# Formation Module 6 Partie- 1

Ihab ABADI / UTOPIOS

# SOMMAIRE

# SOMMAIRE

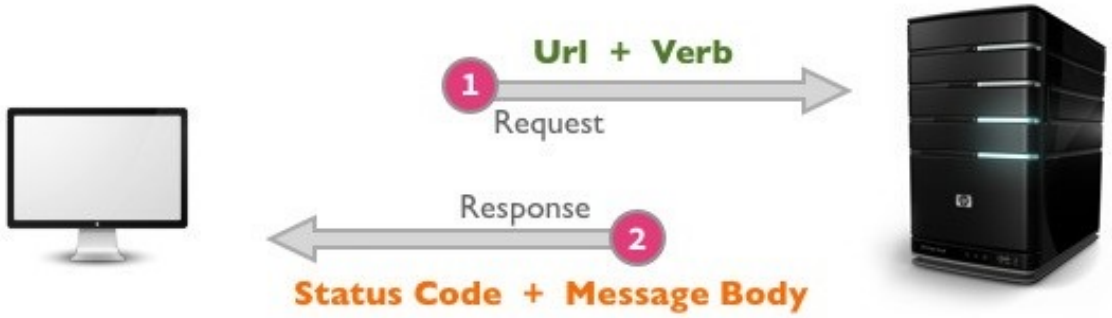
- Introduction à Flask Restful
- Les ressources d’une API REST.
- Les endPoints d’une API REST.
- Les blueprints flask.
- Les middlewares.
- La sécurisation d’une API REST.
- La documentation d’une API REST.
- Le déploiement d’une API REST.
- Alternatif au rest (gRPC).

# Communication client/serveur

## HTTP : les concepts

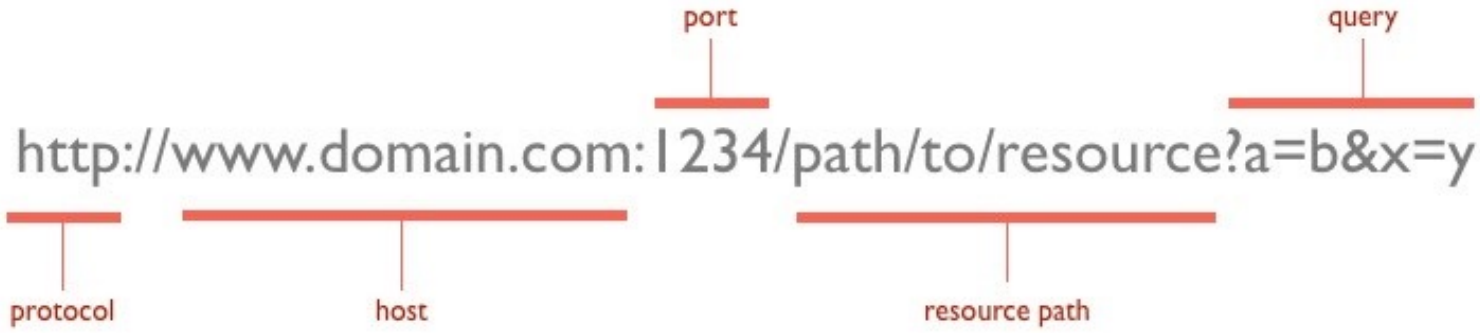
HTTP s'appuie sur 4 concepts fondamentaux :

- le binôme requête/réponse
- les URLs
- les verbes
- les codes de statut.



# Communication client/serveur

# HTTP : les URLs




Les urls sont à la base du fonctionnement de http car elles permettent d'identifier une ressource :

– **protocol** : le protocole utilisé (http, https, ftp, news, ssh...)
– **host** : nom de domaine identifiant le serveur (FQDN)
– **port** : le port utilisé (80 pour http, 443 pour https, 21 pour ftp)
– **ressource path** : identifiant de la ressource sur le serveur
– **query** : paramètres de la requête.

# Communication client/serveur HTTP les verbes

Les **verbes** permettent de manipuler les ressources identifiées par les URLs. Ceux principalement utilisés sont :

- **GET** : le client demande à lire une ressource existante sur le serveur
- **POST** : le client demande la création d'une nouvelle ressource sur le serveur
- **PUT** : le client demande la mise à jour d'une ressource déjà existante sur le serveur.
- **PATCH** : le client demande la mise à jour d'une partie d'une ressource déjà existante sur le serveur.
- **DELETE** : le client demande la suppression d'une ressource existante sur le serveur.

Ils sont invisibles pour l'utilisateur mais sont envoyés lors des échanges réseaux. Chaque requête est accompagnée d'un verbe pour indiquer l'action à effectuer sur la ressource ciblée.

```
GET http://welcome.com.intra/

GET http://www.monsite.fr/index.php
POST http://api.utopios.net/monappli/users/
```

# Communication client/serveur
# HTTP : les codes de statut

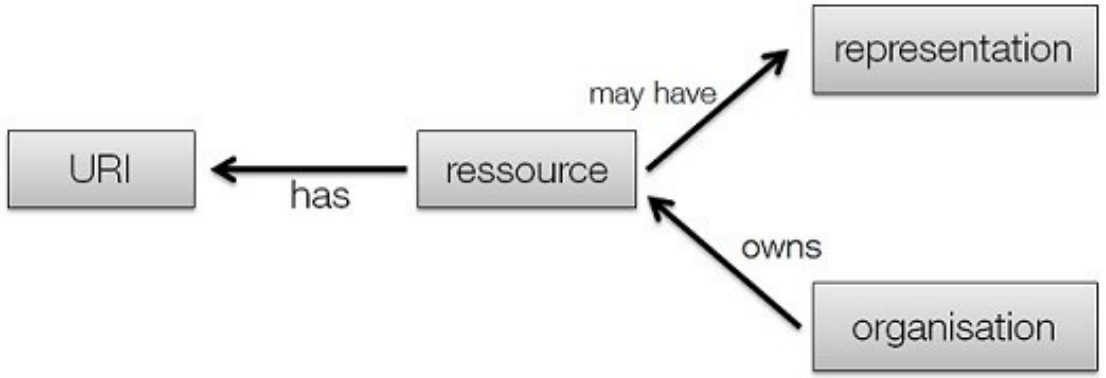
Chaque requête de la part d'un client reçoit une réponse de la part du serveur, comportant un **code de statut**, pour informer le client du bon déroulement ou non du traitement demandé.

Ces codes de statut sont rangés par **plages numériques** :

- **1xx** : message d'information provisoire
- **2xx** : requête reçue, interprétée, acceptée et traitée avec succès
- **3xx** : message indiquant qu'une action complémentaire de la part du client est nécessaire (exemple : redirection vers une autre url)
- **4xx** : erreur du serveur du fait des données en entrée envoyées par le client (exemple : authentification, autorisations, paramètres d'entrée)
- **5xx** : erreur du serveur du fait d'un motif interne au serveur (exemple : indisponibilité d'un composant du serveur, erreur inattendue).

# qu'est-ce qu'une API REST ?

Une **API REST (REpresentational State Transfer)** permet à une application d'exposer les services qu'elle offre aux autres applications (pourvues d'une IHM ou pas).




REST s'articule autour de la notion de **ressource** :

- une ressource représente n'importe quel concept (une commande, un client, un message...)
- une représentation est un document qui capture l'état actuel d'une ressource (au format Json, XML, pdf...)
- une ressource appartient à une organisation (une entreprise, un service public...)
- une ressource est accessible via une URI.

# 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.

# Qu'est-ce qu'une API REST ?

## HATEOAS

HATEOAS (Hypermedia As The Engine Of Application State) est un pilier de REST, permettant la **découvrabilité (discoverability)** de l'API à partir d'un point d'entrée unique.

Lorsque le serveur envoie sa réponse (la représentation d'une ressource) au client, il doit également ajouter les liens qui permettront au client de **modifier l'état de la ressource** en question ou de **naviguer** vers d'autres ressources.

Conséquences :

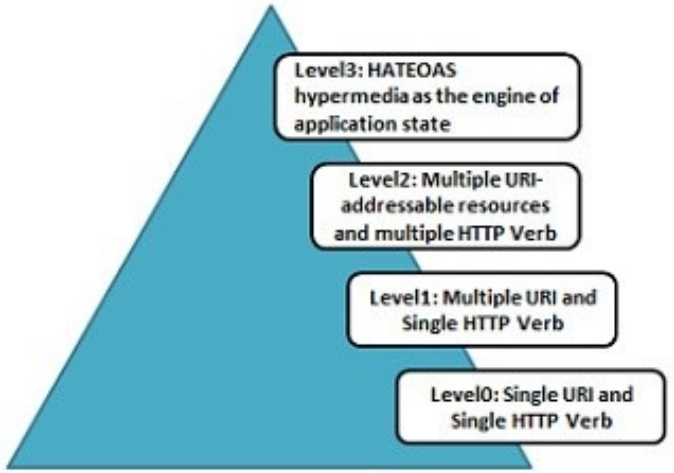
- –plus le message est pauvre (représentation sans hyperlien), plus le client doit être intelligent (connaître ce qu'il peut faire à partir de tel état)
- – plus le message est riche (avec hyperliens), moins le client doit être intelligent car il n'a qu'à suivre ce que lui indique le serveur.

Un site web respecte cette logique avec des liens envoyés par le serveur pour naviguer entre les pages (ressources), dans un format (HTML CSS, images...) lisible facilement par un humain. Entre machines, seules les informations métiers (au format Json par exemple) sont utiles,  avec les liens qui les unient.

# Qu'est-ce qu'une API REST ?

## modèle de maturité de Richardson

Le modèle de Richardson permet de mesurer le degré de maturité d'une API :




Richardson Maturity Model

# Qu'est-ce qu'une API REST ?

## modèle de maturité de Richardson

- **niveau 0** :
  - utilisation de *HTTP* servant de transport uniquement
  - verbe, URL et code retour uniques

  *exemple : webservices SOAP*

- **niveau 1** :
  - niveau 0 + *URLs différentes* pour identifier les ressources

  *exemple : navigation web*

- **niveau 2** :
  - niveau 1 + *verbes HTTP* pour manipuler les ressources + *codes retour* pertinents

  *exemple : APIs REST classiques*

- **niveau 3** :
  - niveau 2 + HATEOAS (liens)

  *vraie API REST idéalement*

# 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

```
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.

- 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.

# 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

- 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

- 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

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

- 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 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
```

# API: ToDoList

TP

**1/ identifier les ressources qui constitueront l'API**

La plupart du temps on retrouve :

- des ressources **entités** : concepts manipulés par l'API
- des ressources **composites** : agrégation de plusieurs ressources entités en une seule
- des ressources **collections** d'entités ou de composites.

Dans le cas d'une API de **ToDoList**, on peut imaginer avoir les ressources suivantes :

- entités : **ToDoList** et **ToDoItem**
- collections : **ToDoLists** et **ToDoItems**.

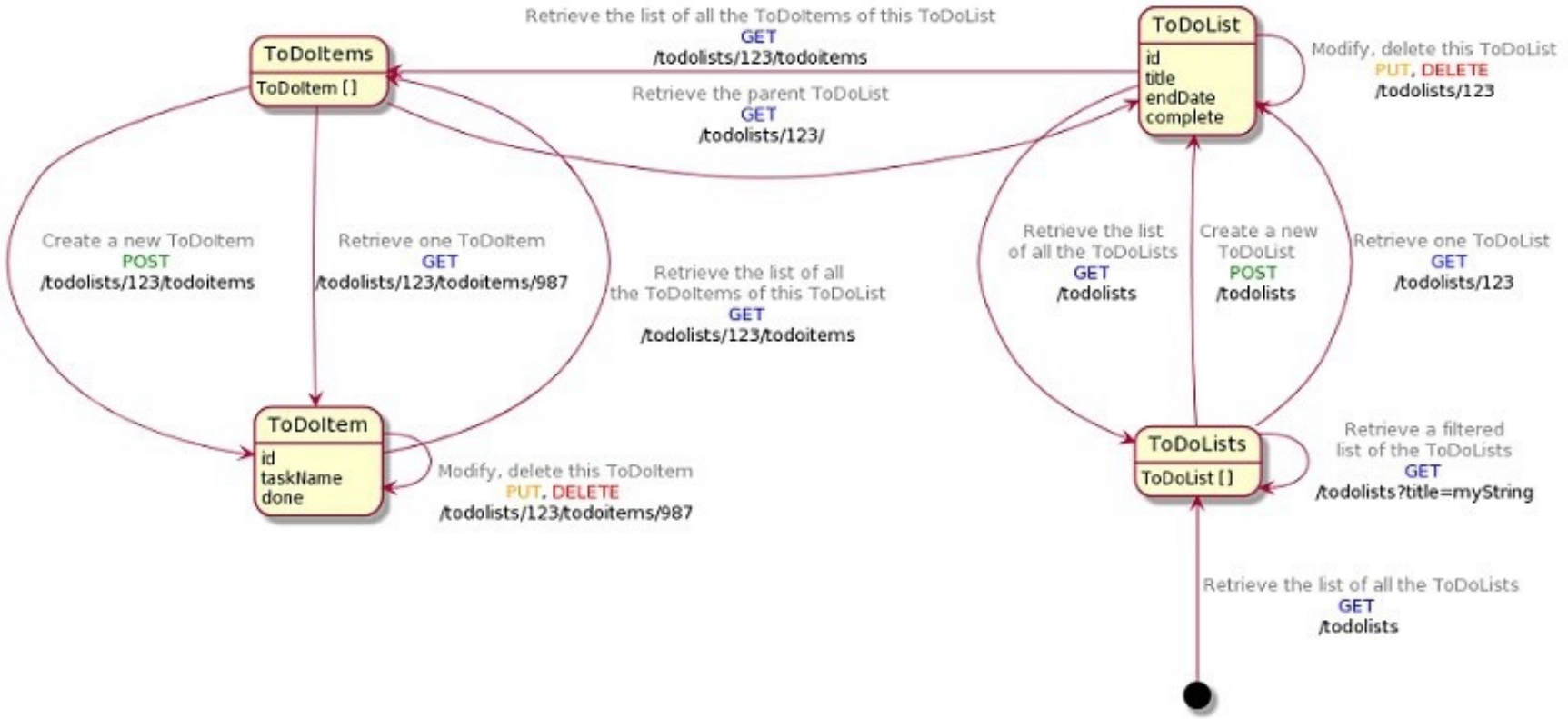
**2/ déterminer les URLs de ces ressources et les liens activables**

Pour chaque ressource, il faut déterminer :

- comment y accéder (leur URL)
- les actions autorisées :
  - changement d'état de la ressource
  - navigation vers une autre ressource.

# API: ToDoList

Pour représenter plus clairement les ressources et leur *cinématique*, on peut utiliser un **diagramme d'interactions** :



# Récupérer les données d'une requêtes

- Pour récupérer les données passer dans l'url : Nous pouvons utiliser la propriété args de request
- Pour récupérer les données passer par le form : Nous pouvons utiliser la propriété form de request
- Pour récupérer les données passer par json : Nous pouvons utiliser la propriété json de request

# API: ToDoList

TP

**3/ Complémenter l'api todolist en ajoutant les fonctionnalités suivantes:**

- **Ajouter une todolist dans une liste**
- **Supprimer une todolist de la liste**
- **Ajouter un todoitem dans une todolist**
- **Supprimer un toditem d'une todolist**