Python et technologies Web

1. Programmer des API avec Python et Flask

Salim Lardjane
Université de Bretagne Sud

Objectifs

- Les API (Application Programming Interfaces) Web sont des outils permettant de rendre de l'information et des fonctionnalités accessibles via internet.

- Dans la suite, on verra :

1. Ce que sont les API et quand les utiliser

2. Comment construire une API qui mette des données à disposition des utilisateurs

3. Quelques principes du bon design des API et on les appliquera sur un exemple.

Qu’est-ce qu’une API ?

- Une API web permet à de l’information et à des fonctionnalités d’être manipulées par des programmes informatiques via internet.

- Par exemple, avec l’API web Twitter, on peut écrire un programme dans un langage comme Python ou Javascript qui collecte des métadonnées sur les tweets.

Qu’est-ce qu’une API ?

- Plus généralement, en programmation, le terme API (abréviation de Application Programming Interface) réfère à une partie d’un programme informatique conçue pour être utilisée ou manipulée par un autre programme, contrairement aux interfaces qui sont conçues pour être utilisées ou manipulées par des humains.

- Les programmes informatiques ont souvent besoin de communiquer entre eux ou avec le système d’exploitation sous-jacent et les API sont une façon de le faire.

- Dans la suite, toutefois, on se limitera aux API Web.

Quand créer une API ?

- En général, on crée une API quand :

- Notre jeu de données est important, ce qui rend le téléchargement par FTP lourd ou consommateur de ressources importantes.

- Les utilisateurs ont besoin d’accéder aux données en temps réel, par exemple pour une visualisation sur un site web ou comme une partie d’une application.

- Nos données sont modifiées ou mises à jour fréquemment.

- Les utilisateurs n’ont besoin d’accéder qu’à une partie des données à la fois.

- Les utilisateurs ont à effectuer d’autres actions que simplement consulter les données, par exemple contribuer, mettre à jour ou supprimer.

Quand créer une API ?

- Si on dispose de données qu'on souhaite partager avec le monde entier, créer une API est une façon de le faire.

- Toutefois, les API ne sont pas toujours la meilleurs façon de partager des données avec des utilisateurs.

- Si le volume des données qu'on souhaite partager est relativement faible, il vaut mieux proposer un « data dump » sous la forme d'un fichier JSON, XML, CSV ou Sqlite. Selon les ressources dont on dispose, cette approche peut être viable jusqu'à un volume de quelques Gigaoctets.

Quand créer une API ?

- Notons qu’on peut fournir à la fois un data dump et une API; à charge des utilisateurs de choisir ce qui leur convient le mieux.

Terminologie des API

- Lorsqu'on construit ou qu'on utilise une API, on rencontre fréquemment les termes suivants :

- HTTP (HyperText Transfert Protocol) : c'est le principal moyen de communiquer de l'information sur Internet. HTTP implémente un certain nombre de « méthodes » qui disent dans quelle direction les données doivent se déplacer et ce qui doit en être fait. Les deux plus fréquentes sont GET, qui récupère les données à partir d'un serveur, et POST, qui envoie de nouvelles données vers un serveur.

Terminologie des API

- URL (Uniform Ressource Locator) : Adresse d'une ressource sur le web, comme http://www-facultesciences.univ-ubs.fr/fr. Une URL consiste en un protocole (http://), un domaine (www-facultesciences.univ-ubs.fr), un chemin optionnel (/fr). Elle décrit l'emplacement d'une ressource spécifique, comme une page Web. Dans le domaine des API, les termes URL, request, URI, endpoint désignent des idées similaires. Dans la suite, on utilisera uniquement les termes URL et request, pour être plus clair. Pour effectuer une requête GET ou suivre un lien, il suffit de disposer d'un navigateur Web.

Terminologie des API

- JSON (JavaScript Object Notation) : c’est un format texte de stockage des données conçu pour être lisible à la fois par les êtres humains et les machines. JSON est le format le plus usuel des données récupérées par API, le deuxième plus usuel étant XML.

- REST (REpresentational State Transfer) : c’est une méthodologie qui regroupe les meilleurs pratiques en termes de conception et d’implémentation d’APIs. Les API conçues selon les principes de la méthodologie REST sont appelées des API REST (REST APIs). Il y a toutefois de nombreuses polémiques autour de la signification exacte du terme. Dans la suite, on parlera plus simplement d’API Web ou d’API HTTP.

Implémenter une API

- Dans la suite, on va voir comment créer une API en utilisant Python et le cadre de travail Flask.

- Notre exemple d'API portera sur un catalogue de livres allant au delà de l'information bibliographique standard.

- Plus précisément, en plus du titre et de la date de publication, notre API fournira la première phrase de chaque livre.

Implémenter une API

- On commencera par utiliser Flask (http://flask.pocoo.org) pour créer une page web pour notre site.

- On verra comment Flask fonctionne et on s’assurera que notre logiciel est proprement configuré.

- Une fois que nous aurons une petite application Flask fonctionnelle sous la forme d’une page Web, nous transformerons le site en une API fonctionnelle.

Flask

- Flash est un cadre de travail (framework) Web pour Python. Ainsi, il fournit des fonctionnalités permettant de construire des applications Web, ce qui inclut la gestion des requêtes HTTP et des canevas de présentation.

- Nous allons créer une application Flask très simple, à partir de laquelle nous construirons notre API.

Pourquoi Flask ?

- Python dispose de plusieurs cadre de développement permettant de produire des pages Web et des API.

- Le plus connu est Django, qui est très riche.

- Django peut toutefois être écrasant pour les utilisateurs non expérimentés.

- Les applications Flask sont construites à partir de canevas très simples et sont donc plus adaptées au prototypage d’APIs.

Flask

- On commence par créer un nouveau répertoire sur notre ordinateur, qui servira de répertoire de projet et qu'on nommera projects.

- Les fichiers de notre projet seront stockés dans un sous-répertoire de projects, nommé api.

Flask

- On lance ensuite Spyder 3 et on saisit le code suivant :

```
import flask

app = flask.Flask(__name__)
app.config["DEBUG"] = True


@app.route('/', methods=['GET'])
def home():
    return "<h1>Distant Reading Archive</h1><p>This site
is a prototype API for distant reading of science
fiction novels.</p>"

app.run()
```

Flask

- On sauvegarde ensuite le programme sous le nom api.py dans le répertoire api précédemment créé.

Pour l'exécuter, on exécute le programme sous Spyder.

- On obtient dans le console l'affichage suivant (entre autres sorties) :

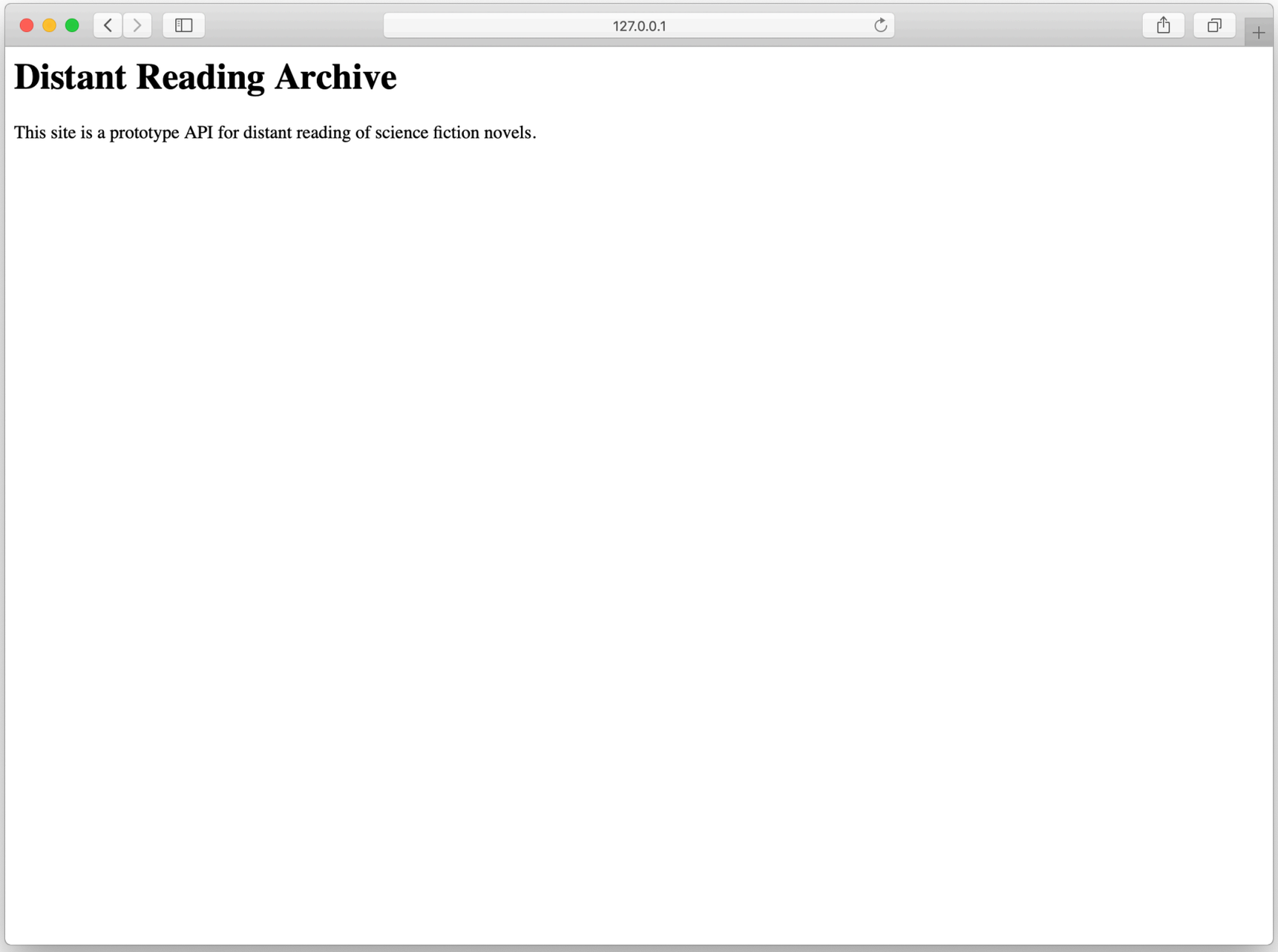
```
* Running on http://127.0.0.1:5000/ (Press
CTRL+C to quit)
```

Flask

- Il suffit de saisir le lien précédent dans un navigateur Web pour accéder à l'application.

- On a ainsi créé une application Web fonctionnelle.

Flask


127.0.0.1
Distant Reading Archive
This site is a prototype API for distant reading of science fiction novels.

Comment fonctionne Flask ?

- Flask envoie des requêtes HTTP à des fonctions Python.
- Dans notre cas, nous avons appliqué un chemin d’URL (‘/‘) sur une fonction : home.
- Flask exécute le code de la fonction et affiche le résultat dans le navigateur.
- Dans notre cas, le résultat est un code HTML de bienvenue sur le site hébergeant notre API.

Comment fonctionne Flask ?

- Le processus consistant à appliquer des URL sur des fonctions est appelé routage (routing).
- L'instruction :

```
@app.route('/', methods=[‘GET'])
```

apparaissant dans le programme indique à Flask que la fonction home correspond au chemin /.

Comment fonctionne Flask ?

- La liste methods (methods=['GET']) est un argument mot-clef qui indique à Flask le type de requêtes HTTP autorisées.

- On utilisera uniquement des requêtes GET dans la suite, mais de nombreuses application Web utilisent à la fois des requêtes GET (pour envoyer des données de l'application aux utilisateurs) et POST (pour recevoir les données des utilisateurs).

Comment fonctionne Flask ?

- import Flask : Cette instruction permet d'importer la bibliothèque Flask, qui est disponible par défaut sous Anaconda.

- app = flask.Flask(__name__) : Crée l'objet application Flask, qui contient les données de l'application et les méthodes correspondant aux actions susceptibles d'être effectuées sur l'objet. La dernière instruction app.run( ) est un exemple d'utilisation de méthode.

- app.config[« DEBUG » = True] : lance le débogueur, ce qui permet d'afficher un message autre que « Bad Gateway » s'il y a une erreur dans l'application.

- app.run( ) : permet d'exécuter l'application.

Création de l'API

- Afin de créer l'API, on va spécifier nos données sous la forme d'une liste de dictionnaires Python.

- A titre d'exemple, on va fournir des données sur trois romans de Science-Fiction. Chaque dictionnaire contiendra un numéro d'identification, le titre, l'auteur, la première phrase et l'année de publication d'un livre.

- On introduira également une nouvelle fonction : une route permettant aux visiteurs d'accéder à nos données.

Création de l’API

- Remplaçons le code précédent d’api.py par le code suivant :

```
import flask
from flask import request, jsonify

app = flask.Flask(__name__)
app.config["DEBUG"] = True

# Create some test data for our catalog in the form of a list of dictionaries.
books = [
    {'id': 0,
     'title': 'A Fire Upon the Deep',
     'author': 'Vernor Vinge',
     'first_sentence': 'The coldsleep itself was dreamless.',
     'year_published': '1992'},
    {'id': 1,
     'title': 'The Ones Who Walk Away From Omelas',
     'author': 'Ursula K. Le Guin',
     'first_sentence': 'With a clamor of bells that set the swallows soaring, the Festival of Summer came to the
city Omelas, bright-towered by the sea.',
     'published': '1973'},
    {'id': 2,
     'title': 'Dhalgren',
     'author': 'Samuel R. Delany',
     'first_sentence': 'to wound the autumnal city.',
     'published': '1975'}
]
```

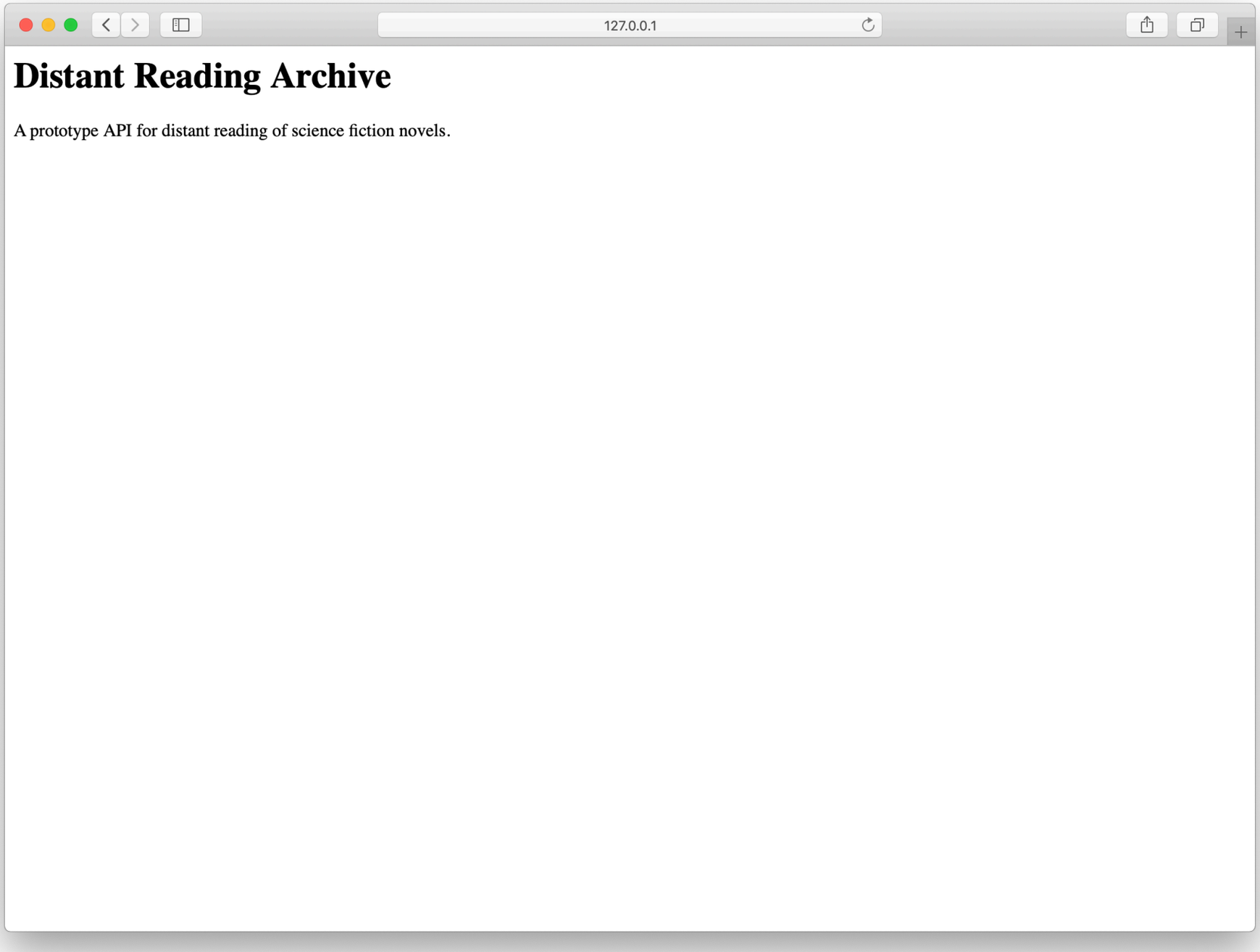
Création de l'API

```
@app.route('/', methods=['GET'])
def home():
    return '''<h1>Distant Reading Archive</h1>
<p>A prototype API for distant reading of science fiction novels.</p>'''


# A route to return all of the available entries in our catalog.
@app.route('/api/v1/resources/books/all', methods=['GET'])
def api_all():
    return jsonify(books)

app.run()
```

Création de l'API

- On exécute le code sous Spyder 3 et on met à jour la fenêtre du navigateur, ce qui donne :

Création de l’API

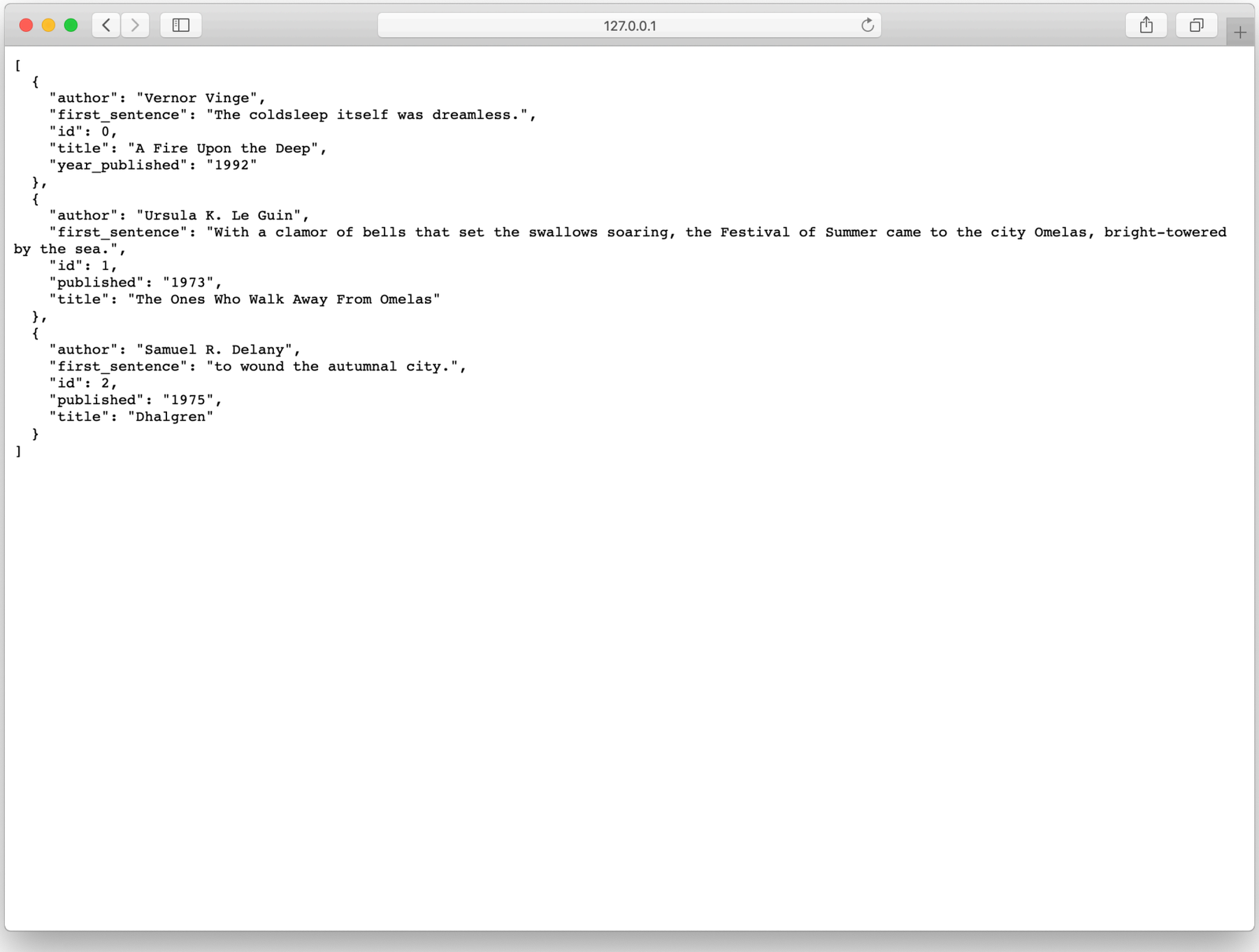


Création de l’API

- Pour accéder à l’ensemble des données, il suffit de saisir dans le navigateur l’adresse :

http://127.0.0.1:5000/api/v1/resources/books/all

Création de l'API



Création de l'API

- On a utilisé la fonction jsonify de Flask. Celle-ci permet de convertir les listes et les dictionnaires au format JSON.

- Via la route qu'on a créé, nos données sur les livres sont converties d'une liste de dictionnaires vers le format JSON, avant d'être fournies à l'utilisateur.

- A ce stade, nous avons créé une API fonctionnelle, bien que limitée.

- Dans la suite, nous verrons comment permettre aux utilisateurs d'effectuer des recherches spécifiques, par exemple à partir de l'identifiant d'un livre.

Accéder à des ressources spécifiques

- En l'état actuel de notre API, les utilisateurs ne peuvent accéder qu'à l'intégralité de nos données; il ne peuvent spécifier de filtre pour trouver des ressources spécifiques.

- Bien que cela ne pose pas problème sur nos données de test, peu nombreuses, cela devient problématique au fur et à mesure qu'on rajoute des données.

- Dans la suite, on va introduire une fonction permettant aux utilisateurs de filtrer les résultats renvoyés à l'aide de requêtes plus spécifiques.

Accéder à des ressources spécifiques

- Le nouveau code est le suivant :

```
import flask
from flask import request, jsonify

app = flask.Flask(__name__)
app.config["DEBUG"] = True

# Create some test data for our catalog in the form of a list of dictionaries.
books = [
    {'id': 0,
     'title': 'A Fire Upon the Deep',
     'author': 'Vernor Vinge',
     'first_sentence': 'The coldsleep itself was dreamless.',
     'year_published': '1992'},
    {'id': 1,
     'title': 'The Ones Who Walk Away From Omelas',
     'author': 'Ursula K. Le Guin',
     'first_sentence': 'With a clamor of bells that set the swallows soaring, the Festival of Summer
came to the city Omelas, bright-towered by the sea.',
     'published': '1973'},
    {'id': 2,
     'title': 'Dhalgren',
     'author': 'Samuel R. Delany',
     'first_sentence': 'to wound the autumnal city.',
     'published': '1975'}
]
```

Accéder à des ressources spécifiques

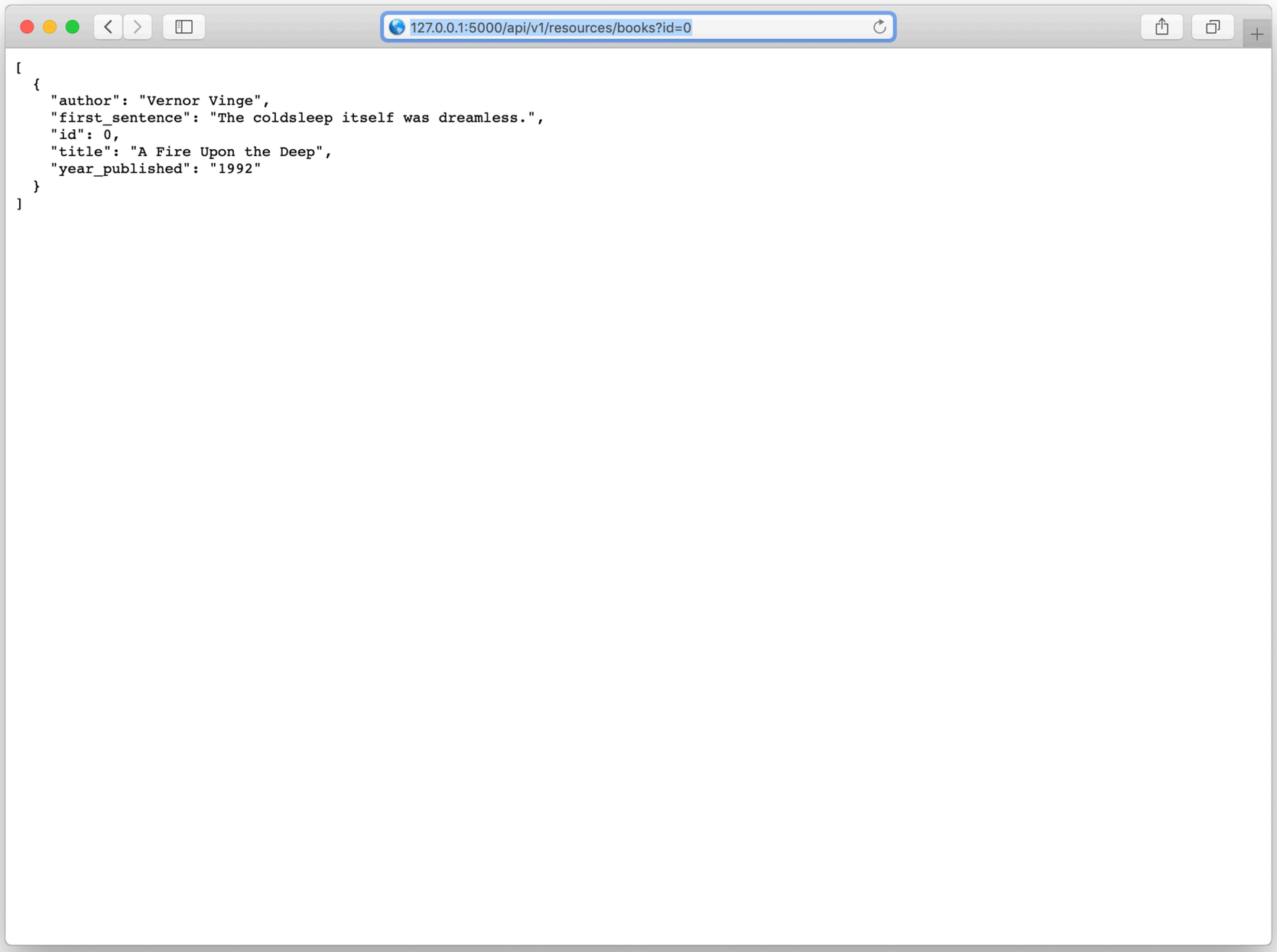
```
@app.route('/', methods=['GET'])
def home():
    return '''<h1>Distant Reading Archive</h1>
<p>A prototype API for distant reading of science fiction novels.</p>'''


@app.route('/api/v1/resources/books/all', methods=['GET'])
def api_all():
    return jsonify(books)


@app.route('/api/v1/resources/books', methods=['GET'])
def api_id():
    # Check if an ID was provided as part of the URL.
    # If ID is provided, assign it to a variable.
    # If no ID is provided, display an error in the browser.
    if 'id' in request.args:
        id = int(request.args['id'])
    else:
        return "Error: No id field provided. Please specify an id."

    # Create an empty list for our results
    results = []

    # Loop through the data and match results that fit the requested ID.
    # IDs are unique, but other fields might return many results
    for book in books:
        if book['id'] == id:
            results.append(book)

    # Use the jsonify function from Flask to convert our list of
    # Python dictionaries to the JSON format.
    return jsonify(results)

app.run()
```

Accéder à des ressources spécifiques

- Après avoir saisi, sauvegardé et exécuté le code sous Spyder 3, on peut saisir les adresses suivantes dans le navigateur :

127.0.0.1:5000/api/v1/resources/books?id=0
127.0.0.1:5000/api/v1/resources/books?id=1
127.0.0.1:5000/api/v1/resources/books?id=2
127.0.0.1:5000/api/v1/resources/books?id=3

Accéder à des ressources spécifiques



Accéder à des ressources spécifiques

- Chacun de ces adresses renvoie un résultat différent, excepté la dernière, qui renvoie une liste vide, puisqu'il n'y a pas de livre d'identifiant 3.

- Dans la suite, on va examiner notre nouvelle API en détail.

Comprendre la nouvelle API

- Nous avons créé une nouvelle fonction api_root, avec l'instruction @app_route, appliquée sur le chemin /api/v1/resources/books.

- Ainsi, la fonction est exécutée dès lors qu'on accède à http://127.0.0.1:5000/api/v1/resources/books.

- Notons qu'accéder au lien sans spécifier d'ID, renvoie le message d'erreur spécifié dans le code : Error: No id field provided. Please specify an id.

Comprendre la nouvelle API

- Dans notre fonction, on fait deux choses :

- On commence par examiner l'URL fournie à la recherche d'un identifiant, puis on sélectionne le livre qui correspond à l'identifiant.

- L'ID doit être fourni avec la syntaxe ?id=0, par exemple.

- Les données passée à l'URL de cette façon sont appelées paramètres de requête. Ils sont une des caractéristique du protocole HTTP.

Comprendre la nouvelle API

- La partie suivante des code détermine s'il y a un paramètre de requête du type ?id=0, puis affecte l'ID fourni à une variable :

```
if 'id' in request.args:
        id = int(request.args['id'])
    else:
        return "Error: No id field
provided. Please specify an id."
```

Comprendre la nouvelle API

- Ensuite, on parcourt le catalogue de livres, on identifie le livre ayant l'ID spécifié et on le rajoute à la liste renvoyée en résultat.

```
for book in books:
        if book['id'] == id:
            results.append(book)
```

Comprendre la nouvelle API

- Finalement, l'instruction return jsonify(results) renvoie les résultats au format JSON pour affichage dans le navigateur.

- A ce stade, on a créé une API fonctionnelle. Dans la suite, on va voir comment créé une API un peu plus complexe, qui utilise une base de données. Les principes et instructions fondamentaux resteront toutefois les mêmes.

Principes de conception d’une API

- Avant de rajouter des fonctionnalités à notre application, intéressons-nous aux décisions que nous avons prises concernant la conception de notre API.

- Deux aspects des bonnes API sont la facilité d’utilisation (usability) et la maintenabilité (maintainability). Nous garderons ces exigences présentes à l’esprit pour notre prochaine API.

Conception des requêtes

- La méthodologie la plus répandue de conception des API (API design) s'appelle REST.

- L'aspect le plus important de REST est qu'elle est basée sur quatre méthodes définies par le protocole HTTP : GET, POST, PUT et DELETE.

- Celles-ci correspondent aux quatre opérations standard effectuées sur une base de données : READ, CREATE, UPDATE et DELETE.

- Dans la suite, on ne s'intéressera qu'aux requêtes GET, qui permettent de lire dans une base de données.

Conception des requêtes

- Les requêtes HTTP jouant un rôle essentiel dans le cadre de la méthodologie REST, de nombreux principes de conception gravitent autour de la façon dont les requêtes doivent perte formatées.

- On a déjà créé une requête HTTP, qui renvoyait l'intégralité de notre catalogue.

- Commençons par une requête mal conçue :

http://api.example.com/getbook/10

Conception des requêtes

- Cette requête pose un certain nombre de problèmes : le premier est sémantique ; dans une API REST, les verbes typiques sont GET, POST, PUT et DELETE, et sont déterminés par la méthode de requête plutôt que par l'URL de requête. Cela entraine que le mot « get » ne doit pas apparaître dans la requête, puisque « get » est impliqué par le fait qu'on utilise une requête HTTP GET.

- De plus, les collections de ressources, comme books ou users, doivent être désignées par des noms au pluriel.

- Cela permet d'identifier facilement si l'API se réfère à un ensemble de livres (books) ou à un livre particulier (book).

Conception des requêtes

- Ces remarques présentes à l'esprit, la nouvelle forme de notre requête est la suivante :

`http://api.example.com/books/10`

- La requête ci-dessus utilise une partie du chemin pour fournir l'identifiant.
- Bien que cette approche soit utilisée en pratique, elle est trop rigide : avec des URL construites de cette façon, on ne peut filtrer que par un champ à la fois.

Conception des requêtes

- Les paramètres de requêtes permettent de filtrer selon plusieurs champs de la base de données et de spécifier des données supplémentaires, comme un format de réponse :

```
http://api.example.com/books?author=Ursula+K.
+Le Guin&published=1969&output=xml
```

Conception des requêtes

- Quand on met au point la structure des requêtes soumises à une API, il est également raisonnable de prévoir les développement futurs.
- Bien que la version actuelle de l'API de fournisse de l'information que sur un type de ressources (books), il fait sense d'envisager qu'on puisse envisager de rajouter d'autres ressources ou des fonctionnalités à notre API, ce qui donne :

```
http://api.example.com/resources/books?id=10
```

Conception des requêtes

- Spécifier un segment « resources » sur le chemin permet d'offrir aux utilisateurs l'option d'accéder à toutes les ressources disponibles, avec des requêtes du type :

```
https://api.example.com/v1/resources/images?id=10
https://api.example.com/v1/resources/all
```

Conception des requêtes

- Un autre façon d'envisager les développements futurs de l'API est de rajouter un numéro de version au chemin.

- Cela permet de continuer à maintenir l'accès à l'ancienne API si on est amené à concevoir une nouvelle version, par exemple la v2, de l'API.

- Cela permet aux applications et aux scripts conçus avec la première version de l'API de continuer à fonctionner après la mise à jour.

Conception des requêtes

- Finalement, une requête bien conçue, dans les cadre de la méthodologie REST, ressemble à :

```
https://api.example.com/v1/resources/books?id=10
```

Exemples et documentation

- Sans documentation, même la meilleure API est inutilisable.
- Une documentation doit être associée à l'API, qui décrit les ressources ou fonctionnalités disponibles via l'API et fournit des exemples concrets d'URLs de requête et de code.
- Il est nécessaire de prévoir un paragraphe pour chaque ressource, qui décrit les champs pouvant être requêtés, comme title et id.
- Chaque paragraphe doit fournir un exemple de requête HTTP ou un bloc de code.

Exemples et documentation

- Une pratique usuelle de la documentation des API consiste à annoter le code, les annotations pouvant être automatiquement regroupées en une documentation à l'aide d'outils comme Doxygen (http://www.doxygen.nl) ou Sphinx (http://www.sphinx-doc.org/en/stable/).

- Ces outils génèrent une documentation à partir des « docstrings », c'est-à-dire des chaînes de caractères documentant les fonctions.

- Bien que ce type de documentation soit précieux, on ne doit pas s'arrêter là. Il faut se mettre dans la peau d'un utilisateur et fournir des exemples concrets d'utilisation.

Exemples et documentation

- Idéalement, on doit disposer de trois types de documentation : une référence détaillant chaque route et son comportement ; un guide expliquant la référence en prose ; un ou deux tutoriels expliquant en détail chaque étape.

- Pour un exemple de documentation d'API, on pourra consulter le New York Public Library Digital Collections API (http://api.repo.nypl.org), qui constitue un bon standard. Un autre exemple, plus fourni, est celui de la MediaWiki Action API (https://www.mediawiki.org/wiki/API:Main_page) qui fournit de la documentation permettant aux utilisateurs de soumettre des requêtes partielles à l'API.

Exemples et documentation

- Pour d’autres exemples de documentation d’API, on pourra consulter l’API de la Banque Mondiale (https://datahelpdesk.worldbank.org/knowledgebase/articles/889392-api-documentation) et l’API European Pro (https://pro.europeana.eu/resources/apis).

Connexion à une base de données

- Dans l’exemple qui va suivre, on va voir comment connecter notre API à une base de donner, gérer les erreurs et autoriser le filtrage des livres par date de publication.

- La base de données utilisée est SQLite, un moteur de base de données très léger et disponible sous Python par défaut.

- L’extension standard des fichiers SQLite est .db.

Connexion à une base de données

- Les données sur les ouvrages se trouvent dans la base books.db (disponible sur le forum). On commence par la recopier dans notre répertoire api.

- La version finale de notre API requêtera cette base de données afin de renvoyer les résultats voulus aux utilisateurs.

- Le code correspondant est le suivant :

Connexion à une base de données

```
import flask
from flask import request, jsonify
import sqlite3

app = flask.Flask(__name__)
app.config["DEBUG"] = True

def dict_factory(cursor, row):
    d = {}
    for idx, col in enumerate(cursor.description):
        d[col[0]] = row[idx]
    return d
```

Connexion à une base de données

```
@app.route('/', methods=['GET'])
def home():
    return '''<h1>Distant Reading Archive</h1>
<p>A prototype API for distant reading of science fiction novels.</p>'''


@app.route('/api/v1/resources/books/all', methods=['GET'])
def api_all():
    conn = sqlite3.connect('books.db')
    conn.row_factory = dict_factory
    cur = conn.cursor()
    all_books = cur.execute('SELECT * FROM books;').fetchall()

    return jsonify(all_books)
```

Connexion à une base de données

```
def page_not_found(e):
    return "<h1>404</h1><p>The resource could not be found.</p>", 404


@app.route('/api/v1/resources/books', methods=['GET'])
def api_filter():
    query_parameters = request.args

    id = query_parameters.get('id')
    published = query_parameters.get('published')
    author = query_parameters.get('author')

    query = "SELECT * FROM books WHERE"
    to_filter = []

    if id:
        query += ' id=? AND'
        to_filter.append(id)
    if published:
        query += ' published=? AND'
        to_filter.append(published)
    if author:
        query += ' author=? AND'
        to_filter.append(author)
    if not (id or published or author):
        return page_not_found(404)

    query = query[:-4] + ';'

    conn = sqlite3.connect('books.db')
    conn.row_factory = dict_factory
    cur = conn.cursor()

    results = cur.execute(query, to_filter).fetchall()

    return jsonify(results)

app.run()
```

Connexion à une base de données

- On saisit le code sous Spyder 3 et on le sauvegarde sous le nom api-final.py dans notre répertoire api.

- On l'exécute ensuite sous Spyder 3.

- Si une version précédente de l'API est encore en cours d'exécution, on doit la terminer en faisant Control-C.

- Une le programme exécuté, on peut soumettre des requêtes du type :

Connexion à une base de données

http://127.0.0.1:5000/api/v1/resources/books/all
http://127.0.0.1:5000/api/v1/resources/books?author=Connie+Willis
http://127.0.0.1:5000/api/v1/resources/books?author=Connie+Willis&published=1999
http://127.0.0.1:5000/api/v1/resources/books?published=2010

Connexion à une base de données

- La base de données books compte 67 enregistrement, un pour chaque vainqueur du prix Hugo de science-fiction entre 1953 et 2014.

- Les données comprennent le titre du roman, l'auteur, l'année de publication et la première phrase.

- Notre API permet de filtrer selon trois champs : id, published (année de publication) et author.

Connexion à une base de données

- La nouvelle API répond aux requêtes des utilisateurs en extrayant l'information de la base de données à l'aide de requêtes SQL.

- Elle permet également de filtrer selon plus d'un champ.

- On verra une application de cette fonctionnalité dans la suite, après avoir examiné le code plus en détail.

Comprendre la nouvelle API

- Les bases de données relationnelles permettent de stocker et de récupérer des données, ces dernières étant mises sous la forme de tables.
- Les tables sont semblables à des feuilles de calcul, en cela qu’elles ont des lignes et des colonnes, les colonnes indiquant ce à quoi correspondent les données, par exemple à un titre ou à une date. Les lignes représentent des données individuelles, qui peuvent des livres, des utilisateurs, des transactions, ou tout autre type d’entité.

Comprendre la nouvelle API

- Notre base de données compte une table, composée de cinq colonnes : id, published, author, title et first_sentence.

- Chaque ligne représente un roman ayant remporté le prix hugo.

- Plutôt que de spécifier les données dans le code, notre fonction api_all les extrait de la base de données books :

Comprendre la nouvelle API

```
def api_all():
    conn = sqlite3.connect('books.db')
    conn.row_factory = dict_factory
    cur = conn.cursor()
    all_books = cur.execute('SELECT * FROM books;').fetchall()

    return jsonify(all_books)
```

Comprendre la nouvelle API

- On commence par se connecter à la base de données en utilisant la bibliothèque sqlite3.

- Un objet représentant la connexion à la base de données est lié à la variable conn.

- L'instruction conn.row_factory = dict_factory dit à l'objet correspondant à la connexion d'utiliser la fonction dict_factory, qui renvoie les résultats sous forme de dictionnaires plutôt que de listes - ce qui se convertit mieux au format JSON.

Comprendre la nouvelle API

- On crée ensuite un objet curseur (cur = conn.cursor( )), qui parcourt la base de données pour extraire les données.

- Finalement, on exécute une requête SQL à l'aide de la méthode cur.execute pour extraire toutes les données disponibles (*) de la table books de notre base de données.

- A la fin dénoter fonction, les données récupérées sont converties au format JSON : jsonify(all_books).

Comprendre la nouvelle API

- L’autre fonction renvoyant des données, api_filter, utilise la même approche pour extraire les données de la base.

- Le but de la fonction page_not_found est de créer une page d’erreur affichée à l’utilisateur s’il spécifie une route non prise ne charge par l’API :

```
@app.errorhandler(404)
def page_not_found(e):
    return "<h1>404</h1><p>The resource could not be found.</p>
```

Comprendre la nouvelle API

- Dans les résultats HTML, le code 200 signifie « OK » (données transférées) alors que le code 404 signifie « not found » (pas de ressources disponibles à l'adresse spécifiée).

- La fonction page_not_found permet de renvoyer 404 si quelque chose se passe mal.

Comprendre la nouvelle API

- La fonction api_filter est une amélioration de la fonction api_id qui renvoyait un roman en se basant sur son identifiant.

- Elle permet de filtrer selon trois champs : id, published et author.

- Elle commence par identifier tous les paramètres de requête fournis dans l'URL, à l'aide de l'instruction :

```
query_parameters = request.args
```

Comprendre la nouvelle API

- Elle récupère ensuite les valeurs des paramètres et les lie à des variables :

```
id = query_parameters.get('id')
published = query_parameters.get('published')
author = query_parameters.get('author')
```

Comprendre la nouvelle API

- La portion de code suivante permet de construire une requête SQL qui est utilisée pour extraire l'information recherchée de la base de données.

- Les requêtes SQL ont la forme :

```
SELECT <columns> FROM <table> WHERE <column=match> AND <column=match>;
```

Comprendre la nouvelle API

- Afin d'obtenir les données recherchées, on doit construire à la fois une requête SQL du type précédent et une liste avec les filtres spécifiés.
- On commence par définir la requête et la liste de filtres :

```
query = "SELECT * FROM books WHERE"
to_filter = []
```

Comprendre la nouvelle API

- Alors, si id, published et author ont été passés en paramètres de requête, on les rajoute à la requête et à la liste de filtres :

```
    if id:
        query += ' id=? AND'
        to_filter.append(id)
    if published:
        query += ' published=? AND'
        to_filter.append(published)
    if author:
        query += ' author=? AND'
        to_filter.append(author)
```

Comprendre la nouvelle API

- Si l'utilisateur n'a spécifié aucun de ces paramètres de requête, on renvoie la page d'erreur 404 :

```
if not (id or published or author):
        return page_not_found(404)
```

Comprendre la nouvelle API

- Afin de parfaire notre requête SQL, on supprime le dernier AND et on complète la requête par le point-virgule requis par SQL :

```
query = query[:-4] + ';'
```

Comprendre la nouvelle API

- Ensuite, on se connecte à la base de données, puis on exécute la requête SQL construite à l'aide de notre liste de filtres :

```
conn = sqlite3.connect('books.db')
conn.row_factory = dict_factory
cur = conn.cursor()

results = cur.execute(query,
  to_filter).fetchall()
```

Comprendre la nouvelle API

- Finalement, on renvoie les résultats au format JSON à l’utilisateur :

```
return jsonify(results)
```

Utilisation de la nouvelle API

- Notre nouvelle API autorise des requêtes plus sophistiquées de la part des utilisateurs.

- De plus, dès que de nouvelles données sont rajoutées à la base, elles deviennent immédiatement disponibles pour les projets construits à l'aide de l'API.

- Supposons par exemple, qu'un nuage de points représentant la longueur de la première phrase en fonction de l'année fasse usage dénoter API.

Utilisation de la nouvelle API

- Au fur et à mesure qu'on rajoute des vainqueurs du prix Hugo à notre base, le graphique fait immédiatement usage des nouvelles données. Il représente les données en temps réel.

- Dans bien des cas, il est pertinent de commencer par créer une interface de type API pour les données d'un projet avant de construire une visualisation, une application ou un site web basé sur les données.

Python et technologies Web

2. Construire des API avec Python, Flask, Swagger et Connexion

Salim Lardjane
Université de Bretagne Sud

Introduction

- On va voir dans la suite comment construire des API REST avec validation des inputs et des outputs et documentation automatiquement générée par Swagger.

- On verra également comment utiliser l'API avec JavaScript pour mettre à jour le DOM (Document Object Model) c'est-à-dire la structure de la page web affichée.

Introduction

- L’API REST qu’on va construire permettra de manipuler un catalogue de personnes, les personnes étant identifiées par leur nom de famille.

- Les mises à jour seront marquées avec un « timestamp » (date/heure).

- On pourrait manipuler le catalogue sous la forme d’une base de données, d’un fichier, ou via un protocole réseau, mais pour simplifier on utilisera des données en mémoire.

- Un des objectifs des APIs est de découpler les données des applications qui les utilisent, en rendant transparents les détails de l’implémentation des données.

REST

- On a mentionné la méthodologie REST dans le dernier cours.

- On va à présent l'examiner plus en détail.

- On peut voir REST comme un ensemble de conventions tirant parti du protocole HTTP pour fournir des outils CRUD (Create, Read, Update, Delete) agissant sur des choses ou des collections de choses.

- CRUD peut être mis en correspondance avec HTTP de la façon suivante :

REST

- Action : Create, Verbe HTTP : POST, Description : Créer une chose nouvelle unique.

- Action : Read, Verbe HTTP : GET, Description : Lire l’information sur une chose ou une collection de choses

- Action : Update, Verbe HTTP : PUT, Description : Mettre à jour l’information sur une chose

- Action : Delete, Verbe HTTP : DELETE, Description : Supprimer une chose.

REST

- On peut effectuer chacune de ces action sur une chose ou une ressource.

- On peut voir une ressource comme une chose à laquelle on peut associer un nom : une personne, une adresse, une transaction.

- L’idée de ressource peut être mise en correspondance avec celle d’URL, comme on l’a vu lors du dernier cours.

REST

- Une URL doit identifier une ressource unique sur le Web, quelque chose qui correspond toujours à la même URL.

- Avec les concepts de ressource et d’URL, on est à même de concevoir des applications agissant sur des choses identifiées de façon unique sur le Web.

- C’est une première étape vers la conception d’APIs à proprement parler, permettant d’effectuant les actions souhaitées via le réseau.

Ce que REST n’est pas

- La méthodologie REST étant très utile et aidant à mettre au point la façon dont on interagit avec une API, elle est parfois utilisée à tort, pour des problèmes auxquels elle n’est pas adaptée.

- Dans de nombreux cas, on souhaite effectuer une action directement.

- Prenons l’exemple de la substitution d’une chaîne de caractères.

Ce que REST n’est pas

- Un exemple d’URL permettant de le faire est :

/api/substituteString/<string>/<search_string>/<sub_string>

- Ici, <string> désigne la chaîne sur laquelle on veut faire la substitution, <search_string> est la sous-chaîne à remplacer et <sub_string> est la nouvelles sous-chaîne.

- On peut très certainement concevoir un code sur le serveur qui permette d’effectuer l’opération souhaitée, mais cela pose des problème en termes de REST.

Ce que REST n’est pas

- Premier problème : l’URL ne pointe pas vers une ressource unique ; ainsi ce qu’elle renvoie dépend entièrement du chemin spécifié
- Deuxième problème : Aucun concept CRUD ne correspond à cette URL.
- Troisième problème : la signification des variables de chemin dépend de leur position dans l’URL.
- On pourrait corriger cela en modifiant l’URL de façon à utiliser une requête :

Ce que REST n’est pas

/api/substituteString?
string=<string>&search_string=<search_string>&sub_string=<sub_string>

- Mais la portion d’URL /api/substituteString ne désigne pas une chose (n’est pas un nom) ou une collection de choses : elle désigne une action (c’est un verbe).

- Cela ne rentre pas dans le cadre des conventions REST et de ce fait ne correspond pas à une API.

- Ce que l’URL précédente représente est en fait une RPC (Remote Procedure Call).

Ce que REST n’est pas

- Conceptualiser ce qu’on veut faire comme une RPC est plus pertinent.

- Cela d’autant plus que les conventions REST et RPC peuvent coexister sans problème au sein d’une API.

- Il y a de nombreuses situations où on souhaite effectuer des opérations CRUD sur quelque chose. Il existe également de nombreuses situations où l’on souhaite effectuer une action sur une chose (comme un paramètre) mais sans nécessairement affecter la chose elle-même.

Conception d’une API REST

- Dans l’exemple suivant, on va mettre au point une API REST permettant d’accéder à un catalogue de personnes et d’effectuer des opérations CRUD sur celui-ci.

- Voici le design de l’API :

Conception d’une API REST

- Action : Create, Verbe HTTP : POST, Chemin d’URL : /api/people, Description : définit une URL unique pour créer une nouvelle personne

- Action : Read, Verbe HTTP : GET, Chemin d’URL : /api/people, Description : définit une URL unique pour lire le catalogue de personnes

- Action : Read, Verbe HTTP : GET, Chemin d’URL : /api/people/Farrell, Description : définit une URL unique pour lire les données d’une personne particulière du catalogue

Conception d’une API REST

- Action : Update, Verbe HTTP : PUT, Chemin d’URL : /api/people/Farrell, Description : définit une URL unique pour mettre à jour les données d’une personne du catalogue

- Action : Delete, Verbe HTTP : DELETE, Chemin d’URL : /api/people/Farrell, Description : définit une URL unique pour supprimer les données d’une personne du catalogue

Premiers pas

- On va commencer par créer un serveur web très simple à l’aide du micro-cadre de développement Flask.

- On crée un répertoire api, un sous-répertoire version_1, puis un sous-répertoire templates ou on sauvegarde les programmes suivants :

Premiers pas

Python (server.py)

```
from flask import (
    Flask,
    render_template
)

# Create the application instance
app = Flask(__name__, template_folder="templates")

# Create a URL route in our application for "/"
@app.route('/')
def home():
    """
    This function just responds to the browser ULR
    localhost:5000/

    :return:        the rendered template 'home.html'
    """
    return render_template('home.html')

# If we're running in stand alone mode, run the application
if __name__ == '__main__':
    app.run(debug=True)
```

Premiers pas

HTML (templates/home.html)

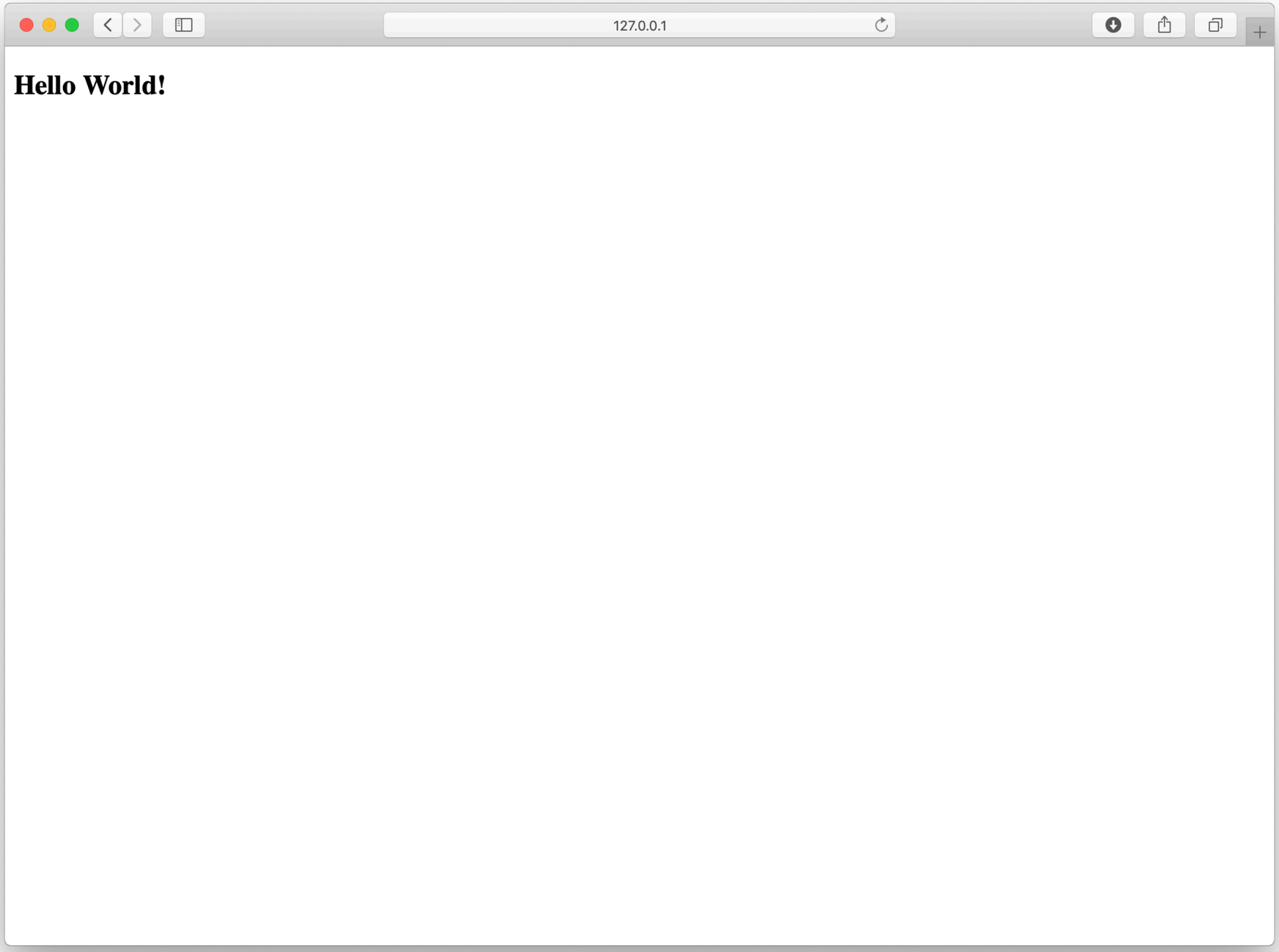
```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Application Home Page</title>
</head>
<body>
    <h2>
        Hello World!
    </h2>
</body>
</html>
```

Premiers pas

- Le programme HTML est nommé home.html plutôt que index.html comme il est d'usage, car index.html pose problème lorsqu'on importe le module Connexion, ce qu'on fera dans la suite.

- On obtient le résultat suivant en saisissant http://127.0.0.1:5000 dans la barre du navigateur :

Premiers pas



Connexion

- A présent qu'on a un service Web fonctionnel, on va ajouter un chemin d'accès REST (REST API endpoint).

- A cet effet, on va installer le module Connexion.

- Pour cela, on saisit dans une fenêtre terminal : conda install -c conda-forge connexion. On peut à présent l'importer sous Python.

- On sauvegarde ensuite les programmes suivants (sous-répertoire version_2 sur le forum) :

Connexion

PYTHON (server.py)

```python
from flask import render_template
import connexion

# Create the application instance
app = connexion.App(__name__, specification_dir='./')

# Read the swagger.yml file to configure the endpoints
app.add_api('swagger.yml')

# Create a URL route in our application for "/"
@app.route('/')
def home():
    """
    This function just responds to the browser ULR
    localhost:5000/
    :return:        the rendered template 'home.html'
    """
    return render_template('home.html')

# If we're running in stand alone mode, run the application
if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000, debug=True)
```

Connexion

YAML (swagger.yml)

```yaml
swagger: "2.0"
info:
  description: This is the swagger file that goes with our server code
  version: "1.0.0"
  title: Swagger REST Article
consumes:
  - "application/json"
produces:
  - "application/json"

basePath: "/api"
```

Connexion

```yaml
# Paths supported by the server application
paths:
  /people:
    get:
      operationId: "people.read"
      tags:
        - "People"
      summary: "The people data structure supported by the server application"
      description: "Read the list of people"
      responses:
        200:
          description: "Successful read people list operation"
          schema:
            type: "array"
            items:
              properties:
                fname:
                  type: "string"
                lname:
                  type: "string"
                timestamp:
                  type: "string"
```

Connexion

- Afin d'utiliser Connexion, on commence par l'importer, puis on crée l'application à l'aide de Connexion plutôt que Flask. L'application Flask est quand même créée de façon sous-jacente mais avec des fonctionnalités additionnelles.

- La création de l'instance de l'application comprend un paramètre specification_dir. Il indique à Connexion où trouver le fichier de configuration, ici le répertoire courant.

- La ligne app.add_api('swagger.yml') indique à l'instance de lire le fichier swagger.yml dans le specification_dir et de configurer le système pour Connexion.

Connexion

- Le fichier swagger.yml est un fichier YAML (ou JSON) contenant l'information nécessaire à la configuration du serveur, permettant d'assurer la validation des inputs, des outputs, de fournir les URL de requête et de configurer l'UI Swagger.

- Le code swagger.yml précédent définit la requête GET /api/people.

- Le code est organisé de façon hiérarchique, l'indentation définissant le degré d'appartenance ou portée.

Connexion

- Par exemple, paths définit la racine de toutes les URL de l'API.

- La valeur /people indentée dessous définit la racine de toutes les URL /api/people.

- Le get : indenté dessous définit une requête GET vers l'URL /api/people, et ainsi de suite pour tout le fichier de configuration.

Connexion

- Dans le fichier swagger.yml, on configure Connexion avec la valeur operationId pour appeler le module people et la fonction read de ce module quand l'API reçoit une requête HTTP du type GET /api/people.

- Cela signifie qu'un module people.py doit exister et contenir une fonction read( ).

- Voici le module people.py :

Connexion

PYTHON (people.py)

```
from datetime import datetime

def get_timestamp():
    return datetime.now().strftime(("%Y-%m-%d %H:%M:%S"))

# Data to serve with our API
PEOPLE = {
    "Farrell": {
        "fname": "Doug",
        "lname": "Farrell",
        "timestamp": get_timestamp()
    },
    "Brockman": {
        "fname": "Kent",
        "lname": "Brockman",
        "timestamp": get_timestamp()
    },
    "Easter": {
        "fname": "Bunny",
        "lname": "Easter",
        "timestamp": get_timestamp()
    }
}
```

Connexion

```
# Create a handler for our read (GET) people
def read():
    """
    This function responds to a request for /api/people
    with the complete lists of people

    :return:        sorted list of people
    """
    # Create the list of people from our data
    return [PEOPLE[key] for key in sorted(PEOPLE.keys())]
```

Connexion

- Dans le module people.py apparaît d'abord la fonction get_timestamp( ) qui génère une représentation de la date/heure courante sous la forme d'une chaîne de caractères.

- Elle est utilisée pour modifier les données quand une requête de modification est transmise à l'API.

- Ensuite, on définit un dictionnaire PEOPLE qui est une simple base de données de noms, ayant comme clef le nom de famille.

Connexion

- PEOPLE est une variable de module et sont état persiste donc entre les appels à l'API.
- Dans une application réelle, les données PEOPLE se trouveraient dans une base de données, un fichier ou une ressource Web, c'est-à-dire qu'elles persisteraient indépendamment de l'exécution de l'application.

Connexion

- Vient ensuite la fonction read( ) qui est appelée lorsque le serveur reçoit une requête HTTP GET pour /api/people.

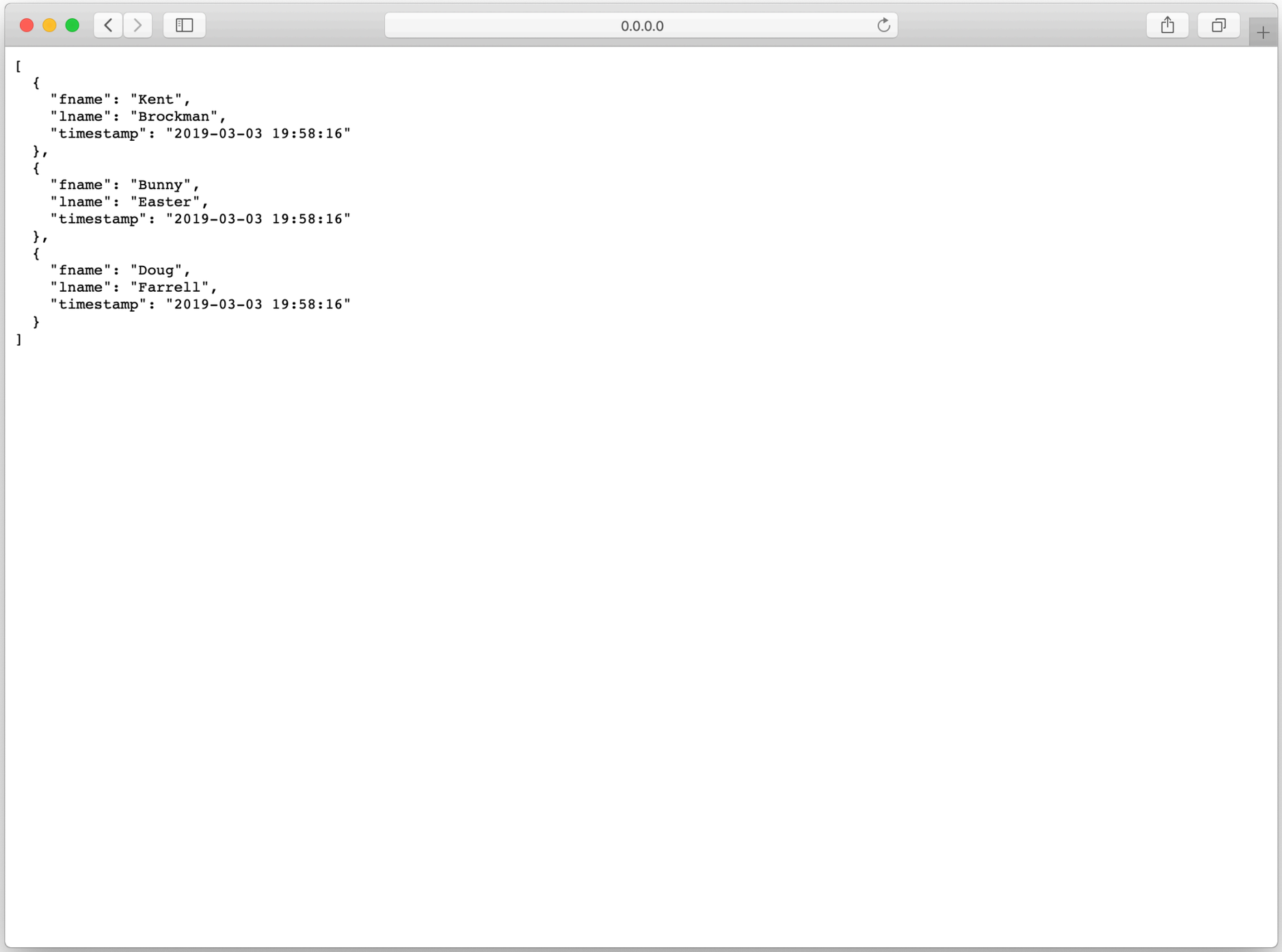
- La valeur de retour de cette fonction est convertie au format JSON (comme spécifié par produces : dans le fichier de configuration swagger.yml).

- Elle consiste en la liste des personnes triée par ordre alphabétique du nom de famille.

Connexion

- A ce stade, on peut exécuter le programme server.py, ce qui donne le résultat suivant en saisissant comme adresse http://0.0.0.0:5000/api/people dans la barre du navigateur.

Connexion


0.0.0.0
[
{
"fname": "Kent",
"lname": "Brockman",
"timestamp": "2019-03-03 19:58:16"
},
{
"fname": "Bunny",
"lname": "Easter",
"timestamp": "2019-03-03 19:58:16"
},
{
"fname": "Doug",
"lname": "Farrell",
"timestamp": "2019-03-03 19:58:16"
}
]

Swagger

- On a ainsi construit une API simple autorisant une seule requête.

- On peut toutefois se poser la question suivante, d'autant plus qu'on a vu comment obtenir le même résultat beaucoup plus simplement avec Flask, lors du cours précédent : *qu'apporte la configuration à l'aide du fichier swagger.yml ?*

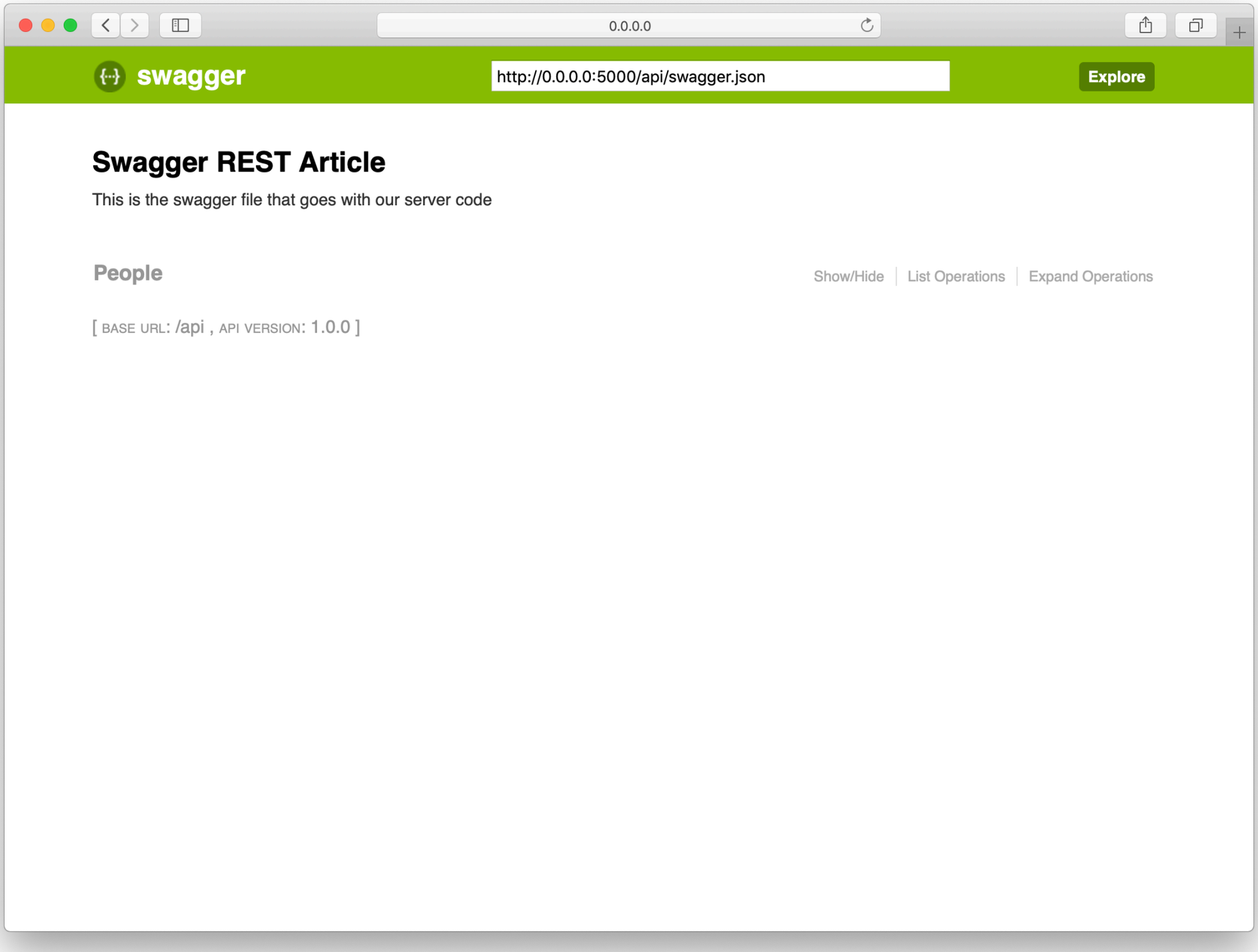
Swagger

- En fait, en plus de l'API une Interface Utilisateur (UI) Swagger a été créée par Connexion.

- Pour y accéder, il suffit de saisir http://localhost:5000/api/ui.

- On obtient le résultat suivant :

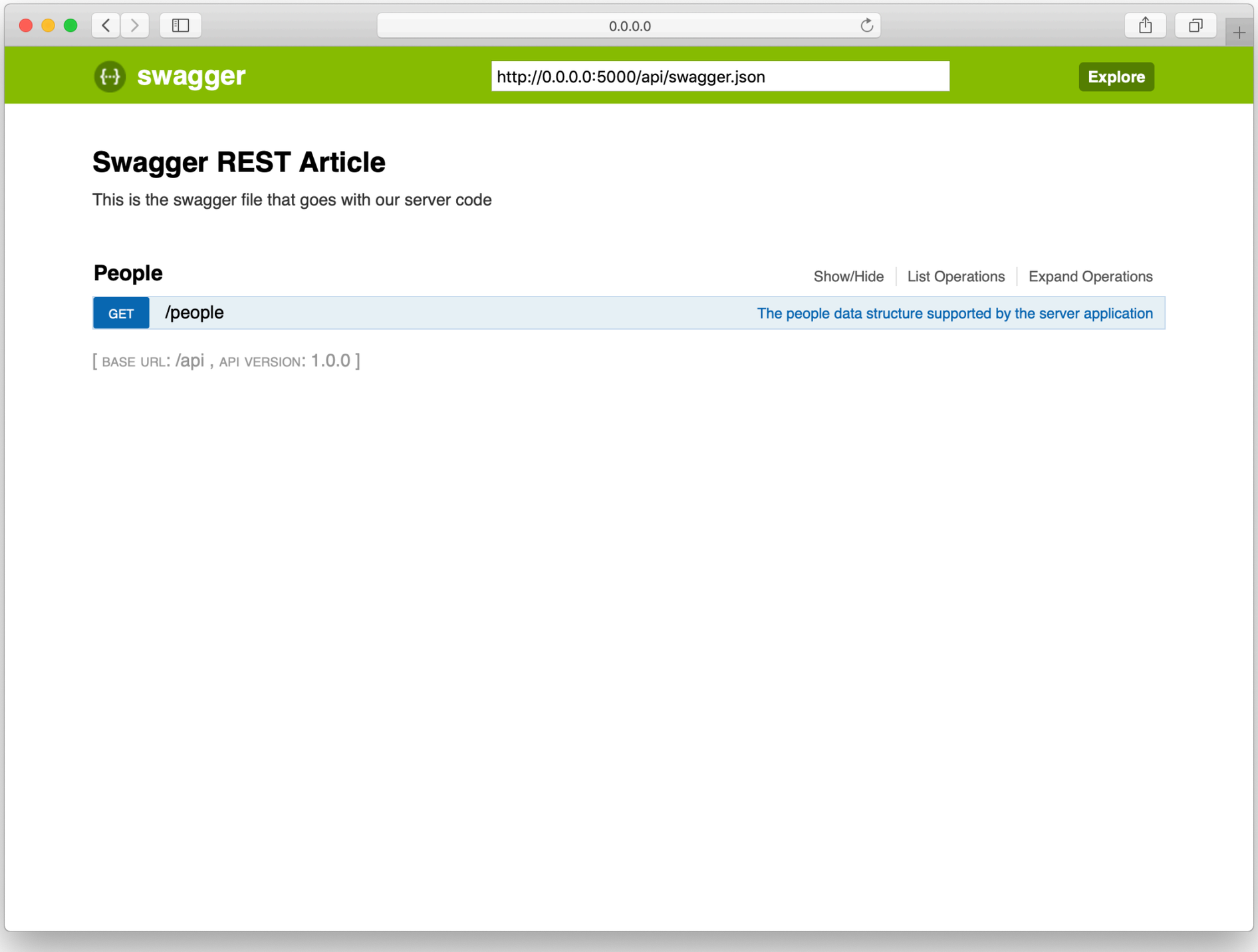
Swagger


0.0.0.0
swagger
http://0.0.0.0:5000/api/swagger.json
Explore
Swagger REST Article
This is the swagger file that goes with our server code
People
Show/Hide
List Operations
Expand Operations
[BASE URL: /api , API VERSION: 1.0.0]

Swagger

- En cliquant sur People, la requête potentielle apparaît :

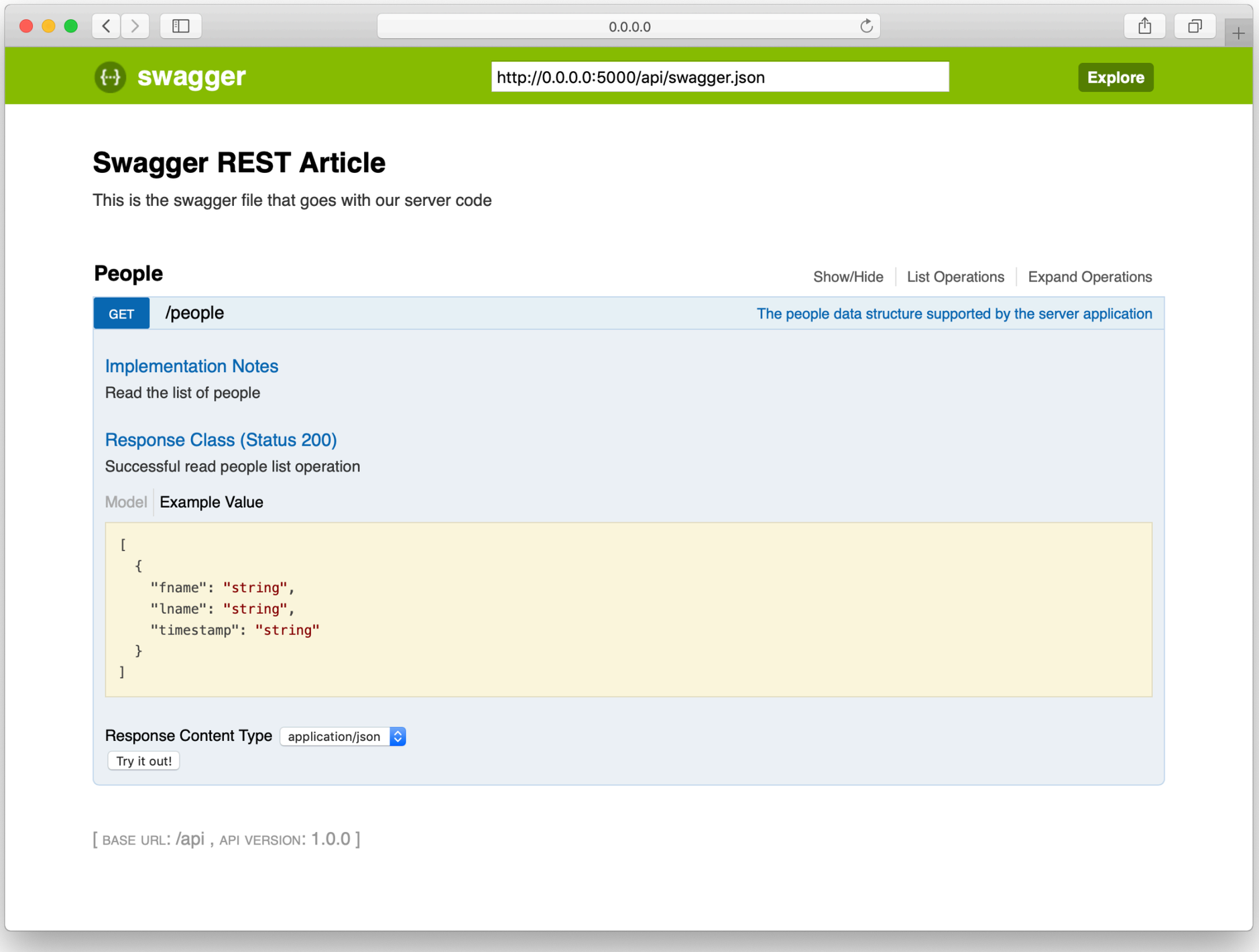
Swagger


0.0.0.0
swagger
http://0.0.0.0:5000/api/swagger.json
Explore
Swagger REST Article
This is the swagger file that goes with our server code
People
Show/Hide
List Operations
Expand Operations
GET
/people
The people data structure supported by the server application
[BASE URL: /api , API VERSION: 1.0.0]

Swagger

- En cliquant sur la requête, des informations supplémentaires sont affichées par l'interface :
  - La structure de la réponse
  - Le format de la réponse (content-type)
  - Le texte renseigné dans swagger.yml à propos de la requête

Swagger

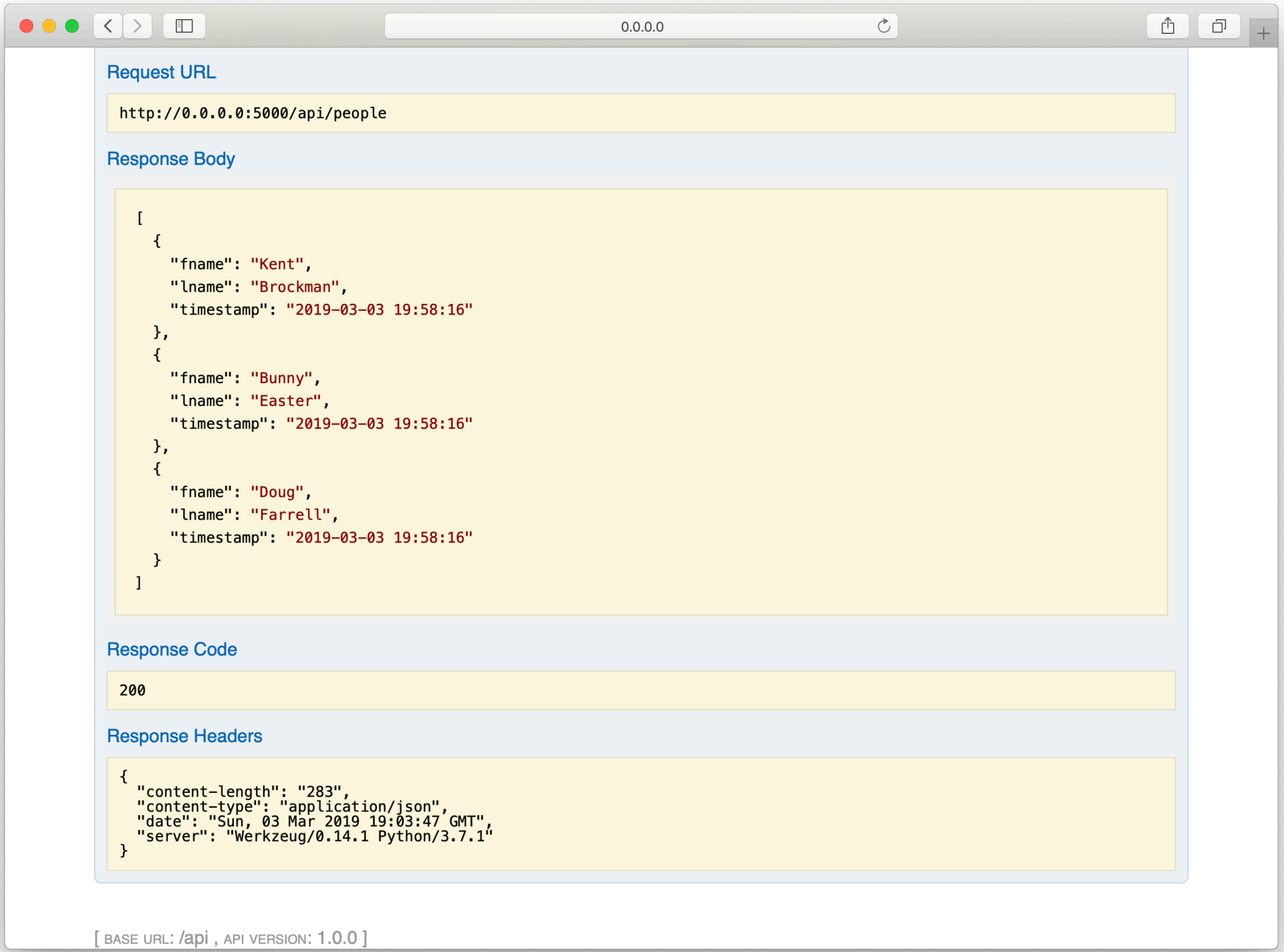

0.0.0.0
swagger
http://0.0.0.0:5000/api/swagger.json
Explore
Swagger REST Article
This is the swagger file that goes with our server code
People
Show/Hide
List Operations
Expand Operations
GET
/people
The people data structure supported by the server application
Implementation Notes
Read the list of people
Response Class (Status 200)
Successful read people list operation
Model
Example Value
[
{
"fname": "string",
"lname": "string",
"timestamp": "string"
}
]
Response Content Type
application/json
Try it out!
[BASE URL: /api , API VERSION: 1.0.0]

Swagger

- On peut même essayer la requête en cliquant sur le bouton Try It Out ! en bas de l'écran.

- L'interface développe encore plus l'affichage et on obtient à la suite :

Swagger


0.0.0.0
Request URL
http://0.0.0.0:5000/api/people
Response Body
[
{
"fname": "Kent",
"lname": "Brockman",
"timestamp": "2019-03-03 19:58:16"
},
{
"fname": "Bunny",
"lname": "Easter",
"timestamp": "2019-03-03 19:58:16"
},
{
"fname": "Doug",
"lname": "Farrell",
"timestamp": "2019-03-03 19:58:16"
}
]
Response Code
200
Response Headers
{
"content-length": "283",
"content-type": "application/json",
"date": "Sun, 03 Mar 2019 19:03:47 GMT",
"server": "Werkzeug/0.14.1 Python/3.7.1"
}
[BASE URL: /api , API VERSION: 1.0.0]

Swagger

- Toute cela est très utile lorsqu'on dispose d'une API complète puisque cela permet de tester et d'expérimenter avec l'API sans écrire de code.

- En plus d'être pratique pour accéder à la documentation, l'interface Swagger a l'avantage d'être mise à jour dès que le fichier swagger.yml est mis à jour.

- De plus, le fichier swagger.yml fournit une façon méthodique de concevoir l'ensemble des requêtes, ce qui permet d'assurer une conception rigoureuse de l'API.

Swagger

- Enfin, cela permet de découpler le code Python de la configuration des requêtes, ce qui peut s’avérer très utile lorsqu’on conçoit des API supportant des applications Web.

- On en verra un exemple dans la suite.

API complète

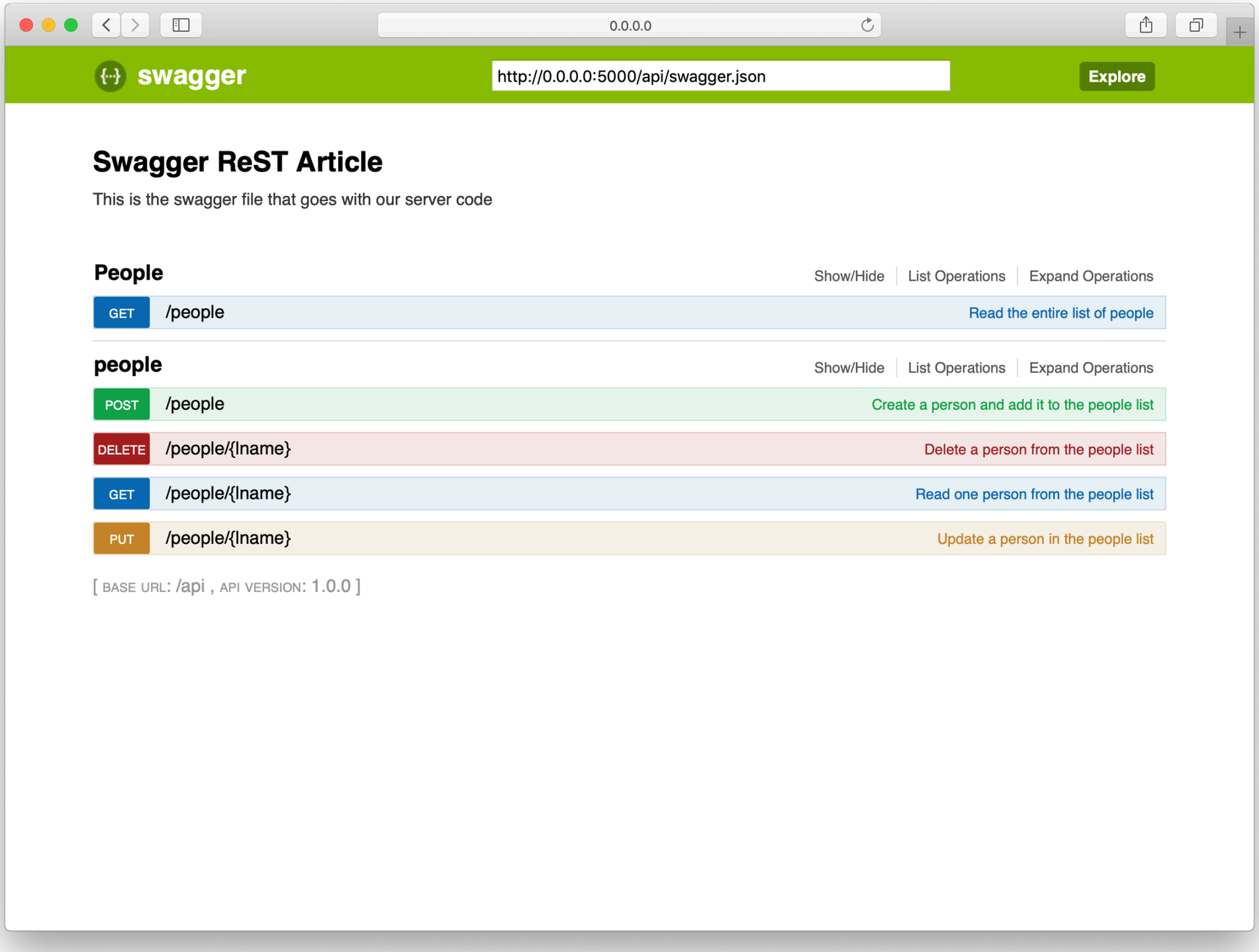
- Notre objectif était de créer une API offrant un accès CRUD à nos données PEOPLE.

- A cet effet, on complète le fichier de configuration swagger.yml et le module people.py afin de satisfaire aux spécifications de notre API.

- Les fichiers complétés sont accessibles sur le forum dans le sous-répertoire version_3 du répertoire api.

- L'exécution de server.py donne alors, pour ce qui est de l'interface Swagger :

API Complète



API complète

- L’interface précédente permet d’accéder à la documentation renseignée dans le fichier swagger.yml et d’interagir avec toutes les URL implémentant les fonctionnalités CRUD de l’API.

Application Web

- On dispose à présent d'une API REST documentée et avec laquelle on peut interagir à l'aide de Swagger.

- Mais qu'en faire à présent ?

- L'étape suivante consiste à créer une application Web illustrant l'utilisation concrète de l'API.

Application Web

- On crée une application Web qui affiche les personnes de la base à l'écran et permet d'ajouter des personnes, de mettre à jour les données des personnes et de supprimer des personnes de la base.

- Cela sera fait à l'aide de requêtes AJAX émises de JavaScript vers les URL de notre API.

Application Web

- Pour commencer, on doit compléter le fichier home.html, qui contrôle la structure et l’apparence de la page d’accueil de l’application web, de la façon suivante :

Application Web

HTML (home.html)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Application Home Page</title>
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/normalize/8.0.0/
normalize.min.css">
    <link rel="stylesheet" href="static/css/home.css">
    <script
      src="http://code.jquery.com/jquery-3.3.1.min.js"
      integrity="sha256-FgpCb/KJQlLNfOu91ta32o/NMZxltwRo8QtmkMRdAu8="
      crossorigin="anonymous">
    </script>
</head>
```

Application Web

```
<body>
    <div class="container">
        <h1 class="banner">People Demo Application</h1>
        <div class="section editor">
            <label for="fname">First Name
                <input id="fname" type="text" />
            </label>
            <br />
            <label for="lname">Last Name
                <input id="lname" type="text" />
            </label>
            <br />
            <button id="create">Create</button>
            <button id="update">Update</button>
            <button id="delete">Delete</button>
            <button id="reset">Reset</button>
        </div>
        <div class="people">
            <table>
                <caption>People</caption>
                <thead>
                    <tr>
                        <th>First Name</th>
                        <th>Last Name</th>
                        <th>Update Time</th>
                    </tr>
                </thead>
                <tbody>
                </tbody>
            </table>
        </div>
        <div class="error">
        </div>
    </div>
</body>
<script src="static/js/home.js"></script>
</html>
```

Application Web

- Le fichier précédent est disponible sur le forum, dans le sous-répertoire templates du sous-répertoire version_4 du répertoire api.

- Le programme HTML complète le programme home.html initial en y adjoignant un appel au fichier externe normalize.min.css (https://necolas.github.io/normalize.css/) , qui permet de normaliser l'affichage sur différents navigateurs.

- Il fait également appel au fichier jquery-3.3.1.min.js (https://jquery.com) qui permet de disposer des fonctionnalités jQuery en JavaScript.

- Celles-ci sont utilisées pour programmer l'interactivité de la page.

Application Web

- Le code HTML définit la partie statique de l'application.
- Les parties dynamiques seront apportées par JavaScript lors de l'accès à la page et lorsque l'utilisateur interagit avec celle-ci.

Application Web

- Le fichier home.html fait référence à deux fichier statiques : static/css/home.css et static/js/home.js.

- Ils apparaissent de la façon suivante dans l'arborescence de l'application :

```
static/
├── css/
│   └── home.css
└── js/
    └── home.js
```

Application Web

- Le répertoire nommé static est immédiatement reconnu par l'application Flask et le contenu des répertoires css et js est ainsi accessible à partir du fichier home.html.

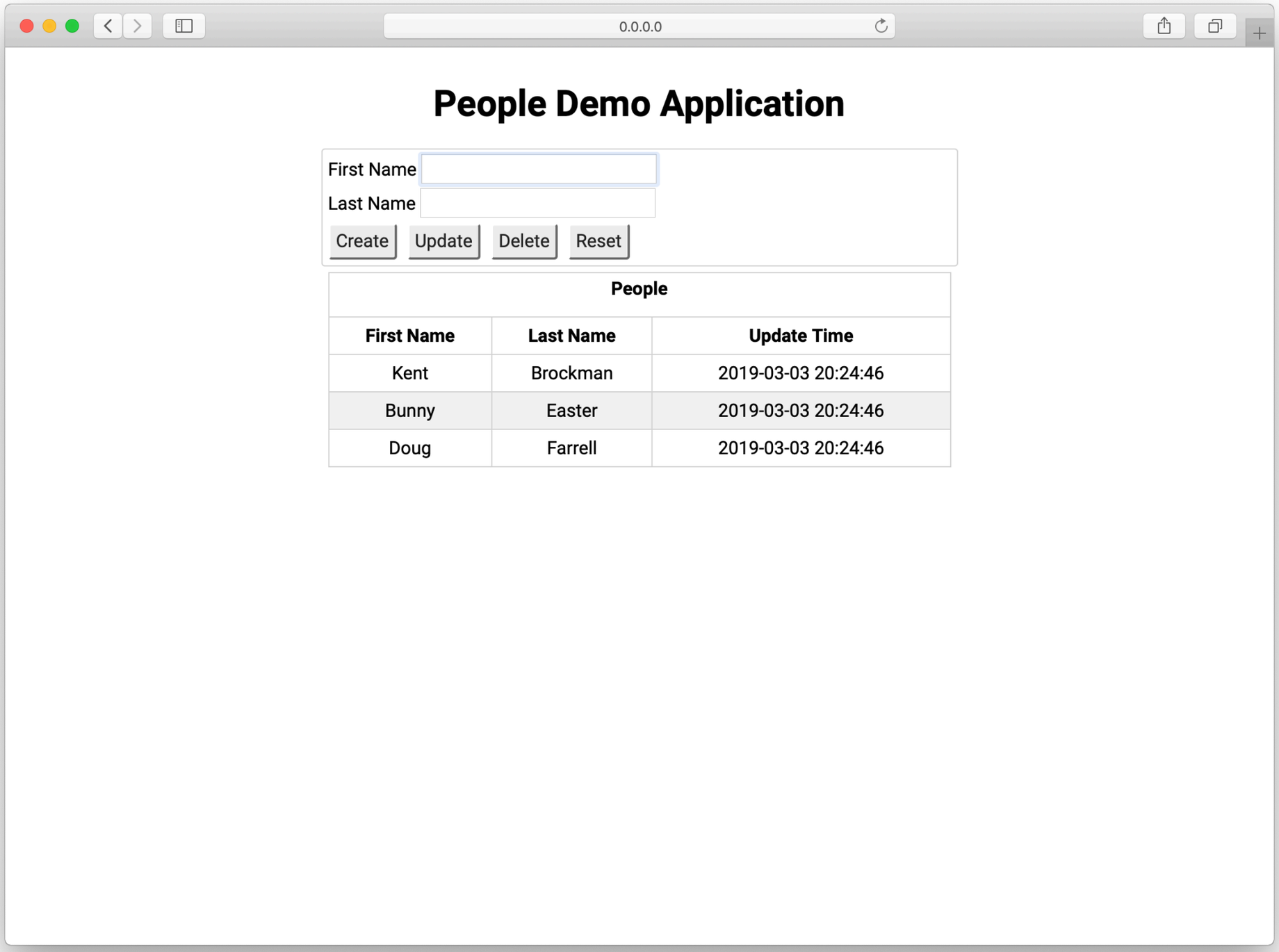
- Les fichiers home.css et home.js sont disponibles sur le forum.

- On parlera davantage de CSS et de JavaScript lors du prochain cours.

Application Web

- Lorsqu’on se place dans le sous-répertoire version_4 du répertoire api et qu’on exécute server.py, on obtient le résultat suivant dans le navigateur :

Application Web



Application Web

- Le bouton Create permet à l'utilisateur de rajouter une personne au catalogue sur le serveur.

- Un double clic sur une ligne du tableau fait apparaître le nom et le prénom dans les champs de saisie.

- Pour mettre à jour (Update), l'utilisateur doit modifier le prénom, le nom étant la clef de recherche.

- Pour supprimer une personne du catalogue, il suffit de cliquer sur Delete.

- Reset vide les champs de saisie.

Application Web

- On a ainsi construit une petite application Web fonctionnelle.

- On verra plus en détail son fonctionnement lors du prochain cours.

Python et technologies Web

3. Construire des API avec Python, Flask, Swagger, Connexion et SQLAlchemy

Salim Lardjane
Université de Bretagne Sud

Prérequis

- Commençons par installer les bibliothèques Python dont nous aurons besoin sous Anaconda, à l'aide de la commande suivante, dans une fenêtre terminal :
- conda install -c conda-forge Flask-SQLAlchemy flask-marshmallow marshmallow-sqlalchemy marshmallow
- Cette commande permet d'installer les bibliothèques suivante :

Prérequis

- Flask-SQLAlchemy, qui permet d'accéder à des bases de données de façon compatible avec Flask

- flask-marshmallow, qui permet de convertir des objets Python en structures sérialisables (c'est à dire codables comme une suite d'informations plus petites).

- marshmallow-sqlalchemy, qui permet de sérialiser et désérialiser des objets Python générés par SQLAlchemy

- marshmallow, qui fournit l'essentiel des fonctionnalités Marshmallow de sérialisation / désérialisation

Introduction

- Lors du dernier cours, on a vu comment créer une API REST autorisant les opérations CRUD et bien documentée en utilisant Python, Flask, Connexion et Swagger.

- Toutefois, les données étaient stockées dans une structure PEOPLE qui était réinitialisée à chaque démarrage du serveur.

- On va voir dans la suite comment stocker la structure PEOPLE et les actions fournies par l'API de façon permanente dans une base de données, à l'aide de SQLAlchemy et Marshmallow.

Introduction

- SQLAlchemy fournit un ORM (Object Relationnel Model, Modèle Relationnel Objet) qui permet de stocker les données des objets Python dans une représentation de type base de donnée.

- Cela permet de continuer à programme dans l'esprit Python, sans se préoccuper des détails de la représentation des données d'un objet dans la base de données.

Introduction

- Marshmallow fournit des outils permettant de sérialiser et désérialiser des objets Python au fur et à mesure qu'ils sont émis ou reçus par notre API REST JSON.

- Marshmallow convertit des instances de classe Python en des objets qui peuvent être convertis en JSON.

- Le code correspondant à ce cours se trouve dans le répertoire api2 sur le forum.

Les données

- Lors du dernier cours, on avait représenté nos données par un dictionnaire Python, de clef le nom de famille des personnes présentes dans la base.

- Le code correspondant était le suivant :

Les données

PYTHON

```
# Data to serve with our API
PEOPLE = {
    "Farrell": {
        "fname": "Doug",
        "lname": "Farrell",
        "timestamp": get_timestamp()
    },
    "Brockman": {
        "fname": "Kent",
        "lname": "Brockman",
        "timestamp": get_timestamp()
    },
    "Easter": {
        "fname": "Bunny",
        "lname": "Easter",
        "timestamp": get_timestamp()
    }
}
```

Les données

- Les modifications que l'on va effectuer vont transférer les données vers une table d'une base de données.

- Cela signifie que les données seront sauvegardées « en dur » et accessibles entre deux exécutions de server.py.

- Comme le nom de famille était la clef du dictionnaire Python PEOPLE, le code empêchait de modifier le nom d'une personne. Le transfert vers une base de données permettra d'autoriser cette modification, le nom de famille n'étant plus utilisé comme clef.

Les données

- Conceptuellement, on peut voir une base de données comme un tableau bi-dimensionnel où chaque ligne correspond à une enregistrement et où chaque colonne correspond à un champ de l'enregistrement.

- Les bases de données disposent souvent d'un indice auto-incrémenté permettant d'identifier les lignes.

- C'est ce qu'on appelle la **clef primaire**.

Les données

- A chaque enregistrement de la table correspondra une valeur unique de la clef primaire.

- Le fait de disposer d'un clef primaire indépendante de nos données autorise à modifier n'importe quel autre champ des enregistrements, dont le nom de famille.

- On adopte une convention standard pour les bases de données en donnant un nom singulier à la table : on l'appellera person.

Interaction avec la base de données

- On va utiliser SQLite comme moteur de base de données pour stocker les données PEOPLE.

- SQLite est la base de données la plus distribuée au monde et elle est disponible en standard sous Python.

- Elle est rapide, travaille sur des fichiers et est adaptée à de nombreux types de projets.

- Il s’agit d’un RDBMS complet (Relational DataBase Management System), qui inclut SQL.

Interaction avec la base de données

- Supposons que la table person existe dans une base de données SQLite.

- On peut alors utiliser SQL pour récupérer les données.

- Contrairement aux langages de programmation de type Python, SQL ne dit pas *comment* récupérer les données mais se contente de spécifier *quelles* données sont requises, en laissant le soin du *comment* au moteur de base de données.

Interaction avec la base de données

- Une requête SQL permettant de récupérer la liste de toutes les personnes présente dans notre base de données, triée par ordre alphabétique du nom de famille, se présente de la façon suivante :

SQL

```SELECT * FROM person ORDER BY 'lname';```
```

Interaction avec la base de données

- On peut soumettre la requête précédente dans une fenêtre terminal, ce qui donne le résultat suivant :

SHELL

```
sqlite> SELECT * FROM person ORDER BY
lname;
2|Brockman|Kent|2018-08-08 21:16:01.888444
3|Easter|Bunny|2018-08-08 21:16:01.889060
1|Farrell|Doug|2018-08-08 21:16:01.886834
```

Interaction avec la base de données

- La sortie précédente correspond à l'ensemble de enregistrements de la base de données, les différents champs étant séparés par le caractère « pipe » |.

- En fait il est possible de soumettre la requête précédente directement à partir de Python. Le résultat serait une liste de tuples. La liste contient l'ensemble des enregistrements, chaque tuple correspondant à un enregistrement.

Interaction avec la base de données

- Toutefois, récupérer les données de cette façon n'est pas vraiment compatible avec l'esprit Python.

- Le fait d'obtenir une liste correspond bien à Python mais le fait d'avoir des tuples est problématique : c'est au programmeur de connaitre l'indice de chaque champ pour accéder aux données.

- On procèderait de la façon suivante sous Python :

Interaction avec la base de données

PYTHON

```python
import sqlite3

conn = sqlite3.connect('people.db')
cur = conn.cursor()
cur.execute('SELECT * FROM person ORDER BY lname')
people = cur.fetchall()
for person in people:
    print(f'{person[2]} {person[1]}')
```

Interaction avec la base de données

- Le programme précédent se déroule de la façon suivante :
- La ligne 1 importe le module sqlite3
- La ligne 3 crée une connexion au fichier de base de données
- La ligne 4 crée un **curseur** de connexion (qui va parcourir la base)
- La ligne 5 utilise le curseur pour exécuter une requête SQL exprimée sous la forme d'une chaîne de caractères

Interaction avec la base de données

- La ligne 6 récupère l'ensemble des enregistrements renvoyés par la requête SQL et les assigne à la variable people

- Les lignes 7 et 8 itèrent sur la liste people et imprime le nom et le prénom de chaque personne

- La variable people a la structure suivante :

Interaction avec la base de données

PYTHON

```
people = [
    (2, 'Brockman', 'Kent', '2018-08-08 21:16:01.888444'),
    (3, 'Easter', 'Bunny', '2018-08-08 21:16:01.889060'),
    (1, 'Farrell', 'Doug', '2018-08-08 21:16:01.886834')
]
```

Interaction avec la base de données

- Le résultat de l'exécution du programme précédent est le suivant :

SHELL

```
Kent Brockman
Bunny Easter
Doug Farrell
```

Interaction avec la base de données

- Dans le programme précédent, on doit savoir que le nom de famille est en position 1 et le prénom en position 2 dans les tuples renvoyés.

- De plus, la structure interne de person doit être connue lorsqu'on passe la variable d'itération person comme paramètre à une fonction ou méthode.

- Ce serait beaucoup plus pratique d'avoir person sous la forme d'un objet Python, les différents champs correspondant à des attributs de l'objet.

- Ceci peut être fait avec SQLAlchemy.

Interaction avec la base de données

- Dans le programme précédent, la requête SQL est une simple chaîne de caractères passée à la base de données pour exécution.

- Cela ne pose pas de problème car la chaîne est entièrement sous le contrôle du programmeur.

- Par contre, notre application Web partira d'un input utilisateur vers l'API REST pour construire une requête SQL.

- Cela peut constituer une faille de sécurité de notre application.

Interaction avec la base de données

- La requête API REST pour accéder à une personne particulière a la forme :

```
GET /api/people/{lname}
```

- Cela signifie que l'API attend une variable, lname, dans le chemin d'URL, qu'elle utilise pour trouver une personne particulière.

Interaction avec la base de données

- Le code Python correspondant prend la forme suivante :

PYTHON

```
lname = 'Farrell'
cur.execute('SELECT * FROM person WHERE
lname = \'{}\''.format(lname))
```

Interaction avec la base de données

- Le code précédent fonctionne de la façon suivante:

- La ligne 1 fixe la valeur de la variable lname à ‘Farrell’. Cela proviendrait du chemin d’URL dans le cadre de l’API.

- Le code SQL généré par la ligne 2 la forme suivante :

SQL

```
SELECT * FROM person WHERE lname = 'Farrell'
```

Interaction avec la base de données

- Lorsque cette requête SQL est exécutée par la base de données, la personne correspondante est recherchée dans la base et les informations associées récupérées.

- C’est le fonctionnement attendu ; toutefois, un utilisateur mal intentionné peut, dans ce cadre de travail, effectuer une injection SQL (SQL injection).

- Par exemple, un utilisateur mal intentionné peut requête l’API de la façon suivante :

Interaction avec la base de données

```
GET /api/people/Farrell');DROP TABLE person;
```

- La requête précédente fixe la valeur de la variable lname à : 'Farrell');DROP TABLE person;', ce qui a pour effet de générer la requête SQL suivante :

SQL

```
SELECT * FROM person WHERE lname = 'Farrell');DROP TABLE person;
```

Interaction avec la base de données

- La requête SQL précédente est valide.
- Elle aurait pour effet de commencer par rechercher dans la table une personne nommée ‘Farrell’ puis, rencontrant ensuite un point-virgule, elle passerait à la requête suivante, qui a pour effet de **supprimer l’intégralité de la table person**.
- Il est clair que cela mettrait notre application hors d’état de fonctionner.

Interaction avec la base de données

- Une façon d'éviter ce genre de situation consiste à filtrer les données fournies par les utilisateurs pour s'assurer qu'elles ne contiennent rien de dangereux pour l'application.

- Il peut être compliqué de le faire et, ce d'autant plus, qu'il faudrait examiner toutes les interactions des utilisateurs avec la base de données.

- Une solution plus pratique consiste à utiliser SQLAlchemy.

Interaction avec la base de données

- SQLAlchemy se chargera de filtrer les requêtes émises par les utilisateurs vers la base de données avant de créer les requêtes SQL correspondantes.

- C’est une autre raison d’utiliser SQLAlchemy lorsqu’on travaille avec des bases de données sur le Web.

Le modèle SQLAlchemy

- SQLAlchemy est un projet important et fournit pléthore d'outils permettant de travailler avec des bases de données sous Python.

- Un des outils qu'elle fournit est un ORM (Object Relational Mapper).

- C'est ce qu'on va utiliser pour créer et travailler avec la base person sous Python.

Le modèle SQLAlchemy

- La Programmation Orientée Objet (P.O.O.) permet de lier les données et leur manipulation, c'est-à-dire les fonctions qui opèrent sur les données.

- En créant des classes SQLAlchemy, on sera à même de relier les champs de la base de données à des manipulations, ce qui permettra d'interagir avec les données.

- La définition en termes de classes SQLAlchemy des données présente dans le table person est la suivante :

Le modèle SQLAlchemy

PYTHON

```
class Person(db.Model):
    __tablename__ = 'person'
    person_id = db.Column(db.Integer,
                          primary_key=True)
    lname = db.Column(db.String)
    fname = db.Column(db.String)
    timestamp = db.Column(db.DateTime,
                          default=datetime.utcnow,
                          onupdate=datetime.utcnow)
```

Le modèle SQLAlchemy

- La classe Person hérite de db.model, qui fournit les attributs et fonctionnalités permettant de manipuler des bases de données.

- Le reste des définitions correspondent à des attributs de classes.

Le modèle SQLAlchemy

- __tablename__ = person fait le lien entre la classe Person et la table person.

- person_id = db.Column(db.Integer, primary_key = True) crée une colonne dans la base de données contenant un entier utilisé comme clef primaire. Cela implique que person_id est une valeur entière auto-incrémentée.

- lname = db.Column(db.String) crée le champ « nom de famille », c’est-à-dire une colonne de la base contenant des chaînes de caractères.

Le modèle SQLAlchemy

- name = db.Column(db.String) crée le champ « prénom », c'est-à-dire une colonne de la base de données contenant des chaînes de caractères.

- timestamp = db.Column(db.DateTime, default = datetime.utcnow, on update = datetime.utcnow) crée un champ « timestamp », c'est-à-dire une colonne dans la base de données contenant des valeurs date/heure. Le paramètre default = datetime.utcnow fixe comme valeur pour défaut pour le « timestamp » la valeur courante utcnow lorsqu'un enregistrement est créé. De même, on update = datetime.utcnow met à jour le « timestamp » avec la valeur utcnow lorsque l'enregistrement est mis à jour.

Le modèle SQLAlchemy

- Pourquoi utiliser utcnow comme « timestamp » ?

- En fait la méthode datetime.utcnow( ) renvoie l'instant UTC courant (UTC : Universel Temps Coordonné). C'est une façon de **standardiser** le « timestamp ».

- Cela permet notamment d'effectuer des calculs plus simples sur les dates/heures.

- Si on accède à l'application depuis des créneaux horaires différents, il suffit de connaître le créneau horaire pour convertir les dates/heures.

- Ce serait plus compliqué si on utilisait un « timestamp » en dates/heures locales.

Le modèle SQLAlchemy

- Que nous apporte la classe Person ?

- L'idée est de pouvoir requêter la table à l'aide de SQLAlchemy et d'obtenir comme résultat une liste d'instances de la classe Person.

- A titre d'exemple, reprenons la requête SQL précédente :

SQL

```
SELECT * FROM people ORDER BY lname;
```

Le modèle SQLAlchemy

- En utilisant SQLAlchemy, la requête prend la forme suivant :

PYTHON

```
from models import Person
 2
 3 people = Person.query.order_by(Person.lname).all()
 4 for person in people:
 5     print(f'{person.fname} {person.lname}')
```

Le modèle SQLAlchemy

- Ignorons la ligne 1 pour l'instant.
- L'instruction SQLAlchemy
  `Person.query.order_by(Person.lname).all( )` fournit une liste d'objets de classe `Person` correspondant à tous les enregistrements de la table `person`, triée par nom de famille.
- La variable `people` contient la liste d'objets.

Le modèle SQLAlchemy

- Ensuite, le programme itère sur la variable people, en affichant successivement le prénom et le nom de chaque personne présente dans la base.

- Notons que le programme **ne fait pas appel à l'indexation** pour accéder aux champs fname et lname ; il utilise les **attributs** des objets Person.

- Utiliser SQLAlchemy autorise à penser en termes d'objets plutôt qu'en termes de requêtes SQL. Plus la base de données est importante, plus c'est pratique et intéressant.

Sérialisation

- Travailler avec le modèle SQLAlchemy est très pratique en termes de programmation.

- C’est surtout pratique quand on écrit des programmes manipulant les données et effectuant des calculs sur celles-ci.

- Toutefois, l’application qu’on a en vue ici est une API REST qui fournit des opérations CRUD sur les données et qui de ce fait n’implique pas grand chose comme manipulation de données.

Sérialisation

- Notre API REST travaille avec des données au format JSON, ce qui peut poser un problème de compatibilité avec le modèle SQLAlchemy.

- Comme les données renvoyées par SQLAlchemy sont des objets Python (des instances de classes Python), le module Connexion ne peut les convertir au format JSON.

- Rappelons que Connexion est le module utilisé pour implémenter notre API, configurée à l'aide d'un fichier YAML.

Sérialisation

- La sérialisation est l'opération consistant à convertir un objet Python en structures de données plus simples, qui peuvent être formatées en utilisant les types de données JSON (JSON datatypes), ces derniers étant listés ci-dessous :

- **string** : chaîne de caractères

- **number** : nombres autorisés par Python (integers, floats, longs)

- **object** : équivalent grossièrement à des **dictionnaires** Python

Sérialisation

- **array** : grossièrement équivalent aux **listes** Python

- **boolean** : prenant sous JSON les valeurs true ou false, mais True ou False sous Python

- **null** : équivalent du None sous Python

Sérialisation

- A titre d'exemple, la classe Person contient un « timestamp », qui est un objet DateTime Python.

- Il n'y a pas d'équivalent sous JSON, donc l'objet DateTime doit être converti en chaîne de caractères pour être manipulé sous JSON.

- La classe Person est en fait suffisamment simple pour qu'on puisse envisager d'en extraire les attributs et de créer manuellement des dictionnaires correspondant aux URL de notre API.

Sérialisation

- Dans des situations plus complexes, avec des modèles SQLAlchemy plus importants, ce n’est plus le cas.

- On a donc recours à module, nommé Marshmallow, qui fera le travail pour nous.

Marshmallow

- Marshmallow permet de créer une classe PersonSchema, qui est le pendant de la classe Person qu'on a créée sous SQLAlchemy.

- Toutefois, au lieu de mettre en correspondance les tables de la base de données et leur champs avec des classes et leurs attributs , la classe PersonSchema définit la façon dont les attributs d'une classe seront converties en un format JSON.

- Voici la définition de classe Marshmallow pour les données de notre table person :

Marshmallow

PYTHON

```python
class PersonSchema(ma.ModelSchema):
    class Meta:
        model = Person
        sqla_session = db.session
```

Marshmallow

- La classe PersonSchema hérite de la classe ma.ModelSchema. Celle-ci est une classe de base Marshmallow et fournit un ensemble d'attributs et de fonctionnalités qui vont permettre de **sérialiser** des objets Python Person au format JSON et de **désérialiser** (l'opération inverse) des données JSON sous la forme d'instances de la classe Person.

Marshmallow

- Le reste de la définition se déroule de la façon suivante :

- **class Meta** : définit une classe nommée Meta au sein de notre classe. La classe ModelSchema dont hérite PersonSchema recherche cette classe interne Meta et l'utilise pour trouver le modèle SQLAlchemy Person et la session de base de données db.session. C'est ce qui permet à Marshmallow de sérialiser/désérialiser les attributs de la classe Person.

Marshmallow

- **model** : spécifie le modèle SQLAlchemy à utiliser pour sérialiser/désérialiser des données.

- **db.session** : spécifie la session de base de données à utiliser pour l'introspection c'est-à-dire l'examen et la détermination des types des attributs.

Créer la base de données

- A ce stade, on a résolu d'utiliser SQLALchemy pour manipuler notre base de données, ce qui permet de se concentrer sur le modèle de données et la façon de les manipuler.

- A présent, on va **créer** notre base de données.

- A cet effet, on va utiliser SQLite.

Créer la base de données

- On utilisera SQLite pour deux raisons :

1. Elle est fournie par défaut avec Python et n’a donc pas à être installée comme un module séparé.

2. Elle sauvegarde toute l’information dans un seul fichier et, de ce fait, elle est simple à configurer et à utiliser.

Créer la base de données

- Il serait possible d'utiliser un serveur de bases de données séparé, du type MySQL ou PostgreSQL, mais cela impliquerait d'installer ces systèmes et de les mettre en route, ce qui est bien au delà du propos de ce cours et plus du ressort des informaticiens.

- Comme la base de données est manipulée à l'aide de SQLAlchemy, les détails de son implémentation ne sont en fait pas importants.

Créer la base de données

- Nous allons créer un programme build_database.py afin de créer et d'initialiser la base de données SQLite people.db qui contiendra nos données.

- Ce faisant, on va créer deux modules additionnels config.py et models.py qui seront utilisés par build_database.py et server.py.

Créer la base de données

- **config.py** : se chargera d'importer et de configurer les différents modules requis. Cela inclura Flask, Connexion, SQLAlchemy et Marshmallow. Comme il sera utilisé à la fois par build_database.py et server.py, une partie de la configuration ne s'appliquera qu'à server.py.

- **models.py** : c'est le module qui se chargera de créer la classe SQLAlchemy Person et la classe Marshmallow PersonSchema. Ce module dépendra du module config.py via certains des objets créés et configurés dans ce dernier.

Le module Config

- Le module config.py regroupe toute l'information de configuration.

- Voici le code correspondant :

Le module Config

PYTHON (config.py)

```
import os
import connexion
from flask_sqlalchemy import SQLAlchemy
from flask_marshmallow import Marshmallow

basedir = os.path.abspath(os.path.dirname(__file__))

# Create the Connexion application instance
connex_app = connexion.App(__name__,
specification_dir=basedir)
```

Le module Config

```
# Get the underlying Flask app instance
app = connex_app.app

# Configure the SQLAlchemy part of the app instance
app.config['SQLALCHEMY_ECHO'] = True
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:////' +
os.path.join(basedir, 'people.db')
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False

# Create the SQLAlchemy db instance
db = SQLAlchemy(app)

# Initialize Marshmallow
ma = Marshmallow(app)
```

Le module Config

- Les lignes 2-4 importent Connexion, SQLAlchemy et Marshmallow.

- La ligne 6 crée la variable basedir qui pointe vers le répertoire dans lequel le programme est exécuté.

- La ligne 9 utilise la variable basedir pour créer l'instance de l'application Connexion et lui fournit le chemin vers le fichier de configuration swagger.yml.

- La ligne 12 crée une variable app qui correspond à l'instance Flask initialisée par Connexion.

Le module Config

- La ligne 15 utilise la variable app pour configurer SQLAlchemy. On commence par fixer SQLAlchemy_ECHO à True, ce qui a pour effet de renvoyer les requêtes SQL exécutées par SQLAlchemy à la console. C'est très utile en phase de mise au point et de débogage. En environnement de production, on la fixe à False.

- La ligne 16 fixe SQLALCHEMY_DATABASE_URI à sqlite:////' + os.path.join(basedir, 'people.db'). Cela indique à SQLALchemy d'utiliser SQLite comme base de données et un fichier nommé people.db dans le répertoire courant comme fichier de données. D'autres moteurs de bases de données, comme MySQL ou PostgreSQL, seraient configurés avec une SQLALCHEMY_DATABASE_URI différente.

Le module Config

- La ligne 17 fixe SQLALCHEMY_TRACK_MODIFICATIONS à False, ce qui a pour effet de désactiver le système de gestion d'événements SQLAlchemy, qui est actif par défaut. Ce système est utile pour les programmes basés sur des évènements (clicks, …) mais ralentit l'exécution. Notre programme n'étant pas basé sur des événements, on le désactive.

- La ligne 19 crée la variable db en faisant appel à SQLAlchemy(app). Cela initialise SQLAlchemy avec l'information de configuration qu'on vient de spécifier. La variable db sera importée dans le programme build_database.py pour lui donner accès à SQLAlchemy et à la base de données. Il en ira de même dans les modules server.py et people.py.

Le module Config

- La ligne 23 crée la variable ma en faisant appel à Marshmallow(app). Cela initialise Marshmallow et autorise **l'introspection** sur les composantes SQLAlchemy attachées à la base de données. Marshmallow doit de ce fait être initialisé **après** SQLAlchemy.

Le module Models

- Le module models.py est utilisé pour définir les classes SQLAlchemy Person et Marshmallow PersonSchema, comme vu précédemment.

- Voici le code correspondant :

Le module Models

PYTHON (models.py)

```python
from datetime import datetime
from config import db, ma

class Person(db.Model):
      __tablename__ = 'person'
      person_id = db.Column(db.Integer, primary_key=True)
      lname = db.Column(db.String(32), index=True)
      fname = db.Column(db.String(32))
      timestamp = db.Column(db.DateTime, default=datetime.utcnow, onupdate=datetime.utcnow)

 class PersonSchema(ma.ModelSchema):
     class Meta:
         model = Person
         sqla_session = db.session
```

Le module Models

- La ligne 1 importe la classe datetime du module datetime disponible par défaut sous Python

- La ligne 2 importe les objets db et ma du module config.py. Cela donne au programme l'accès aux attributs SQLAlchemy et aux méthodes attachés à l'objet db et aux attributs Marshmallow et aux méthodes attachés à l'objet ma.

- Les lignes 4 à 9 définissent la classe Person de façon à ce qu'elle bénéficie des fonctionnalités SQLAlchemy, telles que la connexion à une base de données et l'accès à ses tables.

Le module Models

- Les lignes 11 à 14 définissent la classe PersonSchema de façon à ce qu'elle bénéficie des fonctionnalités Marshmallow comme **l'introspection** sur la classe Person, qui permet de sérialiser/désérialiser les instances de cette classe.

Créer la base de données

- On a vu comment les tables d'une base de données pouvaient être mise en correspondance avec des classes SQLAlchemy.

- A présent, on va créer la base de données et lui affecter des données.

- A cet effet, on va utiliser le programme build_database.py suivant :

Créer la base de données

PYTHON (build_database.py)

```
import os
from config import db
from models import Person

# Data to initialize database with
PEOPLE = [
      {'fname': 'Doug', 'lname': 'Farrell'},
      {'fname': 'Kent', 'lname': 'Brockman'},
      {'fname': 'Bunny','lname': 'Easter'}
 ]
```

Créer la base de données

```
# Delete database file if it exists currently
if os.path.exists('people.db'):
    os.remove('people.db')

# Create the database
db.create_all()

# Iterate over the PEOPLE structure and populate the database
for person in PEOPLE:
    p = Person(lname=person['lname'], fname=person['fname'])
    db.session.add(p)

db.session.commit()
```

Créer la base de données

- La ligne 2 importe l'objet db du module config.py.
- La ligne 3 importe la classe Person du module models.py.
- Les lignes 6 à 10 créent la structure PEOPLE, qui est une liste de dictionnaire contenant les données. **Notons qu'une telle structure peut facilement être créée à partir d'un fichier CSV par exemple.**
- Les lignes 13 et 14 font un peu de ménage en effaçant le fichier people.db s'il existe déjà. Cela permet de s'assurer qu'on repart de zéro si on doit recréer la base de données.

Créer la base de données

- La ligne 17 crée la base de données à l'aide de l'appel db.create_all( ). Cela crée la base de données en utilisant l'instance db importée du module config. L'instance db est notre connexion à la base de données.

- Les lignes 20 à 22 itèrent sur la liste PEOPLE et utilisent les dictionnaires qu'elle contient pour instancier la classe Person. Après qu'elle soit instanciée, on appelle la fonction db.session.add(p). Cela utilise l'instance de connexion db pour accéder à l'objet session. La session est ce qui gère les **actions** effectuées sur la base de données, qui y sont enregistrées. Ici, on exécute la méthode add(p) pour ajouter une nouvelle instance de Person à l'objet session.

Créer la base de données

- La ligne 24 appelle db.session.commit( ) pour sauvegarder l'ensemble des objets Person créés dans la base de données.

- ***A la ligne 22, rien n'est encore sauvegardé dans la base de données*** ; tout est sauvegardé dans l'objet session. ***C'est seulement suite à l'exécution de*** db.session.commit( ) ***que la session interagit avec la base de données et lui applique les actions spécifiées***.

Créer la base de données

- Sous SQLAlchemy, la **session** est un objet important. Celle-ci relie la base de données et les objets SQLAlchemy manipulés dans un programme Python.

- Elle permet de maintenir la cohérence entre les données du programme et les données de la base de données.

- Toutes les actions effectuées y sont sauvegardées et elle met à jour la base de données en fonction des actions explicites ou implicites engagées dans le programme.

Créer la base de données

- On est à présent en mesure de créer et d'initialiser la base de données.

- Il suffit d'exécuter le programme build_database.py, par exemple sous Spyder.

- Lorsque le programme est exécuté, il affiche les messages de SQLAlchemy à la console. C'est dû au fait qu'on a fixé SQLALCHEMY_ECHO à True dans le module config.py.

Créer la base de données

CONSOLE PYTHON

2019-03-09 00:54:02,192 INFO sqlalchemy.engine.base.Engine SELECT CAST('test plain returns' AS VARCHAR(60)) AS anon_1

2019-03-09 00:54:02,192 INFO sqlalchemy.engine.base.Engine ()

2019-03-09 00:54:02,193 INFO sqlalchemy.engine.base.Engine SELECT CAST('test unicode returns' AS VARCHAR(60)) AS anon_1

2019-03-09 00:54:02,194 INFO sqlalchemy.engine.base.Engine ()

2019-03-09 00:54:02,195 INFO sqlalchemy.engine.base.Engine PRAGMA table_info("person")

2019-03-09 00:54:02,196 INFO sqlalchemy.engine.base.Engine ()

2019-03-09 00:54:02,199 INFO sqlalchemy.engine.base.Engine

```
CREATE TABLE person (
        person_id INTEGER NOT NULL,
        lname VARCHAR(32),
        fname VARCHAR(32),
        timestamp DATETIME,
        PRIMARY KEY (person_id)
)
```

Créer la base de données

2019-03-09 00:54:02,199 INFO sqlalchemy.engine.base.Engine ()

2019-03-09 00:54:02,200 INFO sqlalchemy.engine.base.Engine COMMIT

2019-03-09 00:54:02,202 INFO sqlalchemy.engine.base.Engine BEGIN (implicit)

2019-03-09 00:54:02,203 INFO sqlalchemy.engine.base.Engine INSERT INTO person (lname, fname, timestamp) VALUES (?, ?, ?)

2019-03-09 00:54:02,204 INFO sqlalchemy.engine.base.Engine ('Farrell', 'Doug', '2019-03-08 23:54:02.203626')

2019-03-09 00:54:02,205 INFO sqlalchemy.engine.base.Engine INSERT INTO person (lname, fname, timestamp) VALUES (?, ?, ?)

2019-03-09 00:54:02,205 INFO sqlalchemy.engine.base.Engine ('Brockman', 'Kent', '2019-03-08 23:54:02.205423')

2019-03-09 00:54:02,206 INFO sqlalchemy.engine.base.Engine INSERT INTO person (lname, fname, timestamp) VALUES (?, ?, ?)

2019-03-09 00:54:02,207 INFO sqlalchemy.engine.base.Engine ('Easter', 'Bunny', '2019-03-08 23:54:02.206600')

2019-03-09 00:54:02,207 INFO sqlalchemy.engine.base.Engine COMMIT

Créer la base de données

- L’essentiel de ce qui est affiché correspond aux requêtes SQL générées par SQLAlchemy pour créer et renseigner le fichier de base de données people.db.

Mise à jour de l’API

- Les changements qu’on a effectué par rapport à la première version de notre API ne sont pas radicaux.

- On modifie simplement les définitions d’URL pour prendre ne compte le fait que la clef primaires est person_id :

Mise à jour de l’API

- Action : Create, Verbe HTTP : POST, URL : /api/people, Description : définit une URL pour créer une personne.

- Action : Read, Verbe HTTP : GET, URL : /api/people, Description : définit une URL pour lire une collection de personnes.

- Action : Read, Verbe HTTP : GET, URL : /api/people/{person_id}, Description : Définit une URL pour lire une personne particulière identifiée par son person_id.

- Action : Update, Verbe HTTP : PUT, URL : /api/people/{person_id}, Description : Définit une URL pour mettre à à jour l’information d’une personne identifiée par son person_id.

Mise à jour de l’API

- Action : Delete, Verbe HTTP : DELETE, URL : /api/people/{person_id}, Description : Définit un URL pour supprimer une personne identifié par son person_id.

- Alors que la première définition de l’API faisait appel à lname pour identifier une personne, celle-ci fait appel à person_id.

- Cela permet d’autoriser l’utilisateur à modifier le nom de famille lors de la mise à jour de l’information d’une personne.

Mise à jour de l'API

- Afin d'implémenter ces changement, il faut éditer le fichier swagger.yml configurant l'API.

- Le fichier modifié est accessible sur le forum.

- Pour l'essentiel, on remplace partout lname par person_id et on rajoute person_id aux réponses POST et PUT.

Mise à jour des gestionnaires

- Après avoir mis à jour le fichier swagger.yml de façon à prendre en compte l'identifiant person_id, il faut mettre à jour les fonctions correspondantes (appelées **Gestionnaires**, ang. **Handlers**) du fichier people.py pour prendre en compte ces changements.

- De façon analogue à ce qu'on a fait pour swagger.yml, cela consiste essentiellement à remplacer lname par person_id.

- Voici in extrait du nouveau module people.py (disponible sur le forum) montrant le gestionnaire (handler) associé à la requête GET /api/people :

Mise à jour des gestionnaires

PYTHON

```
from flask import (
      make_response,
      abort,
  )
  from config import db
  from models import (
      Person,
      PersonSchema,
  )

def read_all():
    """
    This function responds to a request for /api/people
    with the complete lists of people

    :return:        json string of list of people
    """
    # Create the list of people from our data
    people = Person.query \
        .order_by(Person.lname) \
        .all()

    # Serialize the data for the response
    person_schema = PersonSchema(many=True)
    return person_schema.dump(people).data
```

Mise à jour des gestionnaires

- Les lignes 1 à 9 importent des modules Flask pour créer les réponses de l'API, ainsi que l'instance db du module config.py. De plus, elles importent les classes SQLAlchemy Person et Marshmallow PersonSchema qui permettent d'accéder à la base de données et de sérialiser les résultats.

- La ligne 11 correspond au début de la définition du gestionnaire read_all( ) qui se charge de la requête GET / api/people et renvoie tous les enregistrements de la table person triés par ordre alphabétique du nom de famille.

Mise à jour des gestionnaires

- Les lignes 19 à 22 demandent à SQLAlchemy de requêter la table person pour obtenir l'ensemble des enregistrements, de les trier par ordre croissant du nom de famille et de stocker la liste correspondante d'objets Python Person dans la variable people.

- La ligne 24 crée une instance de la classe Marshmallow PersonSchema en lui passant le paramètre many = True. Ceci autorise l'instance à manipuler un itérable (ici une liste) à sérialiser.

Mise à jour des gestionnaires

- La ligne 25 utilise l'instance person_schema en appelant sa méthode dump( ) avec comme argument la liste people. Le résultat est un objet possédant un attribut data contenant la liste people, qui peut être converti en JSON.

- La conversion est effectuée par Connexion en réponse à la requête de l'API.

- Notons qu'on a du passer par Marshmallow, car la liste people ne peut être convertie directement en JSON par Connexion, qui ne sait pas convertir le champ « timestamp » (objet datetime).

Mise à jour des gestionnaires

- Voici un autre extrait du module people.py, qui contient le gestionnaire pour la requête de lecture des informations d’une personne particulière dans la table person : GET /api/people/{person_id}.

- La fonction read_one(person_id) ci-après reçoit un person_id de l’URL de requête, indiquant que l’utilisateur cherche une personne en particulier.

Mise à jour des gestionnaires

PYTHON

```python
def read_one(person_id):
    """
    This function responds to a request for /api/people/{person_id}
    with one matching person from people

    :param person_id:   ID of person to find
    :return:            person matching ID
    """
    # Get the person requested
    person = Person.query \
        .filter(Person.person_id == person_id) \
        .one_or_none()

    # Did we find a person?
    if person is not None:

        # Serialize the data for the response
        person_schema = PersonSchema()
        return person_schema.dump(person).data

    # Otherwise, nope, didn't find that person
    else:
        abort(404, 'Person not found for Id: {person_id}'.format(person_id=person_id))
```

Mise à jour des gestionnaires

- Les lignes 10 à 12 utilisent le paramètre person_id dans une requête SQLAlchemy en utilisant la méthode filter de l'objet requête pour rechercher une personne ayant le person_id spécifié. Plutôt que d'utiliser la méthode de requête all( ), on utilise la méthode one_or_none( ) pour obtenir les informations d'une personne ou renvoyer None si aucune personne n'a été trouvée.

- La ligne 15 détermine si une personne a été trouvée ou pas.

Mise à jour des gestionnaires

- La ligne 17 illustre le fait que si person n'est pas None (i.e. si une personne a été trouvée) alors la sérialisation des données est un peu différente. On ne spécifie pas many = True lors de la création de l'instance PersonSchema. On laisse many = False (valeur par défaut) car il y a **un seul objet** à sérialiser.

- La ligne 18 fait appel à la méthode dump de person_schema et l'attribut data de l'objet résultant est renvoyé.

- La ligne 23 utilise la méthode Flask abort( ) pour renvoyer un message d'erreur si person vaut None (aucune personne n'a été trouvée).

Mise à jour des gestionnaires

- Le module people.py doit également être modifié pour ce qui est de la création de nouvelle personnes dans la base de données.

- On utilise la classe Marshmallow PersonSchema pour **désérialiser** une structure JSON envoyée avec la requête HTTP afin d'obtenir un objet SQLAlchemy Person.

Mise à jour des gestionnaires

PYTHON

```python
def create(person):
    """
    This function creates a new person in the people structure
     based on the passed-in person data

    :param person:  person to create in people structure
    :return:        201 on success, 409 on person exists
    """
    fname = person.get('fname')
    lname = person.get('lname')

    existing_person = Person.query \
        .filter(Person.fname == fname) \
        .filter(Person.lname == lname) \
        .one_or_none()

    # Can we insert this person?
    if existing_person is None:

        # Create a person instance using the schema and the passed-in person
        schema = PersonSchema()
        new_person = schema.load(person, session=db.session).data

        # Add the person to the database
        db.session.add(new_person)
        db.session.commit()

        # Serialize and return the newly created person in the response
        return schema.dump(new_person).data, 201

    # Otherwise, nope, person exists already
    else:
        abort(409, f'Person {fname} {lname} exists already')
```

Mise à jour des gestionnaires

- Les lignes 9 et 10 définissent les variables fname et lname à partir de la structure Person passée dans le corps de la requête HTTP POST.

- Les lignes 12 à 15 utilisent la classe SQLAlchemy Person pour requêter la base sur l'existence d'une personne de nom et prénom de même valeur que lname et fname.

- La ligne 18 vérifie si existing_person vaut None (la personne n'a pas été trouvée).

Mise à jour des gestionnaires

- La ligne 21 crée une instance de PersonSchema nommée schema.

- La ligne 22 utilise la variable schema pour charger les données contenues dans les variables paramètres de person et créer une nouvelle instance de la classe SQLAlchemy Person appelée new_person.

- La ligne 25 ajoute l'instance new_person à db.session.

- La ligne 26 enregistre l'instance new_person dans la base de données, ce qui a pour effet de lui assigner un identifiant (compteur entier auto-incrémenté) et un « timestamp ».

Mise à jour des gestionnaires

- La ligne 33 utilise la méthode Flask abort( ) pour renvoyer un message d’erreur si existing_person ne vaut pas None, c’est-à-dire si la personne existe déjà dans la base de données.

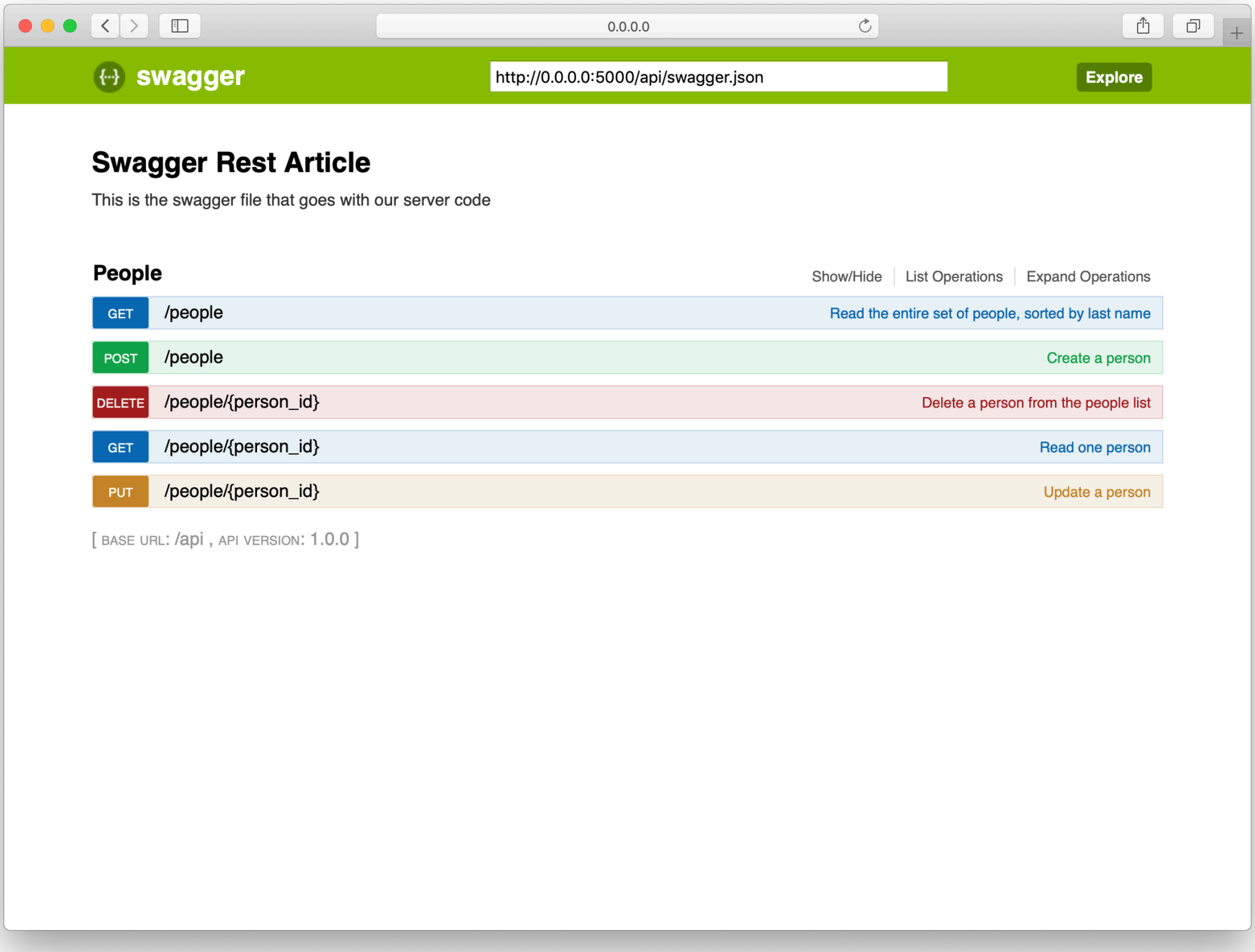
Mise à jour de l’UI Swagger

- Les modifications précédentes ayant été effectuées, notre nouvelle API est fonctionnelle.

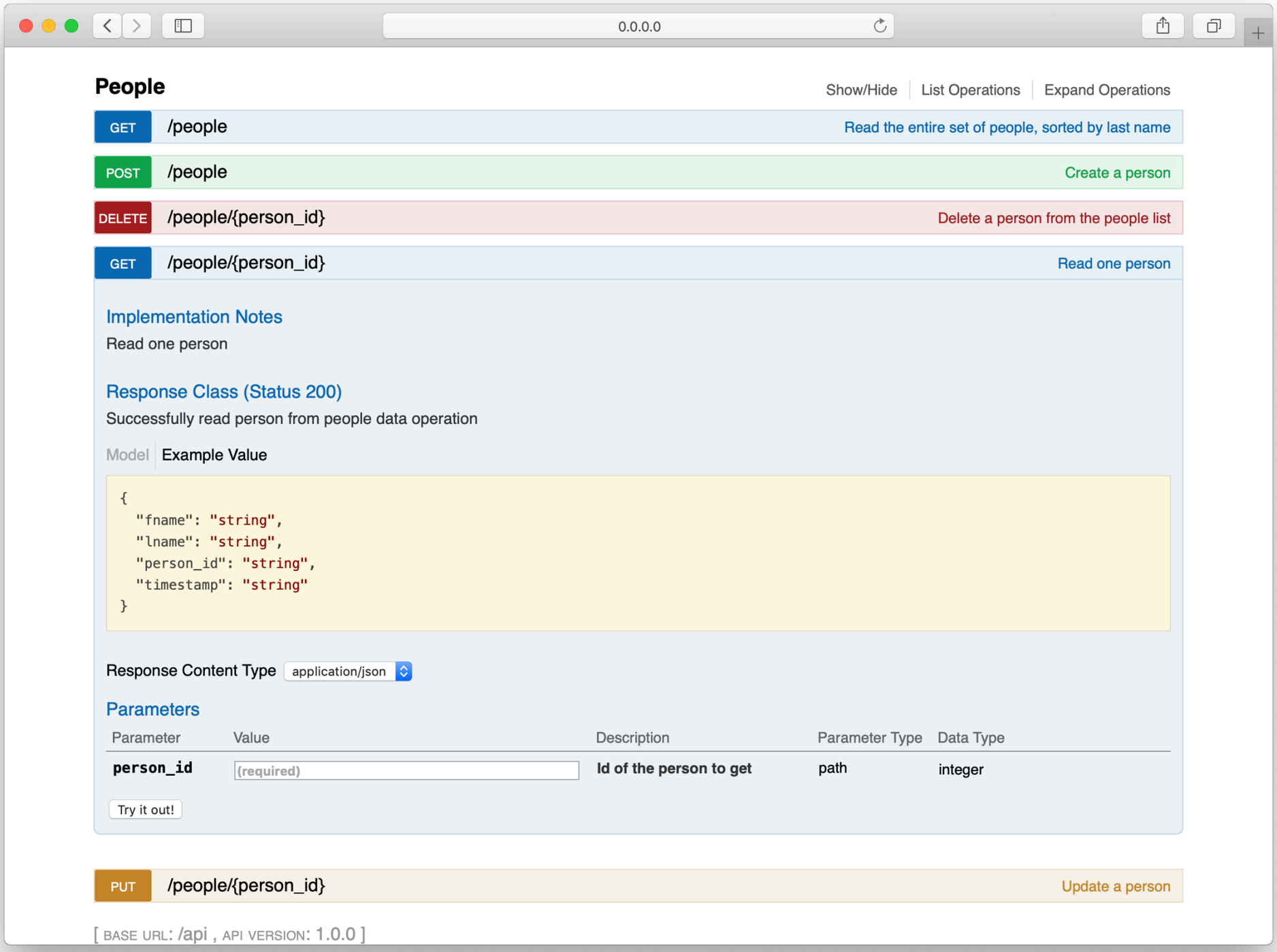
- Ces modifications sont reflétées par l’interface Swagger associée à l’API.

- Voici à quoi ressemble à présent cette dernière :

Mise à jour de l'UI Swagger



Mise à jour de l'UI Swagger



Mise à jour de l’UI Swagger

- Le paramètre de chemin lname a été remplacé par person_id qui est la clef primaire de la base de données.

- Les modifications apportées à l’UI sont le résultats combinés des modifications apportées au fichier swagger.yml et des modifications correspondantes du code.

Mise à jour de l’application Web

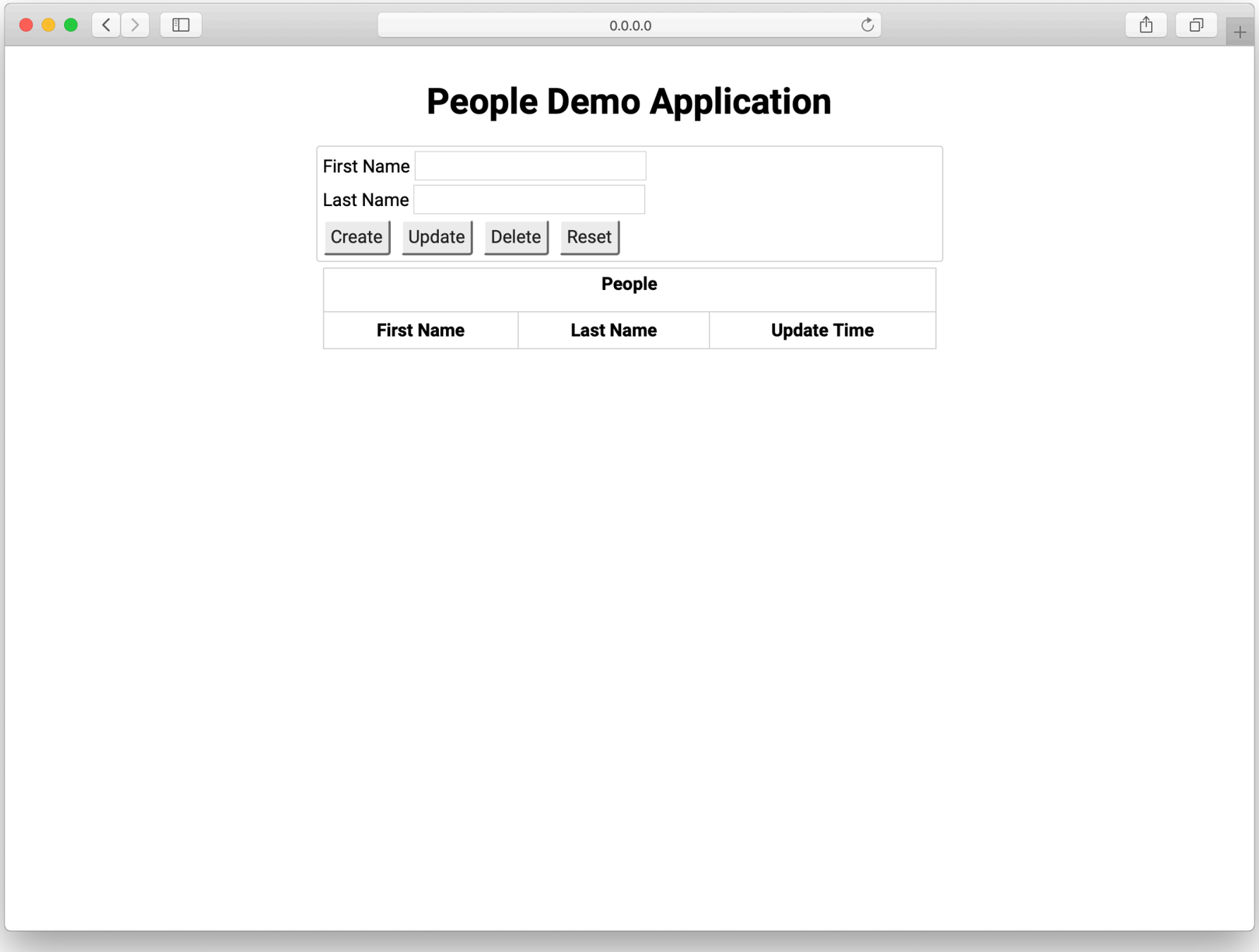
- Notre API REST étant fonctionnelle et le résultat des opérations CRUD étant sauvegardé de façon permanente dans la base de données, il reste à modifier le fichier JavaScript pour mettre à jour notre application Web.

- Ces modifications tiennent essentiellement à remplacer lname par person_id comme clef primaire de notre base de données.

- Le fichier JavaScript final est disponible sur le forum.

- Voici l’écran d’accueil de notre application Web :

Mise à jour de l’application Web



Python et technologies Web

4. Déployer un modèle de Machine Learning à l'aide d'une API REST

Salim Lardjane
Université de Bretagne Sud

Les données

- On va utiliser des données de qualité de vin, récupérées sur l'UCI Machine Learning Repository (https://archive.ics.uci.edu/ml/datasets/Wine+Quality), un site regroupant un grand nombre de jeux de données utilisés pour tester les algorithmes de Machine Learning.

- On souhaite prévoir la qualité du vin (un score entre 0 et 10) à partir de variables mesurant ses caractéristiques physico-chimiques (acidité, densité, etc).

Les données

- On commence par récupérer les données sous Pandas.

- C’est fait en utilisant le code suivant :

Les données

PYTHON

```python
import pandas as pd

#loading our data as a panda
df = pd.read_csv('winequality-red.csv', delimiter=";")


#getting only the column called quality
label = df['quality']


#getting every column except for quality
features = df.drop('quality', axis=1)
```

Le modèle

- Afin de prévoir la qualité du vin, on utilise un simple modèle de régression multiple.

- Pour tester le modèle, on effectue une prévision sur des données artificielles.

- C’est fait à l’aide du code suivant :

Le modèle

PYTHON

```
import pandas as pd

import numpy as np

from sklearn import linear_model


#loading and separating our wine dataset into labels and features

df = pd.read_csv('winequality-red.csv', delimiter=";")

label = df['quality']

features = df.drop('quality', axis=1)
```

Le modèle

```
#defining our linear regression estimator and training it with our wine data

regr = linear_model.LinearRegression()

regr.fit(features, label)


#using our trained model to predict a fake wine

#each number represents a feature like pH, acidity, etc.

print regr.predict([[7.4,0.66,0,1.8,0.075,13,40,0.9978,3.51,0.56,9.4]]).tolist()
```

Sérialisation

- La librairie Pickle est utilisée pour **sérialiser** le modèle dans un fichier, ce qui permet de le charger dans de nouveaux programmes.

- Cela permet de découpler le code lié à l'apprentissage du modèle du code lié à son déploiement.

- L'exportation et importation du modèle est faite à l'aide du code suivant :

Sérialisation

PYTHON

```python
import pickle


#creating and training a model

regr = linear_model.LinearRegression()
regr.fit(features, label)



#serializing our model to a file called model.pkl

pickle.dump(regr, open("model.pkl","wb"))



#loading a model from a file called model.pkl

model = pickle.load(open("model.pkl","r"))
```

Flask

- Afin de déployer le modèle on va utiliser Flask en première approche, afin de lancer un serveur Web possédant une seule route.

- La librairie Pickle permettra de charger le modèle de régression précédent pour utilisation avec le serveur.

Ajouter le modèle au serveur

- Le serveur charge le modèle lors de la phase d’initialisation.

- On peut y accéder en envoyant une requête POST sur la route /predict. La route récupère un ensemble de caractéristiques physico-chimiques à partir du corps de la requête et les dirige vers le modèle. La prédiction du modèle est alors renvoyée à l’utilisateur.

- C’est fait à l’aide du code suivant :

Ajouter le modèle au serveur

PYTHON

```
import pickle
import flask
app = flask.Flask(__name__)

#getting our trained model from a file we created earlier
model = pickle.load(open("model.pkl","r"))
```

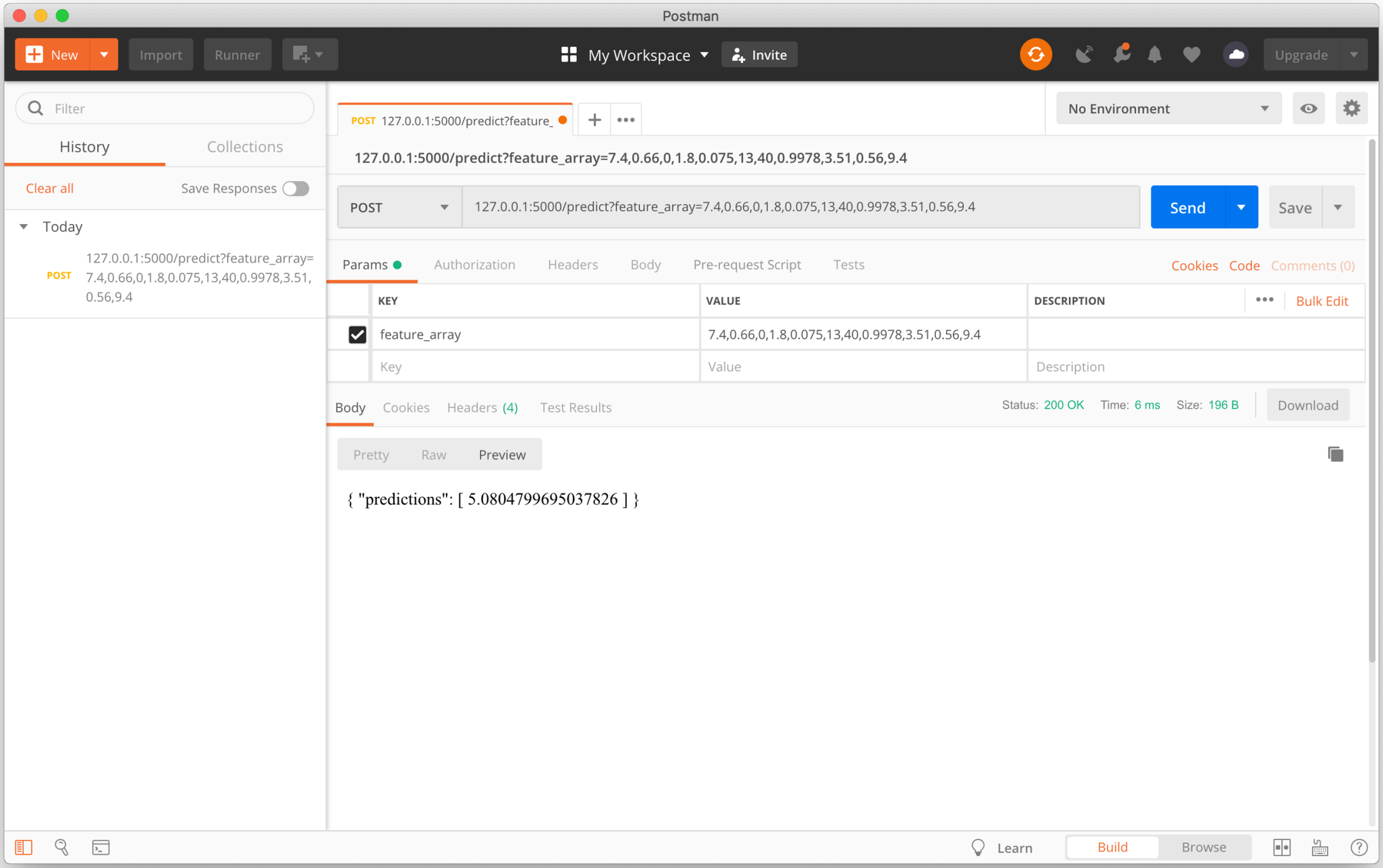
Ajouter le modèle au serveur

```
#defining a route for only post requests
@app.route('/predict', methods=['POST'])
def predict():

        #getting an array of features from the post request's body
        query_parameters = request.args
        feature_array = np.fromstring(query_parameters['feature_array'],dtype=float,sep=",")

        #creating a response object
        #storing the model's prediction in the object
        response = {}
        response['predictions'] = model.predict([feature_array]).tolist()

        #returning the response object as json
        return flask.jsonify(response)

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

Test

- Afin de tester le déploiement de notre modèle, on doit passer une requête HTTP POST à notre serveur.
- Ceci ne peut être fait via la barre d'adresse du navigateur, qui ne transmet que des requêtes GET.
- On va utiliser pour cela le logiciel Postman (https://www.getpostman.com).
- Le résultat est le suivant :

Test


Postman
New
Import
Runner
My Workspace
Invite
Upgrade
Filter
History
Collections
Clear all
Save Responses
Today
POST
127.0.0.1:5000/predict?feature_array=7.4,0.66,0,1.8,0.075,13,40,0.9978,3.51,0.56,9.4
POST 127.0.0.1:5000/predict?feature_
No Environment
127.0.0.1:5000/predict?feature_array=7.4,0.66,0,1.8,0.075,13,40,0.9978,3.51,0.56,9.4
POST
127.0.0.1:5000/predict?feature_array=7.4,0.66,0,1.8,0.075,13,40,0.9978,3.51,0.56,9.4
Send
Save
Params
Authorization
Headers
Body
Pre-request Script
Tests
Cookies
Code
Comments (0)
KEY
VALUE
DESCRIPTION
Bulk Edit
feature_array
7.4,0.66,0,1.8,0.075,13,40,0.9978,3.51,0.56,9.4
Key
Value
Description
Body
Cookies
Headers (4)
Test Results
Status: 200 OK
Time: 6 ms
Size: 196 B
Download
Pretty
Raw
Preview
{ "predictions": [5.0804799695037826] }
Learn
Build
Browse

Test

- L’ensemble du code est disponible sur le forum, dans le répertoire wine-quality.

- On va a présent aborder un exemple plus conséquent.

Données

- Les données proviennent d'une compétition Kaggle (https://www.kaggle.com/c/sentiment-analysis-on-movie-reviews/data).

- Elles portent sur les sentiments de spectateurs a propos de films qu'ils ont visionnés.

- Le fichier train.tsv contient des avis et le label de sentiment associé.

- Le fichier test.tsv contient uniquement des avis.

Données

- Les labels de sentiment sont les suivants :

0 : negative

1 : somewhat negative

2 : neutral

3 : somewhat positive

4 : positive

Données

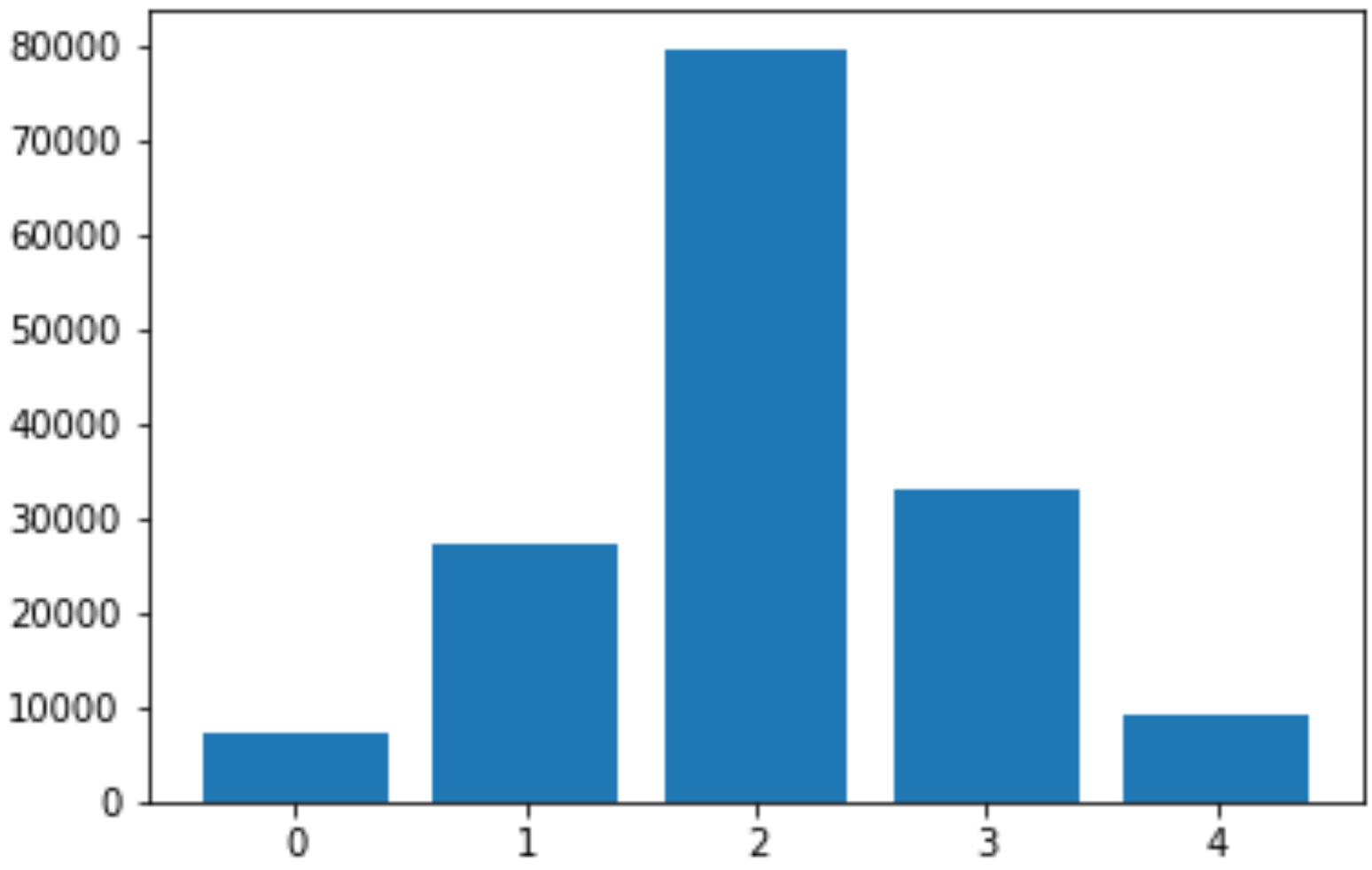
- Les avis sont découpés en phrases et les phrases sont découpées en groupes de mots. Les phrases et leur constituants ont chacun un identifiant (sentenceId, phraseId).

- Toutes les phrases ont un label de sentiment associé. On peut donc entraîner un modèle via lequel les groupes de mots induiront un sentiment négatif, neutre ou positif pour une phrase.

Données

	PhraseId	SentenceId	Phrase	Sentiment
0	1	1	A series of escapades demonstrating the adage ...	1
1	2	1	A series of escapades demonstrating the adage ...	2
2	3	1	A series	2
3	4	1	A	2
4	5	1	series	2
5	6	1	of escapades demonstrating the adage that what...	2
6	7	1	of	2
7	8	1	escapades demonstrating the adage that what is...	2
8	9	1	escapades	2
9	10	1	demonstrating the adage that what is good for ...	2

Distribution des scores



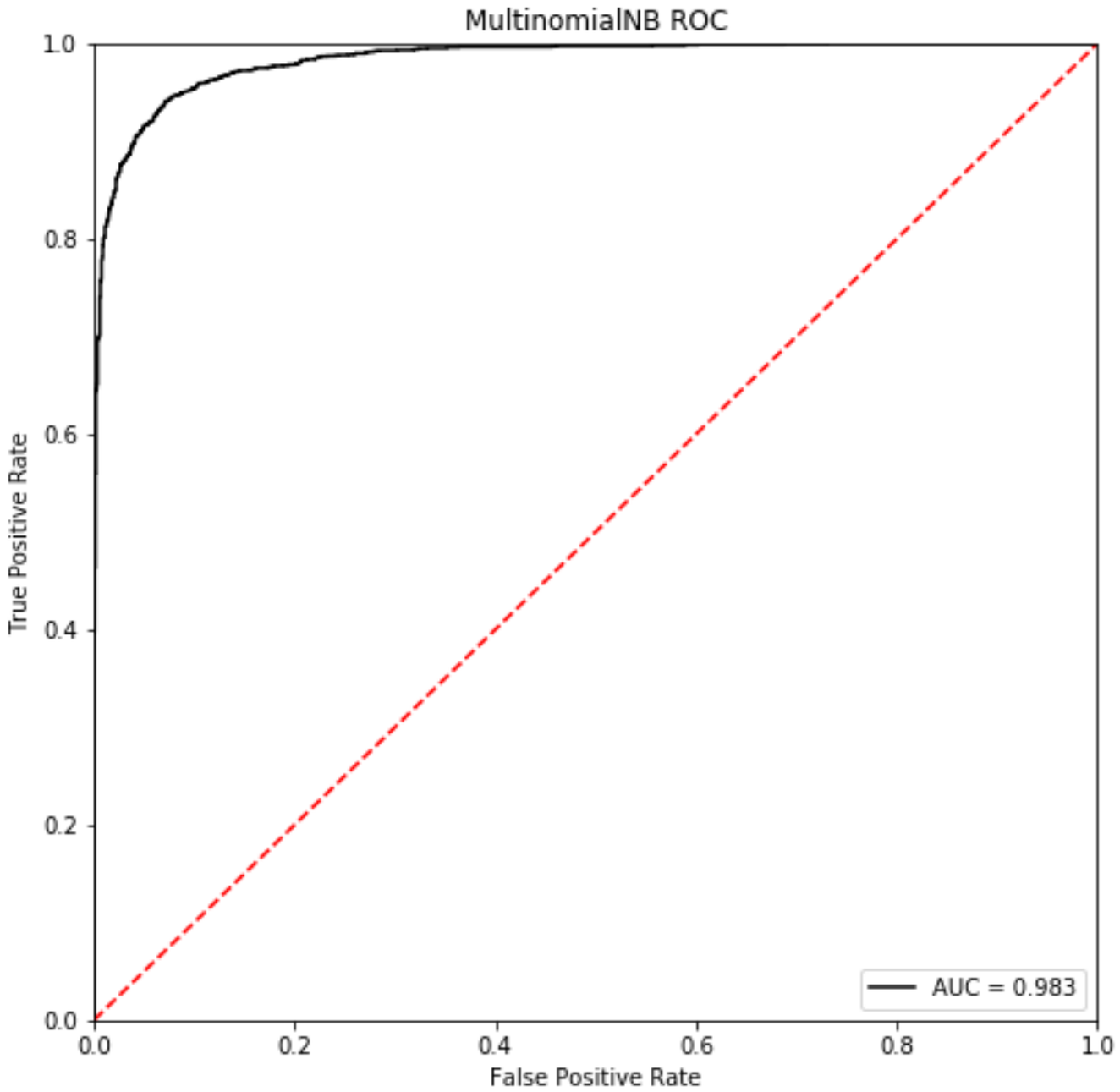
Modèle

- La majorité des groupes de mots a un score neutre. Cela pose problème si on essaye d'utiliser sur ces données un classificateur bayésien naïf.

- On se limite donc aux scores extrêmes (négatif, positif) et on entraîne un classificateur bayésien naïf à les prédire.

- Le code correspondant est disponible sur le forum. On se contente ici de donner la courbe ROC associée.

Modèle


MultinomialNB ROC
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
AUC = 0.983

Modèle

- On voit que le modèle multinômial bayésien naïf est très efficace pour prévoir le sentiment à partir des phrases.

- On ne rentrera pas dans les détails du modèle, qui sont assez complexes.

- On le sauvegarde à l'aide de pickle, comme on l'a vu précédemment.

- On va à présent s'intéresser à la façon de le déployer à l'aide d'une API REST.

API REST

- On commence par importer les librairies qu'on va utiliser, dont pickle, qui permettra de récupérer le modèle.

- Le code correspondant est le suivant :

API REST

PYTHON

```
from flask import Flask
from flask_restful import reqparse, abort, Api, Resource
import pickle
import numpy as np

from model import NLPModel
app = Flask(__name__)
api = Api(app)

# create new model object
model = NLPModel()

# load trained classifier
clf_path = 'lib/models/SentimentClassifier.pkl'
with open(clf_path, 'rb') as f:
    model.clf = pickle.load(f)

# load trained vectorizer
vec_path = 'lib/models/TFIDFVectorizer.pkl'
with open(vec_path, 'rb') as f:
    model.vectorizer = pickle.load(f)
```

API REST

- On va utiliser à présent un « parser » (analyseur) qui va analyser les requêtes reçues par l'API.

- Les arguments de requête seront mis dans un dictionnaire ou un objet JSON.

- Dans notre exemple, on va chercher la clef « query » (requête).

- La requête sera une phrase à laquelle l'utilisateur souhaite associer un sentiment (positif ou négatif).

- Le code correspondant est le suivant :

API REST

PYTHON

```
# argument parsing
parser = reqparse.RequestParser()
parser.add_argument('query')
```

API REST

- Dans la suite, on va utiliser une ressource Flask-RESTful.

- Falsk-RESTful est une librairie Python dédiée à la construction d'API REST.

- Les ressources sont des outils permettant de mettre au point le routage de façon plus simple.

- Elles offrent un accès au diverses requêtes HTTP. Il suffit pour cela de définir des méthodes associées aux ressources.

- Sur notre exemple, le code correspondant est le suivant :

API REST

PYTHON

```
class PredictSentiment(Resource):
    def get(self):
        # use parser and find the user's query
        args = parser.parse_args()
        user_query = args['query']
        # vectorize the user's query and make a prediction
        uq_vectorized = model.vectorizer_transform(
            np.array([user_query]))
        prediction = model.predict(uq_vectorized)
        pred_proba = model.predict_proba(uq_vectorized)
        # Output 'Negative' or 'Positive' along with the score
        if prediction == 0:
            pred_text = 'Negative'
        else:
            pred_text = 'Positive'

        # round the predict proba value and set to new variable
        confidence = round(pred_proba[0], 3)
        # create JSON object
        output = {'prediction': pred_text, 'confidence': confidence}

        return output
```

API REST

- Le code suivant définit l'URL de base pour la ressource de prédiction de sentiment.

- On peut imaginer avoir des URL multiples, chacune correspondant à un modèle différent, qui fournirait des prédictions différentes.

- Par exemple, on pourrait définir l'URL /ratings permettant de prévoir la classification d'un film à partir du genre, du budget et de l'équipe de production.

- On aurait a créer un autre objet ressource pour ce deuxième modèle.

- On peut simplement lister les ressources les unes après les autres, comme le montre le code suivant :

API REST

PYTHON

```
api.add_resource(PredictSentiment, '/')

# example of another endpoint
api.add_resource(PredictRatings, '/
ratings')
```

API REST

- On termine comme d’habitude par le bloc suivant, qui sert à lancer le serveur :

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

Requêtes

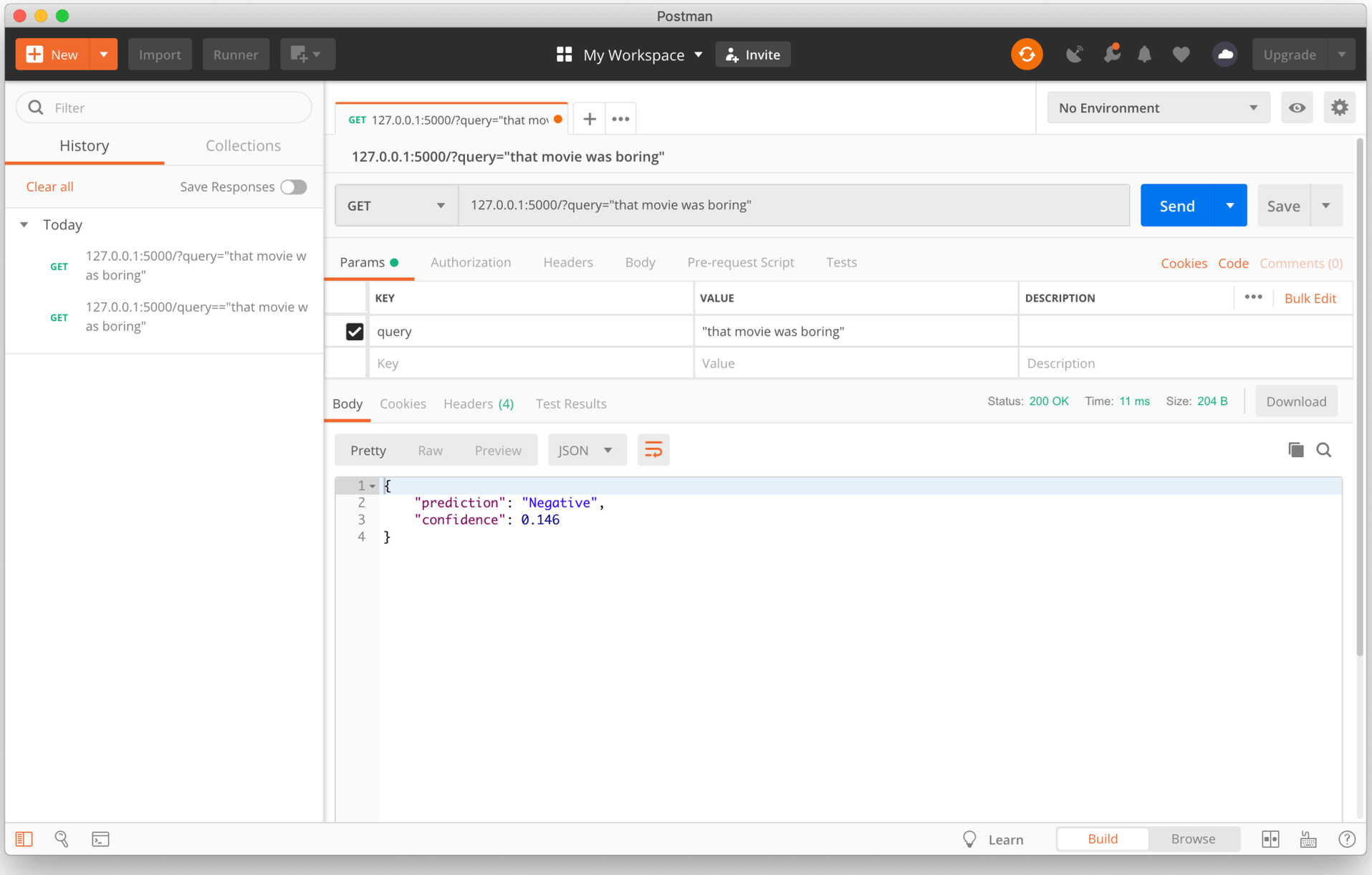
- L’ensemble du code est disponible sur le forum, dans le répertoire sentiment-clf.

- Pour que celui-ci s’exécute correctement, il faut s’assurer d’avoir installé flask-restful, par exemple à l’aide de la commande terminal : conda install -c conda-forge flask-restful.

- On va voir à présent comment requêter notre API.

- Une façon simple est d’utiliser Postman.

Requêtes


Postman
New
Import
Runner
My Workspace
Invite
Upgrade
Filter
History
Collections
Clear all
Save Responses
Today
GET 127.0.0.1:5000/?query="that movie was boring"
GET 127.0.0.1:5000/query=="that movie was boring"
GET 127.0.0.1:5000/?query="that mov
No Environment
127.0.0.1:5000/?query="that movie was boring"
GET
127.0.0.1:5000/?query="that movie was boring"
Send
Save
Params
Authorization
Headers
Body
Pre-request Script
Tests
Cookies
Code
Comments (0)
KEY
VALUE
DESCRIPTION
Bulk Edit
query
"that movie was boring"
Key
Value
Description
Body
Cookies
Headers (4)
Test Results
Status: 200 OK
Time: 11 ms
Size: 204 B
Download
Pretty
Raw
Preview
JSON
{
"prediction": "Negative",
"confidence": 0.146
}
Learn
Build
Browse

Requêtes

- Une façon encore plus simple, s’agissant d’une requête GET, est d’utiliser la barre de navigation, ce qui donne :

Requêtes

Conclusion

- On voit donc qu'on peut utiliser tout autant des requêtes GET que des requêtes POST pour déployer des modèles de Machine Learning sur le Web.

- De plus, on a vu que Flask-RESTful fournissait une façon de plus de créer des API REST à l'aide de Flask.

- Une fois un modèle de Machine Learning déployé sur le Web, ses prévisions sont accessibles à tous, programmes comme humains.

- Si on souhaite limiter l'accès à l'API, le plus simple est de demander un mot de passe pour la connexion au serveur. Voir les références à ce sujet.

Ressources Internet

- *Creating Web APIs with Python and Flask*, Patrick Smyth, 2018 : https://programminghistorian.org/en/lessons/creating-apis-with-python-and-flask.

- *Building and Documenting Python REST APIs with Flask and Connexion*, Part 1, Doug Farrell, 2018 : https://realpython.com/flask-connexion-rest-api/.

- *Building and Documenting Python REST APIs with Flask and Connexion*, Part 2, Doug Farrell, 2018 : https://realpython.com/flask-connexion-rest-api-part-2/.

Ressources Internet

- *A beginner's guide to training and deploying Machine Learning models using Python*, Ivan Yung, 2018 : https://medium.freecodecamp.org/a-beginners-guide-to-training-and-deploying-machine-learning-models-using-python-48a313502e5a.

- *Deploying a Machine Learning model as a REST API*, Nguyen Ngo, 2018 : https://towardsdatascience.com/deploying-a-machine-learning-model-as-a-rest-api-4a03b865c166.

Ressources Internet

- *Flask RESTful documentation,* 2018 : https://flask-restful.readthedocs.io/en/latest/index.html.

- *Authentication, Authorization and Access Control*, Apache Documentation, 2013 : http://httpd.apache.org/docs/2.0/howto/auth.html.

Bibliographie

- *Flask Web Development : Developing Web Applications with Python* (2ème édition). M. Grinberg. O'Reilly 2018.

- *Architectural Styles and the Design of Network-Based Software Architectures*. T. Fielding. Thèse, Université de Californie, 2000.