

Architectures logicielles

REST – API – PYTHON JS



BACK-END



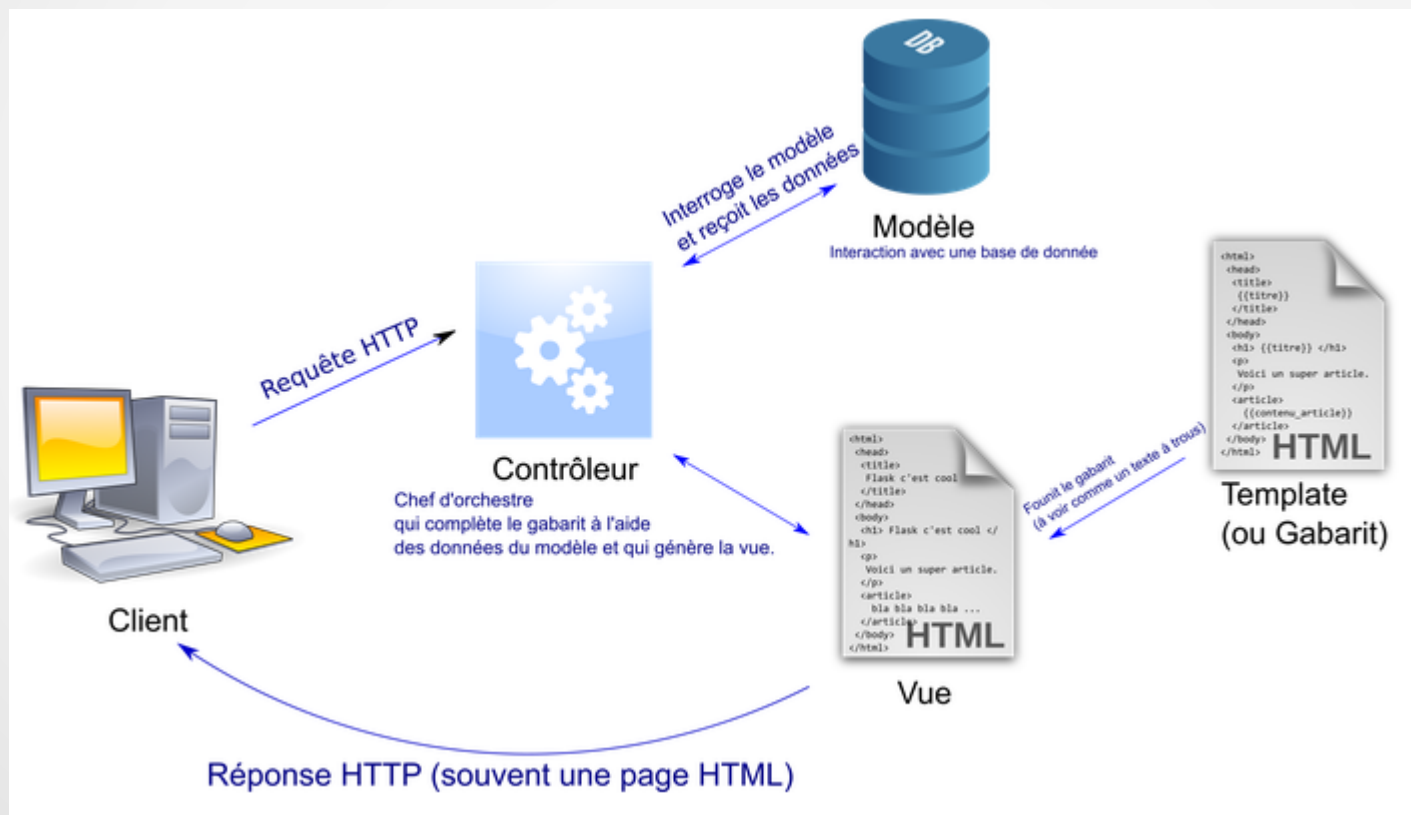
FRONT-END



APIs

Serveur Web type Flask

- Ici, notre programme principal est un routeur, qui fait office de contrôleur.



Le problème

- Page complète rechargée à chaque fois
- Mélange données présentation (statique et dynamique)
- Déséquilibre de traitement (tout est fait sur le serveur)
- Le client ne sert qu'à afficher, sans intelligence

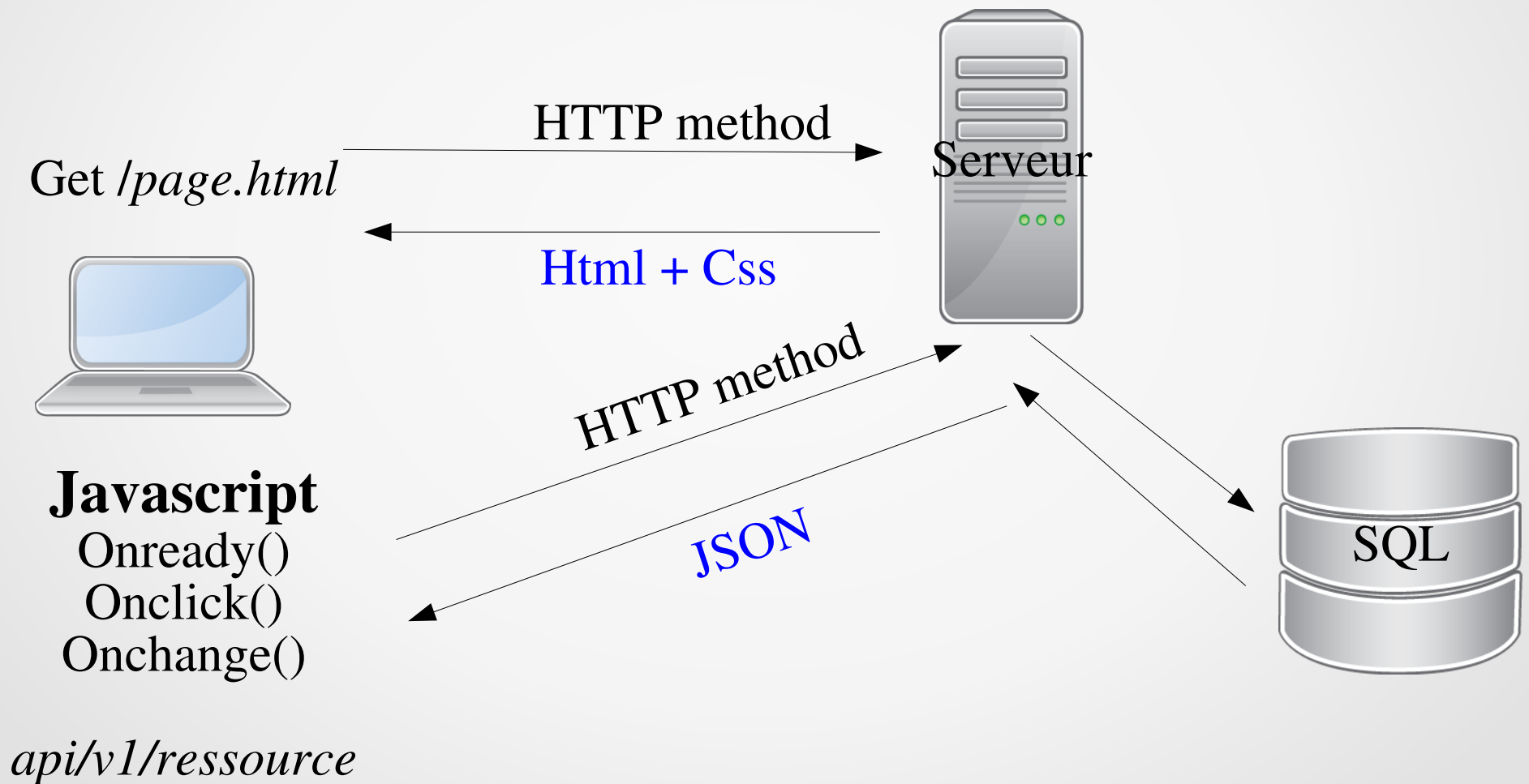
OR

- Souvent une grande partie du résultat est fixe
- Le client est programmable (Javascript)
- Il est en plus capable de mettre en cache
- Le serveur peut aussi servir des messages courts ne contenant que les données

Définir ce qu'est une API

- *Application Programming Interface*
- Permet d'interagir avec le programme
- Typiquement, l'interface graphique interagit avec le coeur du programme via son API
- Permet de découpler les services rendus et leur utilisation
- Permet d'« *exposer son API* » à des programmes tiers
- Liste de fonctionnalités offertes par le programme
- L'API doit être décrite (paramètres en entrée, en sortie, format (par exemple *json*, *xml*, *binaire*), structure)

Avec API



JSON Kesaco (json.org)

- paires de clé/valeur
- booléen, nombre, ou chaîne.. objet {} et tableaux[]

```
{  
  "menu": {  
    "id": 12,  
    "value": "gastronomique",  
    "détail": {  
      "plat": [  
        { "nom": "Tartiflette", "prix": 153 },  
        { "nom": "Raclette", "prix": 97 },  
        { "nom": "Choucroute", "prix": 102 }  
      ],  
      "vin": [  
        { "nom": "Pouilly fumé", "prix": 63 },  
        { "nom": "Sancerre", "prix": 97 }  
      ]  
    }  
  }  
}
```


Pourquoi JSON ?

- Très léger (contrairement à XML)
- Facilement manipulable en php, javascript, java, etc
- Typé
- Lisible par un être humain
- Affichable dans le navigateur (pour tests)

- Pas de commentaires
- Liste de types assez limitée (pas de date, pas de format binaire)
- Well-Formed, mais pas de Valid (contrairement à XML)
- On peut donc avoir un échange en json qui est bien écrit, mais incompris par l'un des deux intervenants (champs manquant, mauvaise valeur...)

Exemple API (opendata gouv.fr)

Découvrir les APIs du service public

Voir les réalisations

À propos

Demander une API

API Hub'Eau - Piézométrie

1.4.0

[Base URL: hubeau.eaufrance.fr/api]

https://hubeau.eaufrance.fr/api/v1/niveaux_nappes/api-docs

Adresse de la documentation <https://hubeau.eaufrance.fr/page/api-piezometrie>

niveaux-nappes Opérations sur le niveau des nappes



GET

/v1/niveaux_nappes/chroniques Lister les chroniques piézométriques

GET

/v1/niveaux_nappes/chroniques.csv Lister les chroniques piézométriques au format CSV

GET

/v1/niveaux_nappes/chroniques_tr Lister les chroniques piézométriques en temps réel

GET

/v1/niveaux_nappes/chroniques_tr.csv Lister les chroniques piézométriques au format CSV

GET

/v1/niveaux_nappes/stations Lister les stations de mesure



Exemple d'API



The screenshot shows the GitHub repository page for 'api-jours-feries-france' by AntoineAugusti. The page has a light green header with the title 'Exemple d'API'. The repository name is 'api-jours-feries-france' and the owner is 'AntoineAugusti'. The page content includes a description of the API, a section for deployment and hosting, and a section for endpoints. The endpoints section lists the API endpoints and provides an example of the JSON response for the year 2019.

API Jours Fériés France

Une API pour lister les jours fériés en France. Ceci se base sur les données [du package suivant](#).

Déploiement et hébergement

Le déploiement est assuré par Netlify, le fichier de build se trouve dans `build.py`.

Endpoints

Les endpoints retournent des données au format JSON.

- Les jours fériés pour une année : `https://jours-feries-france.antoine-augusti.fr/api/:annee`

Exemple :

- <https://jours-feries-france.antoine-augusti.fr/api/2019>

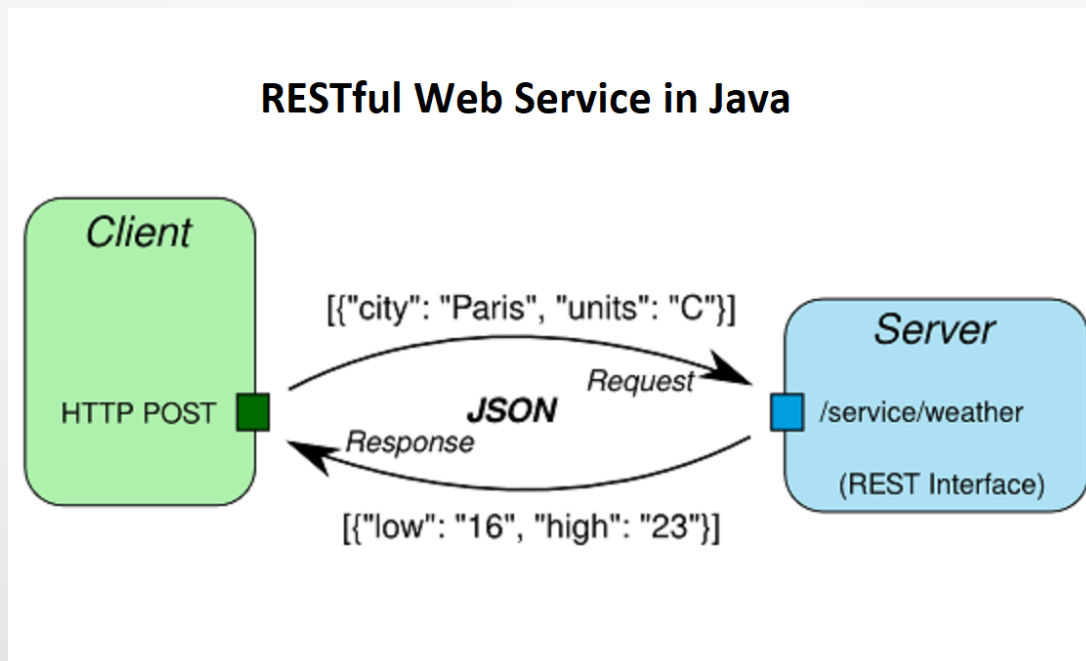
```
[{"date": "2019-01-01", "nom_jour_ferie": "Jour de l'an"}, {"date": "2019-04-22", "nom_jour_ferie": "Lundi de Pâques"}, {"date": "2019-05-01", "nom_jour_ferie": "Fête du travail"}, {"date": "2019-05-08", "nom_jour_ferie": "Victoire des alliés"}, {"date": "2019-05-30", "nom_jour_ferie": "Ascension"}, {"date": "2019-06-10", "nom_jour_ferie": "Lundi de Pentecôte"}, {"date": "2019-07-14", "nom_jour_ferie": "Fête du soldat inconnu"}, {"date": "2019-08-15", "nom_jour_ferie": "Assommoir"}, {"date": "2019-09-02", "nom_jour_ferie": "Fête de la musique"}, {"date": "2019-09-16", "nom_jour_ferie": "Fête de l'indépendance"}, {"date": "2019-10-01", "nom_jour_ferie": "Fête de la fraternité"}, {"date": "2019-10-29", "nom_jour_ferie": "Fête de la ville"}, {"date": "2019-11-01", "nom_jour_ferie": "Fête de la culture"}, {"date": "2019-11-11", "nom_jour_ferie": "Fête de l'armée"}, {"date": "2019-11-23", "nom_jour_ferie": "Fête de la famille"}, {"date": "2019-12-01", "nom_jour_ferie": "Fête de la neige"}, {"date": "2019-12-25", "nom_jour_ferie": "Noël"}]
```

Introduire les API REST

- Comment appeler l'API ?
- Votre système offre une API (mais comment l'appeler?)
- Au sein d'un même langage, les modules/bibliothèques sont assez facilement appelables (`import bib, use bib`)
- Si les langages sont différents, ou l'appelant et l'appelé sur des machines distantes, comment appeler le service ?
- HTTP !
- Pas suffisant pour être REST !

Notion de REST

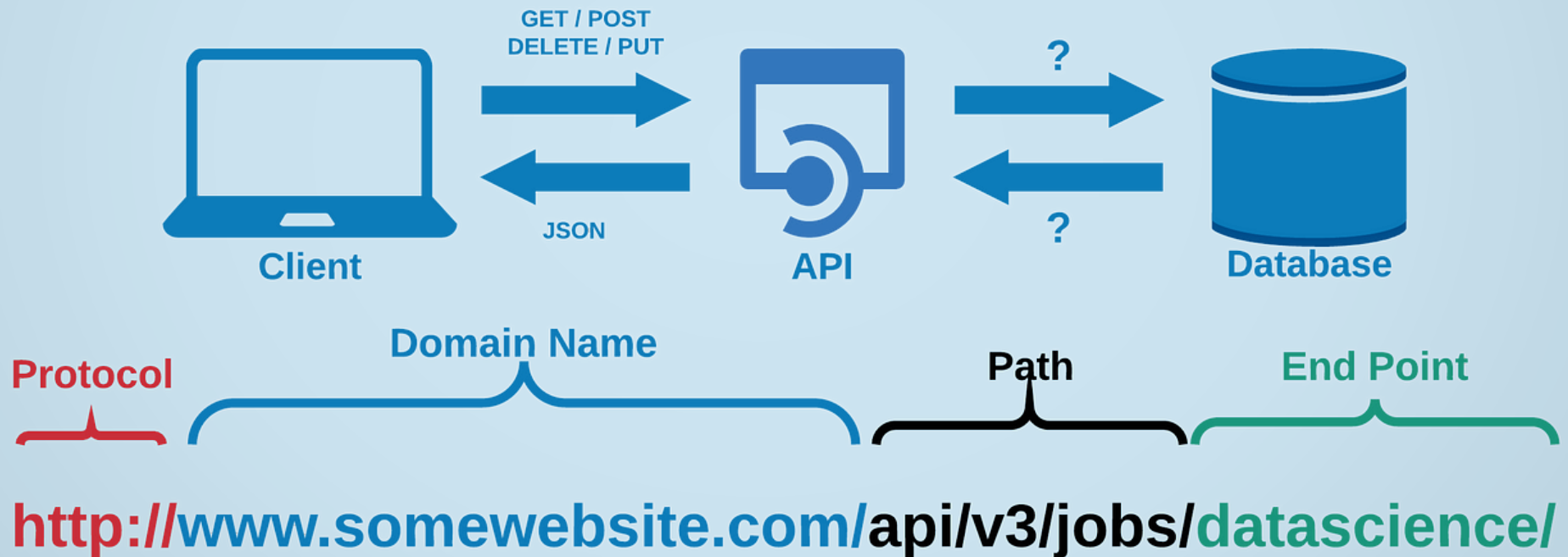
- Découpage atomique en ressources
- On accède à une ressource, on crée/modifie/supprime des ressources via GET ou POST/UPDATE/DELETE
- On interroge le système (ici avec l'ordre POST)



API Python Flask

Une route spécifique qui produit du JSON

Develop an API using Flask and Python3



API Python Flask

Une route spécifique qui produit du JSON

Flask REST API



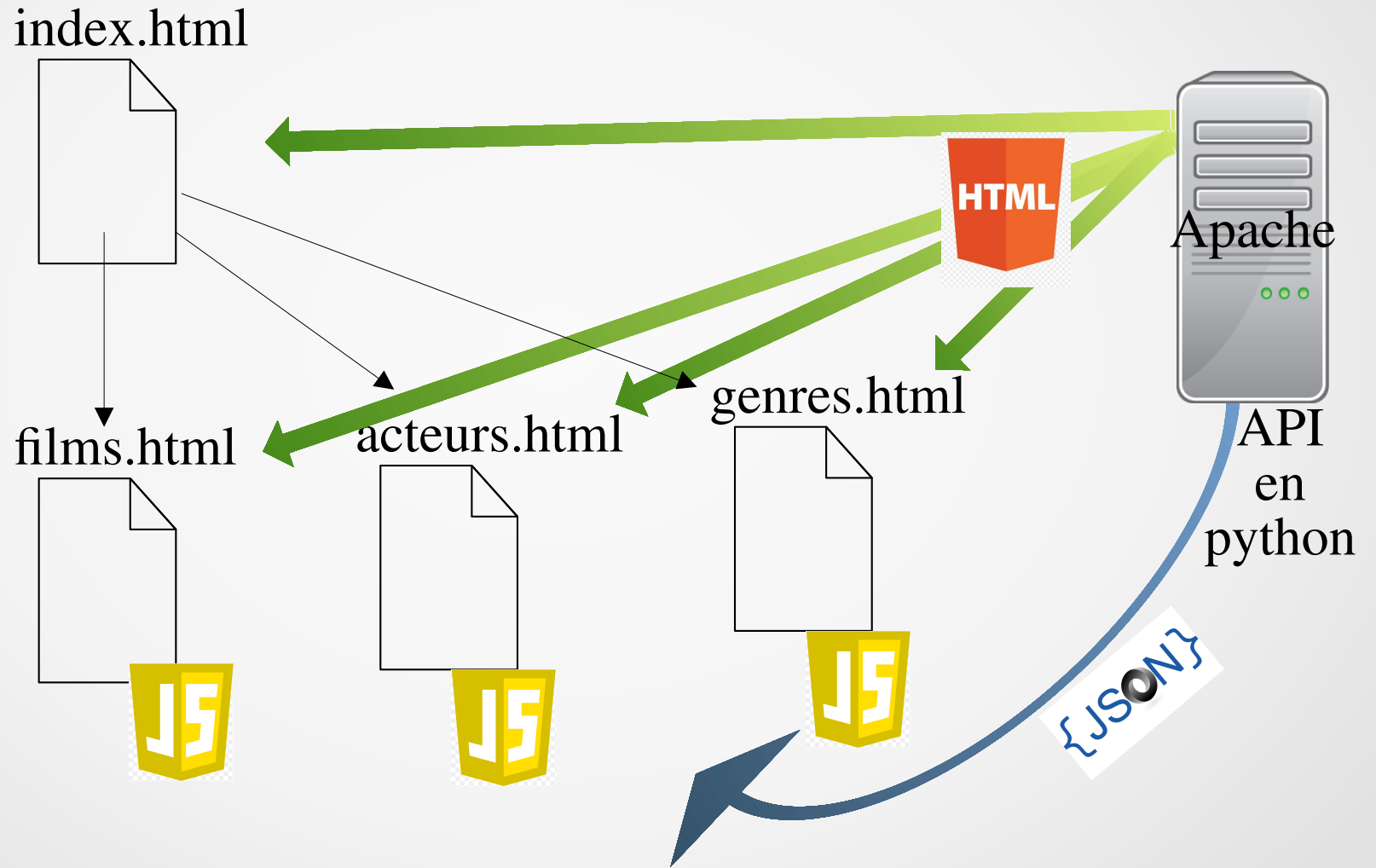
Exemple d'archi pour films

index.html
films.html
genres.html
acteurs.html

app.js (*contient des fonctions javascript qui lancent des requêtes sur `api/v1/xxxx`, et injectent les résultats JSON dans les pages*)

api/v1/acteurs (*get*)
api/v1/acteur (*get, put, post, delete*)
api/v1/films (*get*)
api/v1/film (*get, put, post, delete*)
api/v1/genres (*get*)
api/v1/genre (*get, put, post, delete*)

structure du site



déroulement

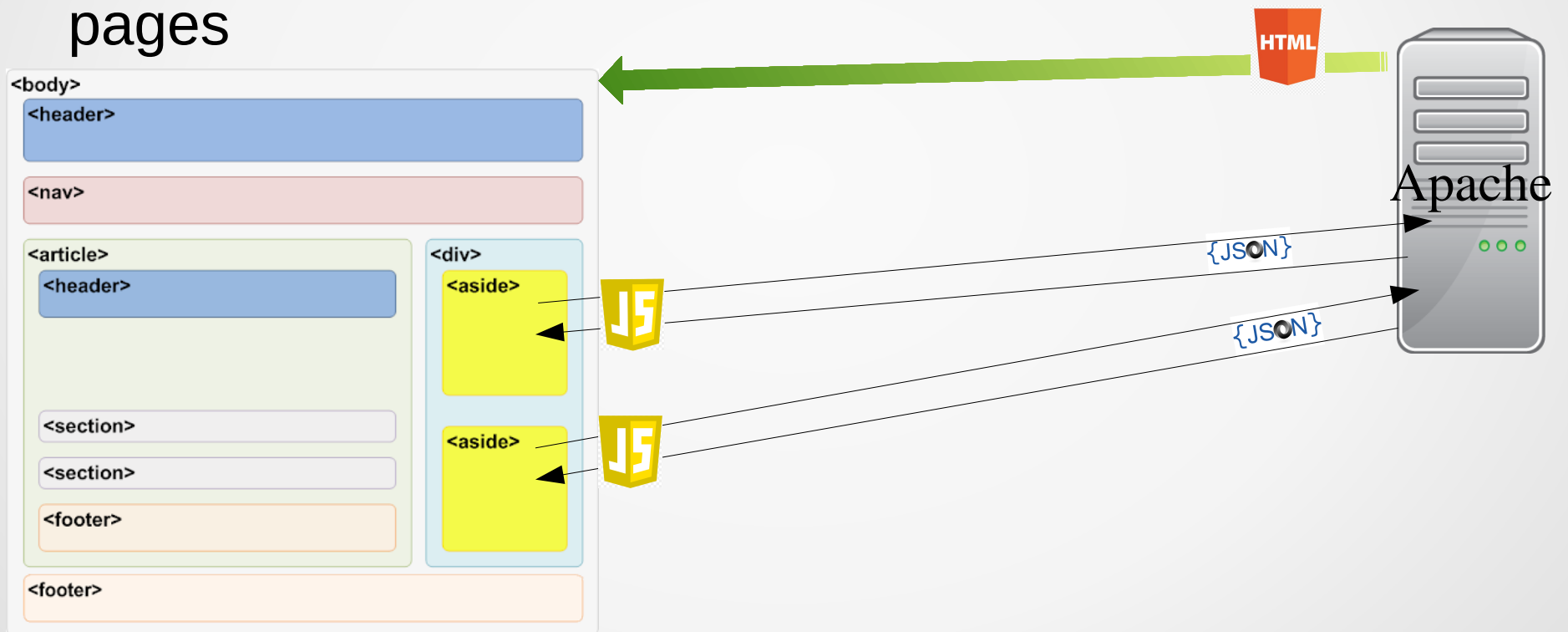
- **index.html** : principalement en html, elle offre les liens vers la suite
- **acteurs, films et genres** (python flask): affiche la base de la structure en html. Contiennent aussi des routines en javascript, pour décrire les actions : premier affichage, action d'ajouter, de supprimer, de modifier
- sur le serveur, une liste de fonctions en python : C'est **l'API. Ces fonctions sont liées à des endpoints, pour que les clients les interrogent.**
- les scripts javascript invoquent les « **endpoints** » de l'API selon les actions de l'utilisateur (ajout, suppression, modif) (parametres fournis en json)
- les « **endpoints** » en python répondent au format **JSON**
- le javascript altère la page selon la réponse

Définir l'architecture

- Lister les opérations de base
- lister les regroupements d'opérations
- les regroupements sont les endpoints de l'API, les opérations se feront par les verbes http (**get, post, put, delete**)
- Faire une présentation cohérente de l'api
 - *api/ressources* pour voir tout
 - *api/ressource/id* pour les actions sur une ressource

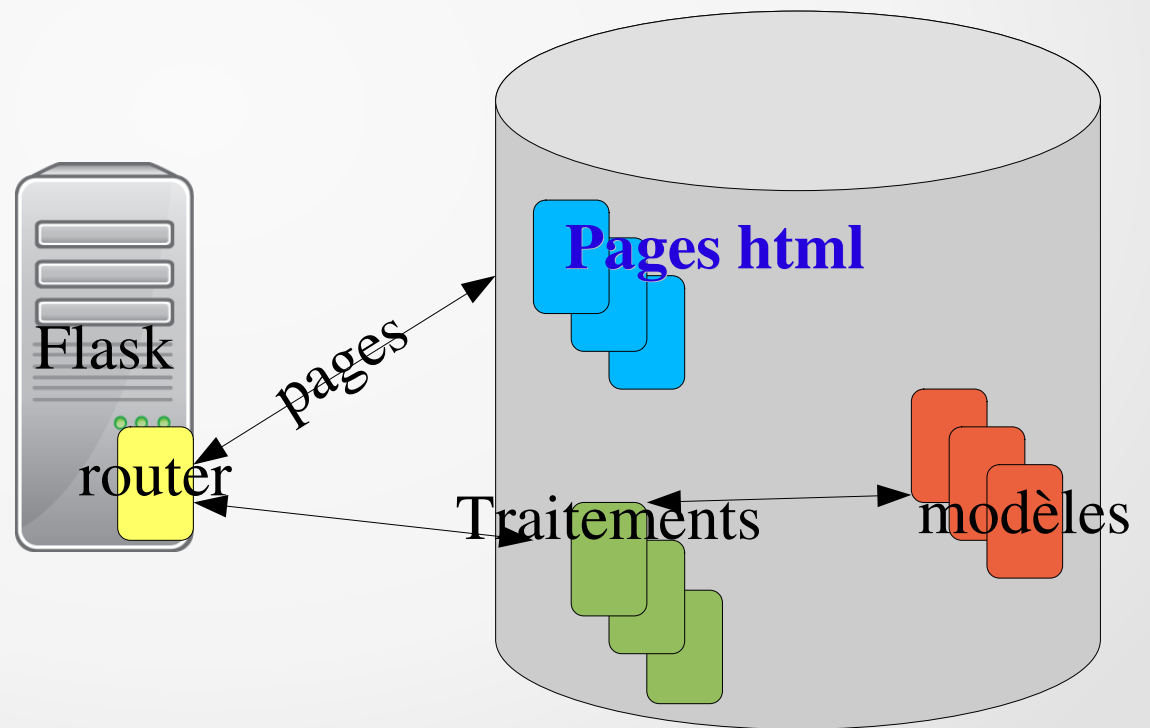
Définir les contenus

- Quelques pages de base complètes
- Beaucoup de fonctionnalités dynamiques, altérant les pages




Coté serveur

- Fournir des pages web complètes (en python, html)
- Fournir une liste de endpoints (API de l'application) en python qui fournissent du json



Exemple d'API (swagger website)

 **swagger**

http://petstore.swagger.io/v2/swagger.json

Explore

Swagger Petstore ^{1.0.0}

[Base url: petstore.swagger.io/v2]
<http://petstore.swagger.io/v2/swagger.json>

This is a sample server Petstore server. You can find out more about Swagger at <http://swagger.io> or on [#swagger](http://irc.freenode.net). For this sample, you can use the api key **special-key** to test the authorization filters.

[Terms of service](#)
[Contact the developer](#)
[Apache 2.0](#)
[Find out more about Swagger](#)

Schemes

HTTP

Authorize

pet Everything about your Pets

POST

/pet Add a new pet to the store

PUT

/pet Update an existing pet

GET

/pet/findByStatus Finds Pets by status

GET

/pet/findByTags Finds Pets by tags

GET

/pet/{petId} Find pet by ID

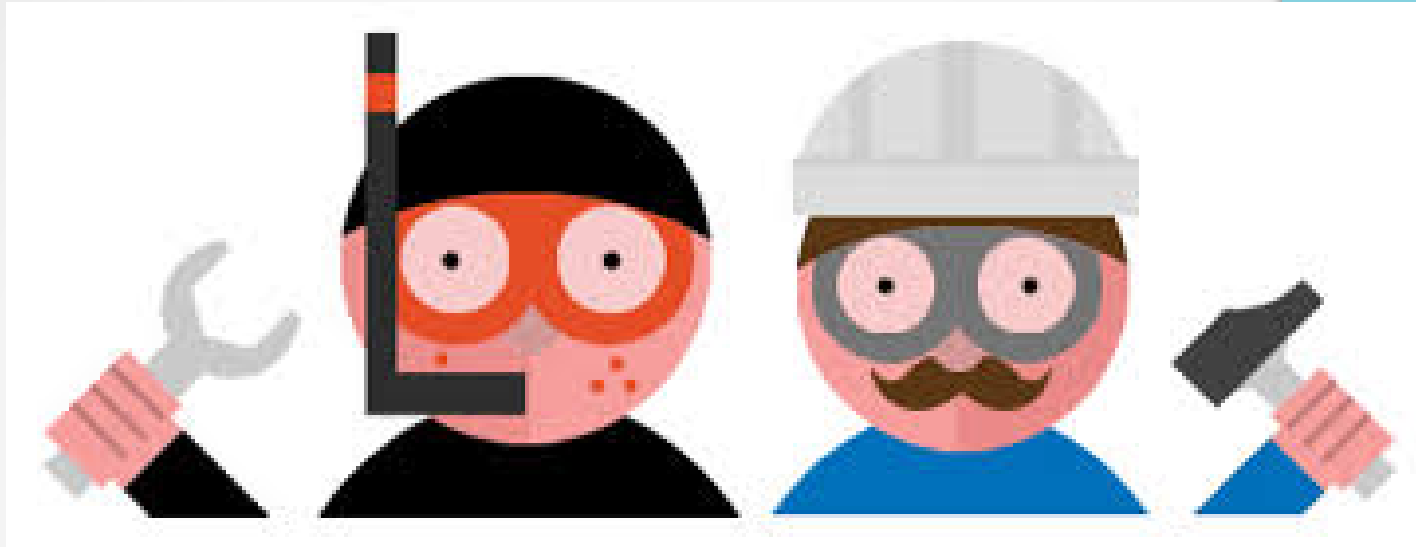
POST

/pet/{petId} Updates a pet in the store with form data

DELETE

/pet/{petId} Deletes a pet

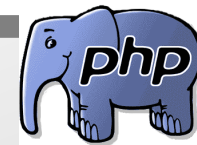
Construction des échanges



Front end:
Interroge le backend
ou lui confie des
données
injecte les réponses
dans le DOM html



Back end :
Offre un accès aux
ressources (les
données, les
notions)
souvent CRUD



Gestion JSON en Python

- json est une suite de caractères (une chaîne)
- L'humain en comprend la hiérarchie, mais pas la machine
- Il faut transformer en un objet type dictionnaire (compris par python, et manipulable par un être humain via des instructions en python) (c'est une structure informatique)
- C'est la sérialisation/dé-sérialisation
- json sert uniquement à l'échange (pour passer sur le réseau)

Json en python

```
>>> import json
>>> moi = {'nom': 'cherrier', 'prenom': 'sylvain'}
>>> print(moi)
{'nom': 'cherrier', 'prenom': 'sylvain'}
>>> type(moi)
<class 'dict'>
>>> monjson=json.dumps(moi)
>>> type(monjson)
<class 'str'>
>>> print(monjson)
{"nom": "cherrier", "prenom": "sylvain"}
>>> moi['role']='prof'
>>> print(moi)
{'nom': 'cherrier', 'prenom': 'sylvain', 'role': 'prof'}
>>> print(moi['prenom'])
sylvain
>>> monjson['role']='prof'
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> print(monjson['prenom'])
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: string indices must be integers, not 'str'
>>> neo=json.loads(monjson)
>>> type(neo)
<class 'dict'>
```

- **json.dumps** pour sérialiser un objet
- **json.loads** pour désérialiser
- *moi* est bien un dictionnaire (structure complexe)
- *monjson* est une simple chaîne, on ne peut plus la gérer comme un dictionnaire
- *neo* est bien un dictionnaire
- A NOTER : **Flask** propose directement des fonctions
- Flask : une route peut **return jsonify(objet)**
- Flask : une route pour récupérer le json reçu : **msg = request.json**

(voir plus loin l'exemple avec les headers du serveur d'absence)

Coté client

- Si c'est du javascript, on encode avant l'envoi avec `JSON.stringify(l'objet javascript)`...
- On décode la chaine *machaine* avec la promise *machaine.json().then(...)*

```
16     <script type="text/javascript">
17         const button = document.getElementById('bouton');
18         button.onclick = event => {
19             //par défaut, la page se recharge
20             event.preventDefault();
21             //version JSON
22             //on crée un objet Javascript
23             let params = {};
24             //on ajoute les éléments (notation [] si la clé est une
variable, sinon, notation pointée)
25             params.name = document.getElementById('nom').value;
26             params.age = document.getElementById('age').value;
27             fetch('exo2.php', {
28                 method: 'POST',
29                 body: JSON.stringify(params)
30             })
31             .then(response => response.json())
32             .then(data => {
33                 const message = "<p>Bonjour, tu es <b>" + data.nom +
34                 "</b> et tu as <i>" + data.age + "</i> ans.</p>";
35                 document.getElementById('message').innerHTML = message;
36             });
37     </script>
```

Header HTTP

- Meta informations associées à l'échange
- Contient par exemple le type de l'échange (*html, image, son, json, pdf, autre*), la taille, l'encodage, la date, un hashcode, la/les langues supportées
- Contient aussi les cookies, et autres informations de sécurité
- À la fois le client et le serveur s'en serve pour se mettre d'accord
- Normalement pas visible pour l'utilisateur lambda, mais utile pour nous
- (*examiner l'élément dans Firefox*)

Exemple d'échange HEADER

▼ En-têtes de la requête (438 o) Texte brut

- Accept: application/json, text/plain, */*
- Accept-Encoding: gzip, deflate, br
- Accept-Language: fr,en;q=0.7,en-US;q=0.3
- Authorization: Bearer RnlVRDBBWfplaWtVSzILbXJ2a2k1bXBBDWkZwMXVGSXA=
- Connection: keep-alive
- Host: ovh.bec3.com:81
- Origin: https://ovh.bec3.com:82
- Referer: https://ovh.bec3.com:82/sector/9
- User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:86.0) Gecko/20100101 Firefox/86.0

Requête pour la page sur la formation **IMAC (sector 9)**. On veut bien du json, ou du texte, en français de préférence. On voit ici mon *autorisation* (un numéro secret (*ici, un bearer*) que m'a donné le serveur, à présenter à chaque échange pour être reconnu... tant que je n'invalide pas ma session (déconnexion))

État **200 OK** ?
Version HTTP/1.1
Transfert 7,29 Ko (taille 6,83 Ko)
Politique de référent no-referrer-when-downgrade

▼ En-têtes de la réponse (480 o) Texte brut

- Access-Control-Allow-Credentials: true
- Access-Control-Allow-Headers: Content-Type, Authorization, X-Requested-With
- Access-Control-Allow-Methods: POST, GET, OPTIONS, PUT, DELETE
- Access-Control-Allow-Origin: *
- Access-Control-Max-Age: 86400
- Cache-Control: no-cache, private
- Connection: Keep-Alive
- Content-Length: 6990
- Content-Type: application/json
- Date: Mon, 15 Mar 2021 10:47:03 GMT
- Keep-Alive: timeout=5, max=100
- Server: Apache
- Vary: Authorization

La réponse est un 200, contenu en **json**, les access control sont fournis (CORS), taille de 6990 octets, le 15 mars.

Générer vos HEADER avec Flask

Le routeur permet d'agrémenter la réponse (outre son contenu) avec le code HTTP et des headers

If a tuple is returned the items in the tuple can provide extra information. Such tuples have to be in the form `(response, status)`, `(response, headers)`, or `(response, status, headers)`. The status value will override the status code and headers can be a list or dictionary of additional header values.

```
@app.route('/hello', methods=["POST"])
def hello():
    return jsonify({"foo": "bar"}), 201, {"Access-Control-Allow-Origin": "*"}
```

•

Header en Python

- Flask : On peut ajouter le code de retour http, et ajouter un ou plusieurs headers

```
from flask import Flask, request, jsonify, make_response
app = Flask(__name__)

@app.route('/headers', methods=['GET', 'POST'])
def headers():
    test={'message':'essai'}
    return jsonify(test), 201, {"Authorization": "bearer 1a124s5ef5786ad124654bc6fd621e"}

@app.route('/multiple', methods=['GET', 'POST'])
def multiple():
    test={'message':'essai'}
    resp = make_response(jsonify(test))
    resp.headers['Authorization'] = 'bearer c122149d5b332ef234ab542bc2424'
    resp.headers['X-Content-Type-Options'] = 'nosniff'
    resp.headers['X-Frame-Options'] = 'DENY'
    return resp, 201

if __name__ == '__main__':
    app.run(host='0.0.0.0', debug=True)
```


Type de réponse : Succès

200 OK

La requête a réussi. Le signification du succès peut varier selon la méthode HTTP :

GET : La ressource a été récupérée et est retransmise dans le corps du message.

HEAD : Les en-têtes d'entité sont dans le corps du message.

POST : La ressource décrivant le résultat de l'action est transmise dans le corps du message.

TRACE : Le corps du message contient le message de requête tel que reçu par le serveur

201 Created

La requête a réussi et une nouvelle ressource a été créée en guise de résultat. Il s'agit typiquement de la réponse envoyée après une requête PUT.

202 Accepted

La requête a été reçue mais n'a pas encore été traitée. C'est une réponse évasive, ce qui signifie qu'il n'y a aucun moyen en HTTP d'envoyer une réponse asynchrone ultérieure indiquant le résultat issu du traitement de la requête. Elle est destinée aux cas où un autre processus ou serveur gère la requête, et peut être utile pour faire du traitement par lots.

203 Non-Authoritative Information

Ce code de réponse signifie que l'ensemble de méta-informations renvoyé n'est pas exactement l'ensemble disponible sur le serveur d'origine, mais plutôt un ensemble collecté à partir d'une copie locale ou tierce. À l'exception de cette condition, une réponse 200 OK est préférable.

Extrait de
developer.mozilla.org

Réponse d'erreurs appli. : 4xx

400 Bad Request

Cette réponse indique que le serveur n'a pas pu comprendre la requête à cause d'une syntaxe invalide.

401 Unauthorized

Une identification est nécessaire pour obtenir la réponse demandée. Ceci est similaire au code 403, mais dans ce cas, l'identification est possible.

402 Payment Required

Ce code de réponse est réservé à une utilisation future. Le but initial justifiant la création de ce code était l'utilisation de systèmes de paiement numérique. Cependant, il n'est pas utilisé actuellement.

403 Forbidden

Le client n'a pas les droits d'accès au contenu, donc le serveur refuse de donner la véritable réponse.

404 Not Found

Le serveur n'a pas trouvé la ressource demandée. Ce code de réponse est principalement connu pour son apparition fréquente sur le web.

405 Method Not Allowed

La méthode de requête est connue du serveur mais a été désactivée et ne peut pas être utilisée.

Extrait de
developer.mozilla.org

Erreur du serveur : 5xx

500 Internal Server Error

Le serveur a rencontré une situation qu'il ne sait pas traiter.

501 Not Implemented

La méthode de requête n'est pas supportée par le serveur et ne peut pas être traitée. Les seules méthodes que les serveurs sont tenus de supporter (et donc pour lesquelles ils ne peuvent pas renvoyer ce code) sont `GET` et `HEAD`.

502 Bad Gateway

Cette réponse d'erreur signifie que le serveur, alors qu'il fonctionnait en tant que passerelle pour recevoir une réponse nécessaire pour traiter la requête, a reçu une réponse invalide.

503 Service Unavailable

Le serveur n'est pas prêt pour traiter la requête. Les causes les plus communes sont que le serveur est éteint pour maintenance ou qu'il est surchargé. Notez qu'avec cette réponse, une page ergonomique peut expliquer le problème. Ces réponses doivent être utilisées temporairement et le champ d'en-tête `Retry-After` doit, dans la mesure du possible, contenir une estimation de l'heure de reprise du service. Le webmestre doit aussi faire attention aux en-têtes de mise en cache qui sont envoyés avec cette réponse (qui ne doivent typiquement pas être mis en cache).

504 Gateway Timeout

Cette réponse d'erreur est renvoyée lorsque le serveur sert de passerelle et ne peut pas donner de réponse dans les temps.

Extrait de
developer.mozilla.org

PYTHON et la base de données

- Comme tout langage, Python peut dialoguer avec une BdD relationnelle
- On délègue à la base de données le STOCKAGE, la GESTION, et la PERSISTANCE des informations.
- Python s'occupe des traitements des données
- Un module permet d'interagir avec la base de données
- Il existe des outils très avancés capable de s'occuper de ces interactions pour vous (en python c'est Alchemy, en Java, c'est Hibernate, en PHP c'est Eloquent)



Python : BdD « à la main »

- Utilisation du module

```
import mysql.connector
```

- Création d'une connexion vers la base

```
mydb = mysql.connector.connect(  
    host="localhost",  
    user="sylvain",  
    password="-----",  
    database="test"  
)
```

- Création d'un « curseur »

```
mycursor = mydb.cursor()
```

Le curseur va être notre interlocuteur avec la BdD

Exemple d'utilisation du curseur

Méthodes :

Execute('requete SQL') :

va demander l'exécution de la requête

Commit()

valide l'opération (faite par défaut au « brouillon »).

L'opération inverse s'appelle le rollback

```
mycursor.execute('''create table etudiants (id int primary key,  
                                nom varchar(50)  
                                )''')  
mydb.commit()  
mycursor.execute('''insert into etudiants values  
                                (1,'Bob leponge'),  
                                (2,'Dora leporatrice'),  
                                (3,'Koro-sensei')''')  
mydb.commit()
```

Python BdD : le Select

- Le SELECT pose problème, à cause de sa réponse multiple. Il faut alors itérer sur le résultat.

```
# On recherche, et on affiche tout
mycursor.execute(''select * from etudiants'')
etuds = mycursor.fetchall()
print(etuds)

# on recharge, car on est arrivé au bout de la liste
print('-----')
mycursor.execute(''select * from etudiants'')
print(mycursor.fetchone())

# On continue la liste (on a déjà lu le premier)
for etud in mycursor:
    print(etud)

mycursor.close()
```


Python et le SELECT de la BdD

Plusieurs choses à remarquer :

le **FETCH** : C'est le truc à bien comprendre. **Fetch** (en base de données) va permettre de récupérer les éléments de réponse :

- ***fetchall()*** donne le tableau complet,
- ***fetchone()*** permet d'avoir le tuple suivant.
- On remarquera l'usage habituel qui est d'utiliser directement une boucle (qui appelle implicitement ***fetchone()*** jusqu'à plus soif)

FLASK : la continuité (session)

- HTTP est un serveur sans état (STATELESS)
- Chaque accès est « nouveau »
- On perd donc le suivi des échanges avec chaque client
- Il faut mettre en place un mécanisme pour assurer la continuité de l'échange
- Principe de « Sessions » (à base de cookies)

Session

- Il faut importer le module session de Flask
- Il se présente sous forme de dictionnaire
- Les informations seront stockées sur le client, qui présentera à chaque accès le contenu de ses cookies
- On interrogera/remplira/supprimera le contenu du dictionnaire coté Serveur
- Le cookie est chiffré, il faudra définir une clé de chiffrement

Suivi d'un utilisateur

- A vous d'inventer votre système
- Vous pouvez créer une sorte de « *jeton* »:
 - À la connexion (*mot de passe, login*) reconnaître votre utilisateur (*table user de la BdD*)
 - Vous stockez une valeur aléatoire dans la table (*id de session*) que vous stockez dans la session pour ce user
 - Vous vérifiez à chaque échange important que l'id dans la session correspond bien à un user de votre table (*voir headers plus haut*)
 - Vous effacez dans la table l'id de session à la déconnexions (et/ou toutes les nuits par exemple)