Module 3

Développement de services REST avec Python

Contenu du module

- Présentation des Web Services
  - Fonctionnement, intérêt, interopérabilité
- Architecture orientée service (SOA)
  - Composantes, technologies
- Architectures REST
  - Les principes et standards : HTTP, Méthodes et URI
  - Les formats de données
- Les outils Python pour implémenter des service REST
  - Flask
  - Django
- Récupérer des données dans la requête
  - Dans l'URL
  - Dans le corps
  - Envoi de paramètres, validations
- Gestion de la réponse
- Gestion des erreurs
- Déploiement d'un service RESTful
- Interrogation de web services REST

Concepts et architectures REST

Introduction

- Les protocoles de communication Web sont un élément clé de l'infrastructure Web 2.0. Les deux approches principales sont REST et SOAP.
  - **REST** (*REpresentational State Transfer*) indique une façon d'échanger et de manipuler des données en utilisant simplement les *verbes* HTTP GET, POST, PUT et DELETE.
  - **SOAP** implique de poster à un serveur des requêtes XML comprenant une suite d'instructions à exécuter.
- Dans les deux cas, les accès aux services sont définis par une interface de programmation (API).
  - Souvent, l'interface est spécifique au serveur.
- Cependant, des interfaces de programmation Web standardisées (par exemple, pour poster sur un blog) sont en train d'émerger. La plupart, mais pas toutes, des communications avec des Services Web impliquent une transaction sous forme XML (*eXtensible Markup Language*).

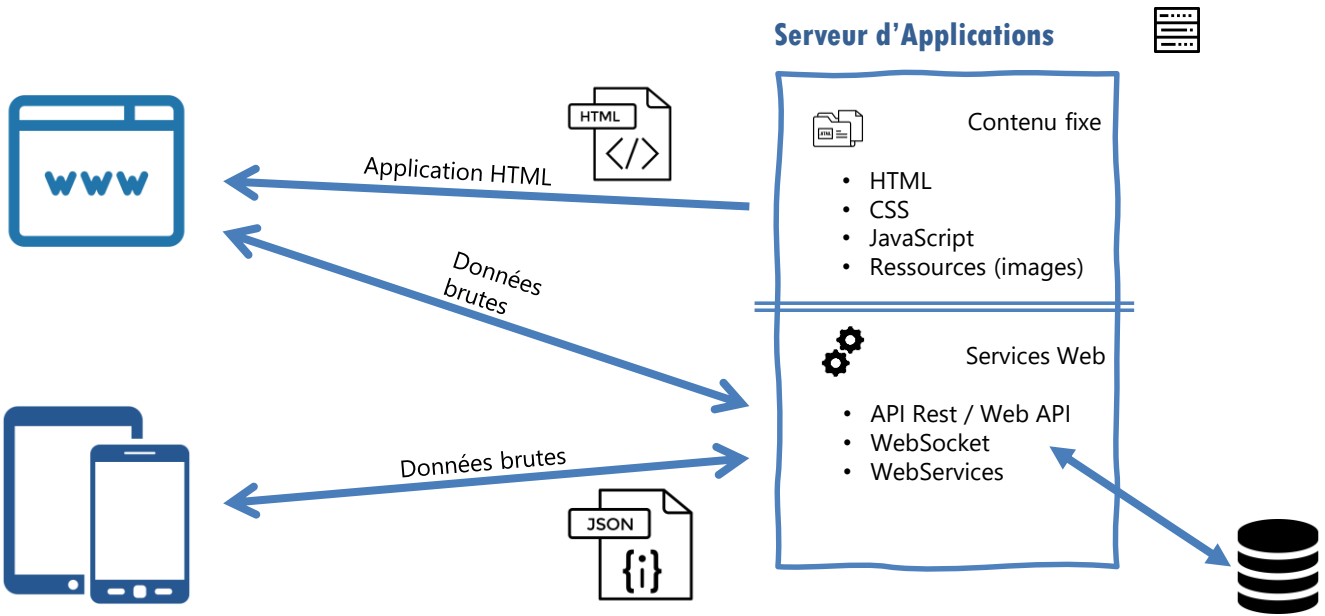
Principes de REST

- Cette architecture part du principe selon lequel Internet est composé de ressources accessibles à partir d'une URL.
  - Par exemple, pour avoir le temps à Paris, un utilisateur pourrait utiliser une adresse de la forme http://www.meteo.fr/paris : Paris serait alors une ressource telle que définie par Météo France.
- A la requête de cet URL serait renvoyée une représentation de la ressource demandée (ex : paris.html). Cette représentation place l'application cliente dans un état (*state*) donné.
- Si l'application cliente lance un appel sur un des liens de la représentation en cours, une autre ressource est appelée, dont une représentation est envoyée. Ainsi, l'application cliente change d'état (*state transfer*) pour chaque représentation de ressource.

Caractéristiques de REST

- Utilise les standards de l'Internet
  - Architecture orientée ressource basée sur des URI (*Uniform Resource* Identifier)
  - Repose sur l'utilisation du protocole HTTP et de ses méthodes (POST, GET, PUT, DELETE)
- Echange de données multi-formats
  - en XML ou autres (JSON, objets sérialisés en binaire)
- Utilisation des standards hypermedia : HTML ou XML qui permettent de faire des liens vers d'autres ressources et d'assurer ainsi la navigation dans l'application REST
- Utilise les types MIME pour la représentation des ressources (text/xml, image/jpeg, application/pdf, text/html, ...)
- Implémentation restreinte : il faut comprendre l'architecture REST et ensuite concevoir des applications ou des services Web selon cette architecture
- Alternative à SOAP censée être plus simple
- Aucun langage standard de description de service
- N'est pas un standard

Architecture REST



Principe de fonctionnement de REST

- Les méthodes du protocole HTTP servent à définir le type traitement à effectuer

Action	SQL	HTTP
Create	Insert	POST
Read	Select	GET
Update	Update	PUT
Delete	Delete	DELETE

- Donc pour l'URI `/client/123`, l'action sur cette ressource est décidée par la méthode HTTP utilisée.
  - Ce ne sont cependant que des conventions !
  - L'implémentation est à réaliser par le programmeur.

Règles de conception d'un Service REST

- Toutes les ressources devant être exposées au travers du service doivent être correctement identifiées, et de manière unique. Chaque ressource devra se voir assigner une URL.
  - Qui plus est, l'URL en question devra être de la forme http://www.site.com/contenus/003 plutôt que http://www.site.com/contenus?id=003.
- Les ressources doivent être catégorisées selon leurs possibilités offertes à l'application cliente
  - Ne peut-elle que recevoir une représentation (GET) ?
  - Ou bien peut-elle aussi modifier/créer une ressource (POST, PUT, DELETE) ?
- Chaque ressource devrait faire un lien vers les ressources liées
- La manière dont fonctionne le service sera décrite au sein d'un simple document HTML
  - Qui servira d'interface et de documentation d'API

Exemple de Service REST

- Prenons une entreprise de jouets qui veut permettre à ses clients d'obtenir une liste des jouets disponibles à la vente et d'obtenir des informations sur un jouet précis.
- La liste des jouets est disponible à l'URL suivante : http://www.jktoys.com/jouets
- Le client reçoit une réponse sous la forme suivante :

```
<?xml version="1.0"?>
<p:Jouets xmlns:p="http://www.jktoys.com/"
xmlns:xlink="http://www.w3.org/1999/xlink">
    <Jouet id="0001"
xlink:href="http://www.jktoys.com/jouets/0001"/>
    <Jouet id="0002"
xlink:href="http://www.jktoys.com/jouets/0002"/>
    <Jouet id="0003"
xlink:href="http://www.jktoys.com/jouets/0003"/>
[...]
</p:Jouets>
```

- La liste des jouets contient des liens pour obtenir des informations sur chaque jouet. C'est là la clef de REST : le lien entre les ressources. Le client peut ensuite choisir parmi les liens proposés pour aller plus loin.

Exemple de Service REST - suite

- Les détails d'un jouet se trouvent à l'URL : http://www.jktoys.com/jouets/0002
- Ce qui renvoi la réponse :
  - ```
    <?xml version="1.0"?>
    <p:Jouet xmlns:p="http://www.jktoys.com"
    xmlns:xlink="http://www.w3.org/1999/xlink">
        <Jouet-ID>0002</Jouet-ID>
        <Nom>Bisounours : Gros Câlin</Nom>
        <Description>Coeur sur le ventre</Description>
        <Details xlink:href=
    "http://www.jktoys.com/jouets/0002/details"/>
        <CoutUnitaire monnaire="EUR">30</CoutUnitaire>
        <Quantite>37</Quantite>
    </p:Jouet>
    ```
- A nouveau, d'autres ressources sont accessibles grâce à un lien...
    ```

JSON

- Le format JSON (*JavaScript Object Notation*) est un format de données textuel inspiré de la notation des objets JavaScript créé à partir de 2002 par Douglas Crockford.
- L'objectif du format JSON est de faciliter les échanges de données applicatives sur le Web qui étaient auparavant systématiquement basés sur XML
  - Jugé trop verbeux et donc consommateur de bande passante, JSON constitue une alternative plus légère pour ces échanges.

Représentation des données en JSON

- JSON représente les données sous forme d'objets, ils sont constitués d'attributs exprimés grâce à des paires clé/valeur, la clé correspondant au nom de l'attribut et la valeur à la donnée associée.
  - Cette donnée peut être un numérique, une chaine de caractères, un autre objet ou bien un tableau.
- Les clés sont des chaines de caractères et il est nécessaire de les exprimer entre guillemets.
- Les valeurs sont séparées des clés par le caractère deux points (:) et chaque couple clé/valeur et séparé du suivant par une virgule (,).
- Exemple de structure :

```
...
"clé1": "Valeur de clé1",
"clé2": "Valeur de clé2"
...
```

JSON : Type de données

- Chaines de caractères
  - Elles sont exprimées en Unicode et entre guillemets ;
- Les numériques
  - Entiers ou nombres décimaux ;
- Les booléens
  - Exprimés avec les mots réservés **true** et **false** ;
- Les objets
  - La base de la notation de JSON ;
- Les tableaux
  - Des ensembles de valeurs des types précédents.

Structures JSON

- Les paires de clé/valeur de JSON sont exprimés entre accolades ({}), c'est la manière de représenter un objet. Ainsi si l'on souhaite exprimer les données d'un objet **personne** qualifié par un **nom**, un **prénom** et un **âge**, on pourrait utiliser la structure suivante :

```
{
    "nom": "DUPONT",
    "prenom": "Robert",
    "age": 56
}
```

- Dans cet exemple, les valeurs pour le nom et le prénom sont des chaines de caractères, l'âge est un entier.
- Les tableaux sont quant à eux exprimé entre crochets ([]), par exemple :

```
...
"clé1": [ "Première valeur de clé1", "Deuxième valeur de clé1" ]
...
```

L'essentiel de REST : HTTP

- Comprendre les mécanismes du protocole HTTP est essentiel pour comprendre le fonctionnement des architectures REST !
- La requête !
  - Les méthodes/verbes HTTP
    - GET, POST, PUT, DELETE
    - Leur signification, différences, usages et objectifs dans une API REST
  - L'identification unique des ressources
    - Associée à une méthode/un verbe HTTP !
- La réponse !
  - Les codes de réponse !
    - 1xx, 2xx, 3xx, 4xx, 5xx !
    - Pour savoir les utiliser à bon escient !

Implémentation en Python

Implémenter des services REST en Python

- Plusieurs librairies et framework Python peuvent permettre la réalisation de services REST
- Parmi les plus populaires et plus complets, on trouve :
  - Django
    - https://www.djangoproject.com/
  - Flask
    - https://flask.palletsprojects.com
- Mais il existe d'autre librairies plus légères :
  - Bottle
    - https://bottlepy.org
  - Pyramid
    - https://trypyramid.com/
  - Tornado, Falcon, Masonite, FastAPI, …

Django

- https://www.djangoproject.com
  - "Django makes it easier to build better Web apps more quickly and with less code."
- Approche « Pythonique » pour le Web
  - Philosophie de Python
  - Concepts hérités d'autres frameworks Web dans d'autres langages…
- Framework complet !
  - Gestion HTTP, rendu des pages, accès aux données, …
  - Outillage de génération de code
- Large communauté et documentation importante
- Mais !
  - Lourd et complexe de prise en main et de mise en œuvre
  - Impose sa solution pour l'accès aux bases de données
  - Le support de REST est une extension qu'il faut ajouter et qui alourdit considérablement le framework

Flask

- Flask est un micro framework Web
- Allant à contre-pied d'autres solutions de développement web, Flask est livré avec le strict minimum, à savoir :
  - un moteur de template (Jinja 2)
  - un serveur web de développement (Werkzeug)
  - un système de distribution de requête compatible REST (dit RESTful)
  - un support de débogage intégré au serveur web
  - un micro framework doté d'une très grande flexibilité
  - une très bonne documentation
- La notion de « micro » signifie que Flask ne fait pas tout !
  - Tout du moins nativement…
- Il vise à garder le noyau simple mais extensible.
  - Flask n'inclut pas de couche d'abstraction de base de données, de validation de formulaire ou tout autre élément dans lequel différentes bibliothèques existent déjà et pouvant gérer cela.
- Flask prend en charge des extensions pour ajouter de telles fonctionnalités à votre application comme si elles étaient implémentée dans Flask lui-même.

Extensions pour Flask

- Un catalogue des extensions est disponible sur le lien suivant : http://flask.pocoo.org/extensions/
- Quelques références populaires :
  - Flask-Login : permet d'ajouter la gestion de sessions utilisateur à Flask. Cette extension prend en charge le login, la déconnexion et mémorisation de la session utilisateur durant un certain temps.
  - Flask-Uploads : permet de prendre en charge le téléversement et le stockage de fichiers sur le serveur. Cette extension permet également de télécharger les fichiers stockés sur le serveur.
  - Flask-SQLAlchemy : apporte le support pour SQLAlchemy à Flask. SQLAlchemy est un ORM (Object Relational Mapper) très puissant supportant de nombreuses bases de données. Un must !
  - Flask-Babel : ajout du support d'internationalisation i18n et localisation l10n à une application Flask en s'appuyant sur le projet Babel. La fonction gettext() et l'utilitaire pybabel permettent de traduire facilement le contenu de l'application.
  - Flask-Mail : permet à une application Flask d'envoyer très facilement des e-mails.

Application Flask

- L'application utilisant le framework Flask traite les requêtes entrantes (communiquées par le serveur) et produit les réponses correspondantes.
- L'application Flask contient des « routes » permettant de réceptionner et de traiter les requêtes des clients pour produire les réponses correspondantes.
- Pour réaliser cette tâche, l'application peut utiliser des connexions vers des bases de données afin d'obtenir les éléments nécessaires au rendu de la page.
  - La connexion sur une base de données peut être établie soit en utilisant les modules Python disponibles, soit en utilisant des extensions Flask comme Flask-SQLAlchemy (ORM) ou Flask-CouchDB.
- De même, l'application Flask peut produire directement du contenu HTML, CSV, JSON, XML, texte et autre.

Installation de Flask

- Flask, ainsi que toutes ses extensions, s'installe avec PIP
  - `pip install flask`
- Il est conseillé de l'installer spécifiquement pour le projet dédié en créant un environnement virtuel pour ce dernier.

Structure d'une application Flask

- La mise en œuvre nécessite de créer une instance de la classe Flask :
  - Cet objet est fondamental : il s'agit de notre application, ou de notre site web
  - En termes techniques, il s'agit d'une application WSGI.
    - WSGI (Web Server Gateway Interface) est une norme et un protocole de communication qui définissent comment un serveur web peut interagir avec une application Python pour envoyer des requêtes et recevoir des réponses.
      - WSGI est une norme pour serveur web comme l'est FastCGI

```python
from flask import *

app = Flask(__name__)

# Démarrer le serveur embarqué
if __name__ == '__main__':
    # Lance le serveur sur port 5000
    app.run()
```

Le serveur interne

- Flask est fournit avec un serveur Web WSGI interne : Werkzeug
  - Il écrit en Python
  - Il reçoit les requêtes HTTP et fait le nécessaire pour décoder les demandes et envoyer les réponses en retour.
  - Werkzeug s'occupe des problématiques de gestion des sockets et connexions, traitement des tâches en parallèles, prise en charge des aspects sécuritaires (au niveau HTTP), etc.
- Werkzeug est très pratique en phase de développement et de test, cela évite de devoir déployer son application sur un serveur réel. En production en revanche, on envisagera de déployer l'application sur un vrai serveur Web compatible Python
- La méthode `run()` de la classe `Flask` permet de lancer ce serveur
  - Il se met par défaut en écoute sur le port 5000
  - `app.run(debug=True, port=8085, host='0.0.0.0')`

Création de services

- La définition de services se fait en créant une fonction Python qui va renvoyer l'information demandée.
- On va utiliser le décorateur `@route` pour associer l'action à une URL.

```python
from flask import *

app = Flask(__name__)


@app.route("/")
def hello():
    # Par defaut un flux HTML et le code de réponse 200 OK
    return "Hello, World!"


if __name__ == '__main__':
    app.run(port=8080)
```

Spécification de la méthode HTTP

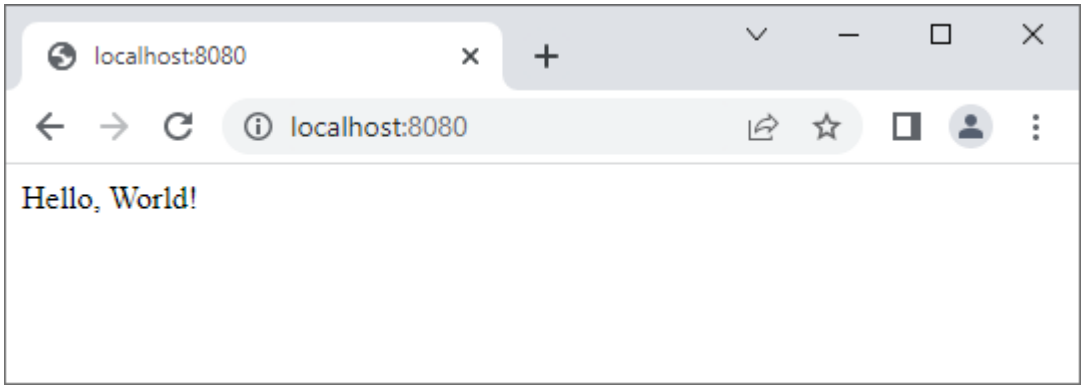
- Le décorateur `@route` permet de spécifier le routage HTTP en précisant :
  - L'URI d'invocation
  - La/Les méthode(s) HTTP à utiliser pour invoquer la fonction grâce à l'attribut `methods`
    - Sans précision, toutes les méthodes peuvent être utilisées

```
@app.route('/personne', methods=['POST'])

@app.route('/add', methods=['POST', 'PUT'])
```

Test simple du service

- Vous pouvez tester l'utilisation du service web avec un simple navigateur :
- Lancer l'application
- Saisissez l'URL de la fonctionnalité dans le navigateur :
  - http://localhost:8080 -> Appel de la méthode HTTP GET



Récupérer des données dans l'URL de la requête

- Le décorateur `@route` permet de spécifier des valeurs variables dans l'URL
  - La valeur sera à transmettre par le client au moment de l'invocation de la requête
- Il s'agit de fournir des paramètres à une URL
- Il faut pour cela ajouter le nom de la variable dans l'URL sous la forme `<nom_variable>`
- La fonction reçoit alors automatiquement la valeur transmise dans un paramètre de la fonction nommé comme la variable

```python
@app.route('/formation/date/<langage>')
def get_date_formation(langage):
    date = ""
    if langage == "Python":
        date = "01/09/2020"
    elif langage == "Java":
        date = "08/09/2020"
    return date
```

Contraintes sur l'URL

- Il est possible d'utiliser un convertisseur pour spécifier et imposer le type du paramètre de l'URL avec la syntaxe sous la forme `<convertisseur:nom_variable>`
- Les convertisseurs disponibles :

Convertisseurs	Type
`string`	Chaîne de caractères
`int`	Nombres entiers
`float`	Nombres réels

Récupérer des données dans le corps de la requête

- Lorsque l'on utilise des requêtes PUT et POST, il sera nécessaire de pouvoir récupérer les données transmises dans le corps (Body) de la requête.
  - Cela se fait avec l'attribut `data` de l'objet global `request`
- On pourra ensuite appeler une fonction de désérialisation pour reconstruire un objet à partir du flux

```python
@app.route('/personne', methods=['POST'])
def ajouter_personne():
    # Récupérer les données de la requete
    data = request.data
    # Désérialisation JSON -> objet
    objet = loads(data)
    # ...
```

Format de données utilisables

- Un service REST peut travailler avec tout type d'informations :
  - Type de base
  - Objet Python
  - Liste, Ensemble, Dictionnaire
- Cependant, il est nécessaire de travailler avec des formats du Web pour les échanges client/service.
  - HTML, XML, Texte brute, JSON, …
- Il faut donc disposer d'outils permettant de :
  - Convertir une données Python en flux
    - Sérialisation
  - Convertir un flux en données Python
    - Désérialisation

Sérialisation / Désérialisation en JSON

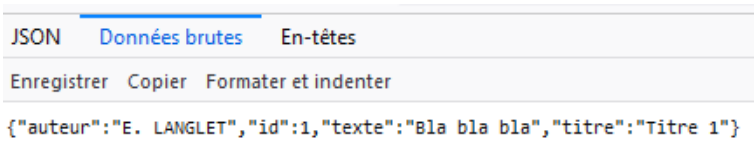
- Avec Flask, la fonction `jsonify()` permet de transformer une structure de données en flux JSON
  - Elle sera donc utilisée pour construire une réponse au format JSON
- Pour ce qui est de la désérialisation, il n'y a pas de fonction proposé par Flask.
  - On utilisera la fonction `loads()` du module `json` de Python
  - A noter que l'on pourrait utiliser la méthode `dump()` de ce même module pour la sérialisation

Exemples

- Sérialisation

```python
@app.route('/article/<int:id>', methods=['GET'])
def lire_article(id: int):
    # La classe Article possède des propriétés...
    a = Article(id, "Titre 1", "E. LANGLET", "Bla bla bla")
    return jsonify(
        id=a.id,
        titre=a.titre,
        auteur=a.auteur,
        texte=a.texte
    )
```

- Résultat obtenu



- Désérialisation

```python
@app.route('/article', methods=['POST'])
def ajouter_article():
    # Récupérer les données de la requete
    data = request.data
    # Désérialisation JSON -> objet
    objet = loads(data)
    print(objet)

    return "Ok"
```

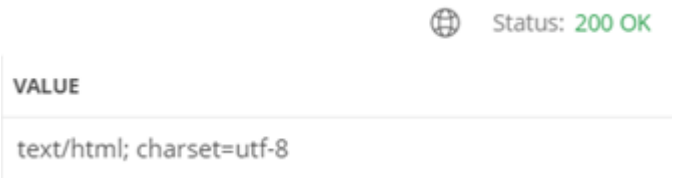
- Résultat obtenu

```
{'id': 2, 'titre': 'Depuis postman', 'auteur': 'E. LANGLET', 'texte': 'Re- Bla bla bla...'}
127.0.0.1 - - [21/Jun/2022 18:20:42] "POST /article HTTP/1.1" 200 -
```

Gestion par défaut de la réponse

- Les fonctions d'un service renvoie l'information demandée et produite par le traitement

```python
@app.route("/message", methods=['GET'])
def get_message():
    return "Ceci est un service Rest Python"
```



- Par défaut Flask renvoie la réponse au format et l'écrit dans le corps du message de réponse.
- Le code de réponse pour cette méthode est 200, en supposant qu'il n'existe pas d'exception non gérée.
- Les exceptions non gérées sont converties en erreurs 5xx.

Gestion explicite de la réponse

- Une action peut renvoyer n'importe quel type de valeurs en spécifiant le type attendu.
- Cette technique a certains inconvénients dans le contexte de services Web :
  - Ne permet pas de définir un code de réponse particulier, notamment en cas d'erreur.
  - Ne permet pas de retourner des valeurs différentes en fonction des arguments reçus, ou du traitement réalisé.
  - Le framework met à disposition la méthode `make_response()` pour résoudre ces problèmes et construire une réponse plus précise et personnalisée

Gestion de la réponse : `make_response()`

- La fonction `make_response()` créé un objet de type `Response`
- Il est ensuite possible d'agir sur cet objet pour :
  - Spécifier le code de réponse
  - Ajouter des entêtes comme le `Content-Type` notamment

```python
@app.route('/article/<int:id>', methods=['GET'])
def lire_article(id: int):
    # La classe Article possède des propriétés...
    a = Article(id, "Titre 1", "E. LANGLET", "Bla bla bla")

    response = make_response(jsonify(
            id=a.id,
            titre=a.titre,
            auteur=a.auteur,
            texte=a.texte
        )
    )
    response.status_code = 200
    response.headers.set("Content-type", "application/json; charset=utf-8")
    return response
```

Gestion des exceptions

- Toutes les exceptions non-gérés dans l'application Flask conduiront à des erreurs 5xx
- Il est donc conseillé de fournir une gestion fine des exceptions afin de pouvoir retourner les informations souhaitées ainsi qu'un code d'erreur pertinent pour le client

```
try:
    """
    Traitement ...
    """

    response = make_response(
        """
        Production de la réponse...
        """
    )
    response.status_code = 200
    response.headers.set("Content-type", "application/json; charset=utf-8")
    return response
except Exception as e:
    message = {"message": e.__str__()}
    return make_response(jsonify(message), 404)
```

Les clients REST

Le navigateur Web

- Pour tester ses services GET, un simple navigateur Web peut être utilisé

- Pour les autres méthodes HTTP, on peut envisager une application cliente HTML/CSS/JS (avec ou sans Framework JS) qui emmétrait des requêtes vers les services…

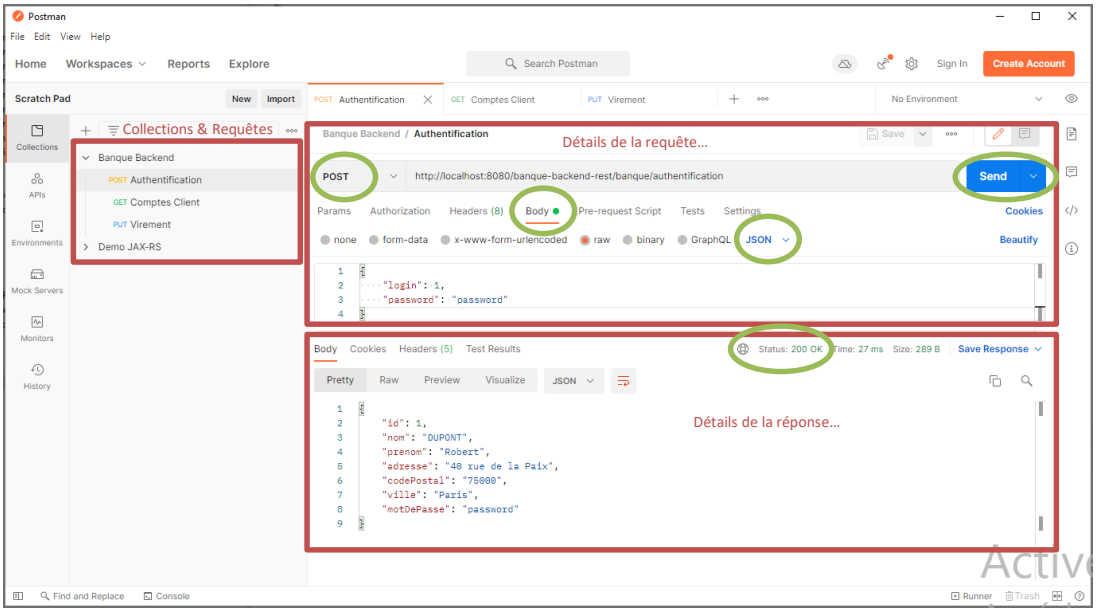
- Mais il est souvent nécessaire de tester ses services avant de les consommer avec des applications clientes !

Tester ses services REST : Approches

- Tests automatisés
  - Tests d'intégration réalisés
  - Ils nécessiteront d'utiliser une API cliente pour consommer les services
  - Fondamentaux pour assurer la non régression
  - Nécessite que les services soient testés manuellement en amont
- Outils de test dédiés
  - SOAP UI
    - Convient pour les services REST et SOAP
    - Version Open Source (*SOAP UI Open Source*) et commerciale (*ReadyAPI*)
    - https://www.soapui.org/
  - Postman
    - Initialement développé par Google sous forme d'un plugin pour Chrome
    - Application standalone ou Cloud
    - https://www.postman.com/

Tests d'API REST avec Postman

- Un compte Postman peut être créé afin de pouvoir stocker ses projets de test dans le Cloud
- Postman utilise le concept de « Collections » pour ranger ses requêtes de test
  - Notion de « projet de test »



Consommation d'un service en Python

- La consommation d'un service Rest en Python consiste à utiliser une API cliente HTTP pour émettre des requêtes et traiter les réponses
- Plusieurs bibliothèques peuvent être utilisées
  - `urllib.request`
    - Bibliothèque standard Python
  - `aiohttp`
    - https://docs.aiohttp.org
    - Gestion efficace de la concurrence
  - `requests`
    - https://requests.readthedocs.io

Exemple avec `urllib.request`

```python
import urllib.request


if __name__ == "__main__":
    with urllib.request.urlopen('http://localhost:8080/article/1') as service:
        code = service.getcode()
        if code == 200:
            contenu = service.read()
            print(contenu)
```

Exemple avec `requests`

```
import requests

if __name__ == "__main__":

    with requests.get('http://localhost:8080/article/1') as request:
        code = request.status_code
        if code == 200:
            contenu = request.json()
            print(contenu)
```

Travaux Pratiques