



API REST FLASK

Serveur de todos puis de quizzes

Exercice 1. Rappels REST

Dans ce TD, on va démarrer à réaliser 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.

On rappelle les définitions :

- RIA = Rich Internet Application
- REST = Representational State Transform
- Logique métier déportée vers le client
- Tâche principale du serveur : Offrir des services de récupération et de stockage de données

Les technologies concurrentes à REST sont XML-RPC et SOAP (Microsoft) REST est une façon moderne de concevoir ce genre de service et possède les avantages suivants :

- Bonne montée en charge du serveur
- Simplicité des serveurs (retour aux sources du protocole HTTP)
- Equilibrage de charge
- le serveur offre une API
- les services sont représentés par des URL's 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 Etat
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. Eventuellement, 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 :

Méthodes HTTP et REST

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

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 2. App de Quiz

Le but final est de réaliser une Application qui permet de gérer des quizzes 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 Todos à la main avec Flask.
2. Puis réaliser une API "à la main" qui permet de gérer des quizzes 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 améliorer le code de l'API.
6. Enfin, nous améliorerons le client en utilisant des composants VueJS.

Exercice 3. 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` :

```
1  tasks = [  
2  {  
3      'id': 1,  
4      'title': 'Courses',  
5      'description': 'Salade, Oignons, Pommes, Clementines',  
6      'done': True  
7  },  
8  {  
9      'id': 2,  
10     'title': 'Apprendre REST',  
11     'description': 'Apprendre mon cours et comprendre les exemples',  
12     'done': False  
13  },  
14  {  
15     'id': 3,  
16     'title': 'Apprendre Ajax',  
17     'description': 'Revoir les exemples et ecrire un client REST JS  
18                     avec Ajax',  
19     'done': False  
20  }  
21 ]
```

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 :

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

et le fichier `.gitignore` suivant :

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

Le fichier `__init__.py` contient juste un ou deux imports :

```

1 from .app import app
2 import todo.views
3 import todo.models

```

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

```

1 from flask import Flask
2 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 :

```

1 from flask import jsonify, abort, make_response, request
2 from .app import app
3 from .models import tasks
4 def make_public_task(task):
5     new_task = {}
6     for field in task:
7         if field == 'id':
8             new_task['uri'] = url_for('get_task', task_id = task['id'],
9                                     _external = True)
10        else:
11            new_task[field] = task[field]
12    return new_task
13 @app.route('/todo/api/v1.0/tasks', methods = ['GET'])
14 def get_tasks():
15     return jsonify(tasks=[ make_public_task(t) for t in tasks ])

```

A quoi sert ici l'uri de la tâche ?

Ajoutez la méthode `get_task(task_id)` associée à la route `'/todo/api/v1.0/tasks/<int :task_id>'`.

Exercice 4. Routes d'envoi de données vers l'API

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

```

1 @app.route('/todo/api/v1.0/tasks', methods = ['POST'])
2 def create_task():
3     if not request.json or not 'title' in request.json:
4         abort(400)
5     task = {
6         'id': tasks[-1]['id'] + 1,
7         'title': request.json['title'],
8         'description': request.json.get('description', ""),
9         'done': False
10    }
11    tasks.append(task)
12    return jsonify( { 'task': make_public_task(task) } ), 201

```

Exercice 5. 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 :

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

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

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

Exercice 6. Tests avec curl et avec Postman

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
```

Nous pouvons aussi tester cette API avec Postman, en utilisant les méthodes GET et POST. Installez pour cela Postman dans le HOME local de votre poste.

Exercice 7. Routes de modification et de suppression de données

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

```
1 @app.route('/todo/api/v1.0/tasks/<int:task_id>', methods = ['PUT'])
2 def update_task(task_id):
3     task = [task for task in tasks if task['id'] == task_id]
4     if len(task) == 0:
5         abort(404)
6     if not request.json:
7         abort(400)
8     if 'title' in request.json and type(request.json['title']) != str:
9         abort(400)
10    if 'description' in request.json and type(request.json['description']) is not str:
11        abort(400)
```

```

12     if 'done' in request.json and type(request.json['done']) is not bool
13         :
14         abort(400)
15     task[0]['title'] = request.json.get('title', task[0]['title'])
16     task[0]['description'] = request.json.get('description', task[0]['
17         description'])
18     task[0]['done'] = request.json.get('done', task[0]['done'])
19     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 8. 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 :

```

1     class Questionnaire(db.Model):
2         id = db.Column(db.Integer, primary_key = True)
3         name = db.Column(db.String(100))
4
5         def __init__(self, name):
6             self.name = name
7
8         def __repr__(self):
9             return "<Questionnaire (%d) %s>" % (self.id, self.name)
10
11        def to_json(self):
12            json={
13                'id': self.id,
14                'name':self.name,
15            }
16            return json
17
18        class Question(db.Model):
19            id = db.Column(db.Integer, primary_key = True)
20            title = db.Column(db.String(120))
21            questionType = db.Column(db.String(120))
22            questionnaire_id = db.Column(db.Integer, db.ForeignKey('questionnaire.
23                id'))
24            questionnaire = db.relationship("Questionnaire", backref = db.backref(
25                "questions", lazy="dynamic"))

```

Exercice 9. 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.