

Dans ce TD, nous allons débiter la conception d'une architecture logicielle simple avec un serveur fondé sur une API *REST* en *Flask*.

REST (*Representational State Transfer*) 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 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 élément | 200 |
| GET | URL pour récupérer une collection d'éléments | 200 |
| POST | URL pour poster une collection d'éléments | 201 |
| DELETE | URL pour effacer un élément | 200 |
| DELETE | URL pour effacer une collection d'éléments | 200 |
| PUT | URL pour modifier un élément | 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

Exercice 1 *API de todos avec Flask*

Nous débutons la construction d'une api de tâches à faire, des *todos*. Chaque tâche a un titre, une description et un état (fait/pas fait)

Nous construisons une App Flask sur le modèle suivant :

```
.
  .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` :

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

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
    }
]
```

Le fichier `__init__.py` rassemble les composants (app, views et models) :

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

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

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

Le fichier `views.py` contient les routes de l'API. C'est sur ce fichier que nous allons principalement travailler.

Exercice 2 *Récupérer les tâches*

Nous commençons par la route de base qui renvoie la liste des tâches :

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

@app.route('/todo/api/v1.0/tasks', methods = ['GET'])
def get_tasks():
    return tasks, 200
```

2.1 Tester votre serveur, vérifier que vous arrivez à récupérer toutes les tâches avec votre navigateur. Noter que le retour est un couple composé des tâches et du code retour http. Ici le code 200 signifie que la requête s'est déroulée avec succès. Le dictionnaire des tâches en Python est converti en JSON.

2.2 Nous modifions légèrement le retour avec la fonction `jsonify`

```
@app.route('/todo/api/v1.0/tasks', methods = ['GET'])
def get_tasks():
    public_tasks = []
    for task in tasks:
        public_tasks.append(task)
    return jsonify( { 'tasks': public_tasks } )
```

La fonction `jsonify` permet de convertir des objets python en JSON, d'ajouter les bons éléments dans l'entête, d'ajouter le code 200 automatiquement. Noter également que nous ne retournons plus une liste, mais un objet JSON. Cela améliore l'évolutivité, la sécurité et est conforme aux standards. Modifiez votre fonction pour intégrer cette transformation.

2.3 On souhaite maintenant ajouter une route pour récupérer une seule tâche à partir de son id. Ajouter cette nouvelle fonction

```
@app.route('/todo/api/v1.0/tasks/<int:task_id>', methods = ['GET'])
def get_task(task_id):
    for task in tasks:
        if task['id'] == task_id:
            return jsonify({'task': task})
    return {'error': 'Not found'}, 404
```

Noter que dans le chemin `task_id`, présent dans le chemin est l'argument de votre requête. Il faut gérer le cas où l'id de la tâche n'existe pas et donc générer un message (sous forme d'un dictionnaire), accompagné du code erreur http 404, indiquant que la ressource n'a pas été trouvée.

Exercice 3 *Gestion des uri des tâches*

3.1 Pour respecter la philosophie REST, il est préférable que chaque tâche soit accompagnée d'une uri, qui permettra notamment de naviguer entre les différentes tâches. Il faut donc modifier notre première fonction pour remplacer l'id, par l'uri de cette tâche. Ajouter cette fonction dans votre fichier

```
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
```

Noter l'utilisation de la fonction `url_for`. En effet, nous n'allons pas écrire les uri de chaque tâche directement dans le code. La fonction `url_for` prend en argument une fonction et retourne la route associée à celle-ci. L'argument `external` indique si l'uri retournée est absolue ou relative. (/todo/api/v1.0/tasks/1" ou http://127.0.0.1:5000/todo/api/v1.0/tasks/1)

```
@app.route('/todo/api/v1.0/tasks', methods = ['GET'])
def get_tasks():
    public_tasks = []
    for task in tasks:
        public_tasks.append(make_public_task(task))
    return jsonify( { 'tasks': public_tasks })
```

3.2 Modifier vos routes pour intégrer cette fonction

Exercice 4 Créer une nouvelle tâche

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

La seule différence avec la route pour obtenir la liste des tâches est la méthode http : `POST` à la place de `GET`.

Les données reçues sont contenues dans `request.json`, et on accède aux différents champs d'une tâche avec son nom. Par exemple `request.json['title']` pour obtenir le titre de la tâche à créer.

Il est nécessaire de vérifier que la requête est bien du JSON. Pour cela Flask examine le `Content-Type` et vérifie qu'il vaut bien `application/json`

Il faut également vérifier que certains champs sont bien présents, avant de créer notre tâche. Dans notre cas, le titre de la tâche est obligatoire. S'il est manquant, nous retournons un nouveau code d'erreur http, 400, indiquant que la requête est mal formée.

Pour la gestion de l'id, on incrémente le dernier id de la liste de tâches.

Noter que pour récupérer les champs `description` et `done`, `get` prend deux arguments, le deuxième étant l'argument par défaut.

```
@app.route('/todo/api/v1.0/tasks', methods = ['POST'])
def create_task():
    # vérification des données reçues
    if not request.json or not 'title' in request.json:
        return {'error': 'Bad request'}, 400
    # construction de la nouvelle tâche
    if tasks == []:
        new_id = 1
    else:
        new_id = tasks[-1]['id'] + 1
    task = {
        'id': new_id,
        'title': request.json['title'],
        'description': request.json.get('description', ""),
        'done': request.json.get('done', False),
    }
    # ajout de la nouvelle tâche aux tâches existantes
    tasks.append(task)
```

```
#retour de la nouvelle tâche avec son uri 201 indique qu'une ressource a été créée
return jsonify({'task': make_public_task(task)}), 201
```

4.1 Modifier votre fichier `views.py` en ajoutant cette nouvelle route

Exercice 5 *Tests avec curl et avec Postman*

Pour tester cette route et donc la création de nouvelles tâches, nous allons envoyer une requête avec l'outil `curl`.

Curl est une interface en ligne de commande pour interroger une ressource distante. Il va nous permettre de jouer 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
```

5.1 Consulter le man de `curl` pour découvrir à quoi servent les options `i`, `H`, `X`

5.2 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/>

5.3 Installez pour cela Postman dans le HOME local de votre poste (ou l'extension VSCode).

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

Exercice 6 *Routes de gestion des erreurs*

Nous avons vu les messages et les codes d'erreurs http. Ce type d'envoi d'erreur risque d'arriver fréquemment, et il est nécessaire de factoriser et de rationaliser cette gestion d'erreurs.

Flask propose une fonction `abort` à laquelle on spécifie un code d'erreur. On précise également les fonctions associées à ces codes d'erreur avec les décorateurs.

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 bad_request ( error ):
    return make_response ( jsonify ( { 'error': 'Bad request' } ), 400)
```

Modifier vos précédentes routes par des appels à `abort(400)` ou `abort(404)`

Exercice 7 *Modification d'une tâche*

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):
    # Recherche de la tâche à modifier avec son id
    task = None
    for taski in tasks:
        if taski['id'] == task_id:
            task = taski
            break
    # la tâche avec cette id n'existe pas
    if task is None:
        abort(404)
    # la requête n'est pas au format json
    if not request.json:
        abort(400)
    # Verification des types
    if 'title' in request.json and not isinstance(request.json['title'], str):
        abort(400)
    if 'description' in request.json and not isinstance(request.json['description'], str):
        abort(400)
    if 'done' in request.json and not isinstance(request.json['done'], bool):
        abort(400)
    # modification des champs de la tâche
    task['title'] = request.json.get('title', task['title'])
    task['description'] = request.json.get('description', task['description'])
    task['done'] = request.json.get('done', task['done'])
    # retour de la tâche modifiée
    return jsonify({'task': make_public_task(task)})
```

7.1 Ajouter cette route. Noter que le chemin est identique au chemin pour récupérer une tâche, seule la méthode http change.

7.2 Testez ces routes avec curl et/ou Postman. Penser à tester également les cas d'erreur

Exercice 8 *Suppression d'une tâche*

8.1 Ajouter une nouvelle route pour la suppression de tâche. Vous retournez {"status": "deleted"}