ( error ):
    return make_response ( jsonify ( { 'error': 'Bad request' } ), 400)
```

3.1 Ajouter ces deux nouvelles routes

Exercice 4 *Tests avec curl et avec Postman*

Curl est interface en ligne de commande pour interroger une ressource distante. Il va nous permettre de jouter le rôle d'un client de notre serveur REST. Nous pouvons tester cette API avec curl, par exemple pour récupérer la liste des tâches :

```
curl -i http://localhost:5000/todo/api/v1.0/tasks
```

et pour envoyer une nouvelle tâche au serveur :

```
curl -i -H "Content-Type: application/json" -X POST -d '{"title":"
Read a book"} ' http://localhost:5000/todo/api/v1.0/tasks
```

4.1 Tester votre serveur avec `curl`, vérifier que vous arrivez à consulter et à ajouter une nouvelle tâche.

Postman est un programme qui permet d'interroger un serveur, de manière graphique. `https://www.postman.com/`

4.2 Installez pour cela Postman dans le HOME local de votre poste.

4.3 Tester les requêtes GET et POST. Enregistez ces requêtes dans POSTMAN, pour pouvoir les exécuter facilement. Vous rajouterez de nouvelles requêtes par la suite

Exercice 5 *Routes de modification et de suppression de données*

Voici par exemple une route qui permet de modifier une tâche :

```
@app.route ('/todo/api/v1.0/tasks/<int:task_id>', methods = ['PUT'])
def update_task ( task_id ):
    task = [task for task in tasks if task['id'] == task_id ]

    if len(task) == 0:
        abort (404)
    if not request.json:
        abort (400)
    if 'title' in request.json and type( request.json['title']) != str:
        abort (400)
    if 'description' in request.json and type( request.json['description'
        ]) is not str:
        abort (400)
    if 'done' in request .json and type( request.json['done']) is not bool:
        abort (400)
    task[0]['title'] = request.json.get('title', task[0]['title'])
    task[0]['description'] = request .json.get('description', task [0]['
       description'])
    task[0]['done'] = request.json.get('done', task[0]['done'])
    return jsonify ( { 'task': make_public_task (task[0]) } )
```

Pourquoi faut-il tester les données reçues et leurs types ? Réaliser vous-même une route de suppression de tâches. Testez ces routes avec curl et Postman.

Exercice 6 *Changement de modèle de données : utilisation de Questionnaires et de Questions*

On va maintenant changer le modèle de données pour utiliser un modèle plus riche avec des questionnaires et des questions. Par exemple

```
class Questionnaire (db.Model ):

    id = db.Column(db.Integer, primary_key = True)
    name = db.Column (db.String (100) )


    def __init__ (self, name):
        self.name = name

    def __repr__ (self):
        return "<Questionnaire (%d) %s>" % (self.id , self.name)

    def to_json (self):
        json ={
            'id':self.id ,
            'name':self.name
        }

        return json

class Question (db.Model ):
    id = db.Column (db.Integer , primary_key = True)
    title = db.Column (db.String(120) )
    questionType = db.Column (db.String(120) )
    questionnaire_id = db.Column (db.Integer , db.ForeignKey ('questionnaire .
id'))

    questionnaire = db. relationship ("Questionnaire", backref = db. backref (
"questions", lazy="dynamic"))
```

Exercice 7 *Routes de gestion des questions et des questionnaires*
Mettez en place une API complète pour gérer les questionnaires et les questions. Testez cette API avec curl et Postman