R5.A.05 – Programmation Avancée

Développement Fullstack avec Flask & Angular

Présentation

+Développeur Fullstack chez Akkodis

+Contact mail: t.louvet@iut.univ-paris8.fr

+Cours dispo sur Moodle après chaque séance

Déroulé du cours

+ 06/09 AM -> Développement Backend avec Flask
+ 06/09 PM -> Développement Frontend avec Angular
+ 08/11 PM -> Authentification, l'exemple avec JWT
+ 22/11 AM -> Flask pour le SSR, utilisation d'ORM
+ 29/11 AM -> Performance et sécurité applicative, documentation d'API
+ 06/12 AM -> Séance dédiée SAE
+ 13/12 AM -> TP Projet Complet

I. Développement Backend avec Flask

A. Le backend dans l'application

Application **multicouches**

Présentation
Frontend

Applicative
Backend / API

Persistance
BDD

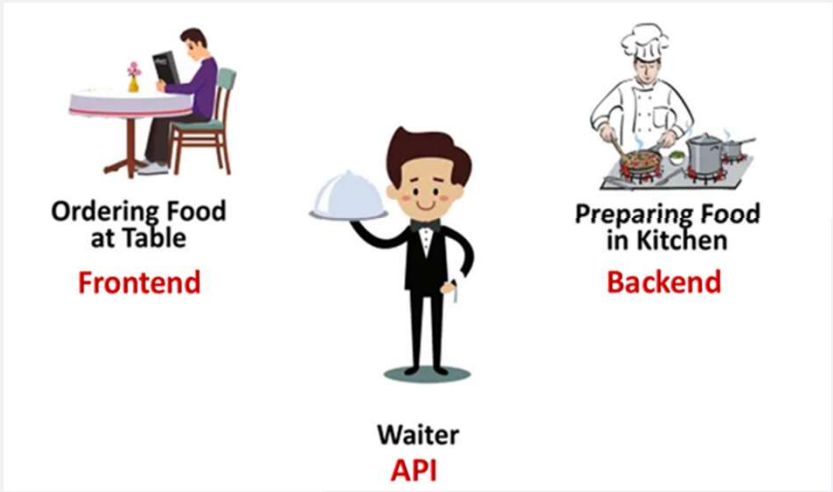
A. Le backend dans l'application

+ Couche applicative
+ Logique métier
+ Gestion des requêtes et des réponses
+ Authentification et autorisation
+ Intégration de services externes

A. Le backend dans l'application

+ Langages: Python / PHP / Ruby / Javascript (NodeJS) / Java / C# / GoLang ...
+ Quelques Frameworks :
  + Python : Flask / Django
  + PHP : Symfony / Laravel
  + Ruby : RoR
  + NodeJS : Express / NestJS
  + Java : Spring Boot
  + C# : .Net

B. Les API


Ordering Food
at Table
Frontend
Waiter
API
Preparing Food
in Kitchen
Backend

B. Les API

API = Application Programming Interface

Communication entre deux systèmes

Pas uniquement WEB

API OS -> Communication Applications vers OS

API SDK -> Accéder à des fonctions d'une bibliothèque logicielle

B. Les API - Rôles

Facilitent la communication

Favorisent l'encapsulation et l'abstraction

Permettent l'évolution indépendante d'un système

B. Les API WEB – REST vs SOAP

REST

- JSON principalement
- Créé pour des réponses rapides
- Facile à implémenter

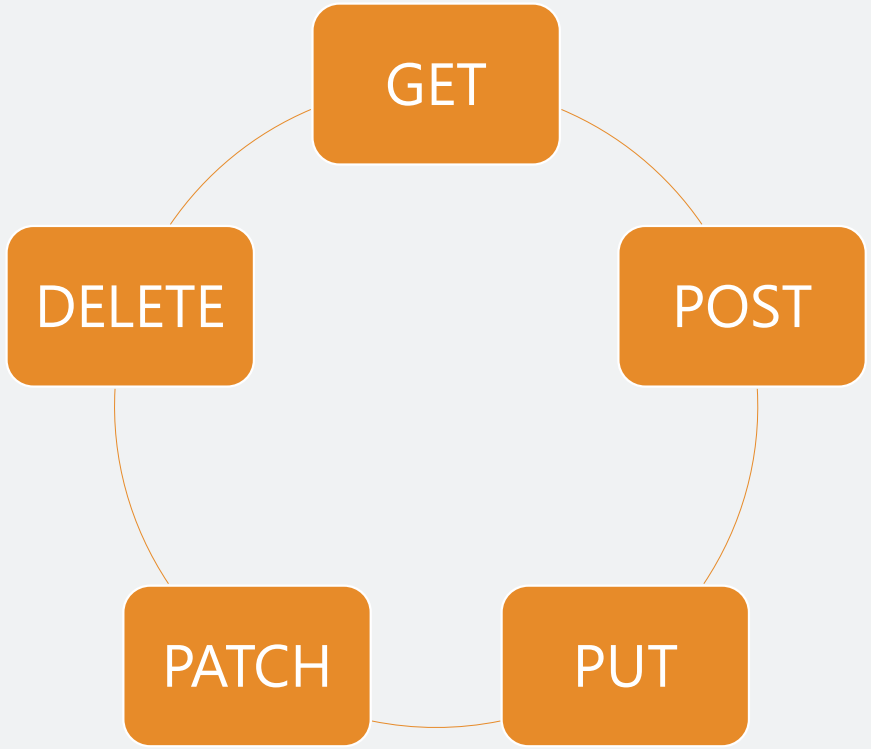
SOAP

- XML
- Protocole de messaging
- Plus de sécurité par défaut que REST

C. REST – Caractéristiques

- **Stateless**
- **Interface Uniforme**
- **Indépendance Client / Serveur**
- **Cachable**
- **Code à la demande**

C. REST – Méthodes principales



C. REST – Exemple CRUD

- **GET /products?offset=20&limit=20&type=cpu**
- **GET /products/:id**
- **POST /products -> { body: productToCreate }**
- **PUT /products/:id -> { body: productToUpdate }**
- **PATCH /products/:id -> { body: productToUpdate }**
- **DELETE /products/:id**

D. Quelques codes HTTP

Requête acceptée

- 200 OK – GET / PUT / PATCH / DELETE
- 201 Created – POST
- 204 No content – DELETE

Erreurs Requête Client

- 400 Bad Request
- 401 Unauthorized
- 403 Forbidden
- 404 Not Found
- 405 Method not Allowed
- 418 I'm a Teapot (pour briller en société uniquement)
- 429 Too Many Requests

Erreurs Serveur

- 500 Internal Server Error
- 503 Service Unavailable

E. Flask

+ Microframework Python pour le web
+ Création d'applications et d'API REST
+ Léger et minimaliste
+ Facile à prendre en main
+ Modulaire

E. Flask – Installation et création

- Créer un dossier, se positionner à l'intérieur et initier le venv:
  - **python –m venv .venv**
- Activer le venv:
  - Sous Windows: **.venv\Scripts\activate**
  - Sous Linux**: source .venv/bin/activate**
- Installer Flask
  - **pip install Flask**
- Par convention, Flask cherche un fichier **app.py**
- Un serveur se créé en 2 lignes
  - **from flask import Flask**
  - **app = Flask(__name__)**
- Lancer Flask run

app.py > ...

```
1  from flask import Flask
2
3  app = Flask(__name__)
```

```
 * Serving Flask app 'app'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
```

E. Flask – Debug Mode

+ Redémarrer le serveur à chaque changement = perte de temps
+ **flask run --debug**
+ Relance le serveur automatiquement à chaque sauvegarde de changement

Autre solution:

+ **pip install python-dotenv**
+ Créer un fichier .flaskenv
+ FLASK_DEBUG = 1

E. Flask – .flaskenv

- Via l'installation de **python-dotenv (pip install python-dotenv)**
- Fichier de configuration qui va contenir les secrets de l'application
- Evite de hardcoder les secrets dans le code -> Sécurité ++
- Ce fichier n'est **jamais** versionné, on utilisera un fichier d'exemple ou un README
- Fonctionne en clé=valeur

.flaskenv

```
1   FLASK_APP = app
2   FLASK_DEBUG = 0  # development mode (1) or production mode (0)
```

- FLASK_APP = le nom du fichier.py par défaut lorsqu'on lance **flask run**

E. Flask – Routes

+ Les routes s'ajoutent via des **décorateurs**
+ On peut définir une route et passer les méthodes associées (optionnel):

```
@app.route('/', methods=['GET'])
def hello_world():
    return 'Hello, World!'
```

+ Deux éléments: l'endpoint « / » -> Où récupérer la ressource : http://localhost:5000/
+ Les méthodes accéptées: uniquement GET

E. Flask – Routes

+ Endpoint différent (redémarrer le serveur) http://localhost:5000/hello

```
@app.route('/hello')
def hello_world():
    return 'Hello, World!'
```

+ Erreur 404 sur http://localhost:5000/ car plus utilisé
+ Préférez l'utilisation de @app.[methode] – cf slide suivante

E. Flask – Routes

+ Route multiméthodes
+ Gestion de la logique par méthode dans la même fonction
+ Code 405 pour les méthodes non prises en charge
+ Lourd à lire
+ Difficile à faire évoluer

app.py > ...

```python
from flask import Flask, request

app = Flask(__name__)

@app.route('/hello/<string:id>', methods=['GET', "PUT", "PATCH", "DELETE"])
def hello_world():
    if  request.method == 'GET':
        return 'Hello, World!'
    elif request.method == 'PUT':
        # quelque chose
        return 'PUT'
    elif request.method == 'PATCH':
        # quelque chose
        return 'PATCH'
    elif request.method == 'DELETE':
        # quelque chose
        return 'DELETE'
    else:
        return 'Méthode non autorisée', 405
```

E. Flask – Routes

+ Solution – Séparation des routes par méthode
+ Chaque méthode possède sa logique indépendante
+ Chaque méthode va gérer ses autorisations et validations
+ Plus facile à lire, à faire évoluer
+ Single Responsibility Principle

```python
@app.get('/hello/<string:id>')
def find_one(id):
  # logique de recherche
  return 'La ressource demandée'

@app.put('/hello/<string:id>')
def update_full(id):
    # logique de recherche
    # mise à jour
  return 'La ressource mise à jour'

@app.patch('/hello/<string:id>')
def update_partial(id):
    # logique de recherche
    # mise à jour
  return 'La ressource mise à jour'

@app.delete('/hello/<string:id>')
def delete_one(id):
    # logique de recherche
    # suppression
  return {}, 204
```

E. Flask – Requêtes - Params

+ Passage de **paramètre** -> /mon-endpoint/**:id**
+ Types acceptés: string / int / uuid (type d'id) / path (string avec des « / »)
+ **:id** est un paramètre

```
@app.get('/mon-endpoint/<string:id>')
def get_one(id):
  return 'La ressource demandée'
```

+ Les paramètres sont cumulables -> /store/:storeID/products/:productID

```
@app.get('/store/<string:id>/product/<string:product_id>')
def get_one(id, product_id):
  return 'La ressource demandée'
```

Attention à ne pas avoir deux routes qui ont le même nom de fonction, Flask ne l'autorisera pas

E. Flask – Requêtes - Queries

/endpoint?cle1=v1&cle2=v2

« ? » début
« & » nouveau paramètre

Filtrer & **trier** des ressources

Système clé = valeur

Optionnel, /products peut fonctionner sans que je passe de query

```python
@app.get('/products')
def get_products():
    (category, limit, offset) = request.args.get('category'), request.args.get('limit'), request.args.get('offset')
    print(category, limit, offset)
    return 'La ressource demandée'
```

E. Flask – Requêtes - Body

+ Les **requêtes** POST / PUT et PATCH contiennent un **body**
+ Body = ensemble des données nécessaires au traitement de la requête
+ Extraction via **request.get_json()**
+ Dictionnaire si objet envoyé, liste si tableau
+ Si fichiers, voir ***request.files***

```python
@product_bp.post('/products')
def create_product():
  body = request.get_json()
  return body, 201
```

E. Flask – Réponses

+ Différents types de réponse: nombre, string, objet, liste
+ Standardiser pour mieux exploiter la réponse en frontend
+ Si string ou int, wrapper dans un objet avec clé=valeur
+ Liste ou Dict, renvoyer tels quels
+ Le must, utiliser ***jsonify*** fourni par flask

```python
@app.post('/sum')
def sum():
  body = request.get_json()
  total = body['a'] + body['b']
  return {'sum': total}


@app.get('/list')
def find_all():
  return [{
    'id': 1,
    'name': 'Product 1'
  }, {
    'id': 2,
    'name': 'Product 2'
  }]

@app.get('/list/<int:id>')
def find_one(id):
  return {
    'id': id,
    'name': 'Product 1'
  }
```

E. Flask – Abort

- Lorsqu'on veut renvoyer vers une page d'erreur par défaut
- Importée depuis le package flask
- Utilisation: **abort(code)**
- Peut-être lancée en cours de requête
  - Refuser de continuer le traitement car l'utilisateur n'est pas autorisé
  - Le payload n'est pas bon
  - Etc.

```python
@app.get('/error')
def error():
  abort(500)
```

```python
@app.post('/login')
def login():
  body = request.get_json()
  if body['email'] != "admin@myapp.com":
    abort(401)

  if body['password'] != "admin":
    abort(401)

  return {'message': 'Vous êtes connecté'}
```

E. Flask – CORS

- CORS = Cross Origin Ressource Sharing
  - Mécanisme de sécurité web, empêche les requêtes entre domaines différents
- Package Flask – **flask-cors**
  - **pip install flask-cors**
  - Configuration 1 ligne
  - Configuration plus complète
  - Configuration par route

```
5   # Configuration avancée des CORS
6   CORS(app)
```

```
# Configuration avancée des CORS
CORS(app, resources={
    r"/*": {  # Cible toutes les routes
        "origins": ["http://localhost:4200"],  # Autorise uniquement l'origine http://localhost:4200
        "methods": ["GET", "POST", "PUT", "PATCH", "DELETE", "OPTIONS"],  # Méthodes HTTP autorisées
        "allow_headers": ["Content-Type", "Authorization"],  # En-têtes autorisés
    }
})
```

```
@cross_origin(origins=["http://localhost:4200"])
@app.route('/hello')
def hello_world():
  return 'Hello, World!'
```

F. Architecture - Controller

- Groupe de **Routes** pour un domaine de l'application (ex: « /products », « /auth »)
- On y trouve du code lié au **Framework** (Flask**)**
- Avec Flask, on utilise les **Blueprints**
- Reçoit des **Requêtes**
- Assure l'autorisation et la validation et le mapping d'input, puis passe le relai au service
- Un **Controller** n'assure aucune logique métier / persistance
- Renvoie des **Réponses**
- Intérêt – Penser au S et O de SOLID
  - Single Responsibility Principle: une seule raison de changer -> Nouvelle route
  - Open Closed Principle: La logique métier est étendue grâce à de nouveaux services

F. Architecture – Controller Exemple

```python
from flask import Blueprint, request, abort, Response, jsonify
from services.store_service import get_stores, create_store, find_one_store,update_whole_store, update_partial_store_items, remove_store

store_bp = Blueprint('store', __name__)
not_found_message = {"message": "Pas de store correspondant"}

@store_bp.get('/stores')
def find_many():
    stores = get_stores()
    return stores

@store_bp.get('/stores/<uuid:id>')
def find_one(id):
    store = find_one_store(id)
    if store:
        return store,200

    return not_found_message, 404

@store_bp.post('/stores')
def create():
    body = request.get_json()
    new_store = create_store(body)
    return new_store, 201
```

Pensez à importer store_bp dans votre fichier principal
Pour enregistrer les routes -> app.register_blueprint(store_bp)

F. Architecture – Controller Exemple

```python
@store_bp.put('/stores/<uuid:id>')
def update(id):
    body = request.get_json()
    store = update_whole_store(id, body)
    if store:
        return store,200

    return not_found_message, 404

@store_bp.patch('/stores/<uuid:id>/items')
def update_items(id):
    body = request.get_json()
    store = update_partial_store_items(id, body['items'])
    if store:
        return store, 200

    return not_found_message, 404

@store_bp.delete('/stores/<uuid:id>')
def delete(id):
    store = remove_store(id)
    if store:
        return {}, 204

    return not_found_message, 404
```

F. Architecture – Service

+ Appelé par un **Controller**
+ Gère la logique métier, fait partie du **Domaine**
+ Pas de code framework, ici juste du **Python** standard
+ Interagit avec la couche **Données** via des **Repositories (vu dans une future séance)**
+ Interagit uniquement avec des **Entités** du **Domaine**
+ Si logique très complexe, possible de séparer chaque action en **UseCase**

F. Architecture – Service

services > store_service.py > find_one_store

```python
from db.stores import stores
import uuid

def get_stores():
    return stores

def find_one_store(id):
    for store in stores:
        if store['id'] == id:
            return store
    return None

def create_store(body):
    new_store = {
        'id': uuid.uuid4(),
        'name': body['name'],
        'items': []
    }
    stores.append(new_store)
    return new_store
```

```python
def update_whole_store(id, body):
    for idx, store in enumerate(stores):
        if store['id'] == id:
            stores[idx] = body
            stores[idx]['id'] = id
            return stores[idx]
    return None

def update_partial_store_items(id, items):
    for idx, store in enumerate(stores):
        if store['id'] == id:
            stores[idx]['items'] = items
            return stores[idx]
    return None

def remove_store(id):
    for idx, store in enumerate(stores):
        if store['id'] == id:
            del stores[idx]
            return stores
    return None
```

F. Architecture – DTO vs Entité Métier vs Entité DB

DTO = Data Transfer Object

Donnée simple, spécifique à l'API

Pas de logique

Validation simple et principalement technique

Courte durée de vie, transfert uniquement – vit dans le **Controller**

Entité = Objet métier complexe

Donnée fondamentale dans le domaine de l'application

Inclut la logique métier et les relations vers d'autres objets métiers

Validation avancée de la logique métier

Durée de vie plus longue, vit dans le **Controller / Service / Repository**

Entité DB = Entité Persistante

Donnée simple, types primitifs

Pas de logique

Courte durée de vie, transfert uniquement – vit dans le **Repository**

F. Architecture – Validation d'input DTO

+ POST, PUT et PATCH ont un **body**
+ Le **Controlleur** doit **TOUJOURS vérifier ces données**, ne faites **JAMAIS confiance** aux données entrantes.
+ Risque de sécurité applicative – Injections SQL notamment
+ Avec Flask, plusieurs options:
  + flask_expects_json
  + marshmallow

F. Architecture – Validation d'input DTO

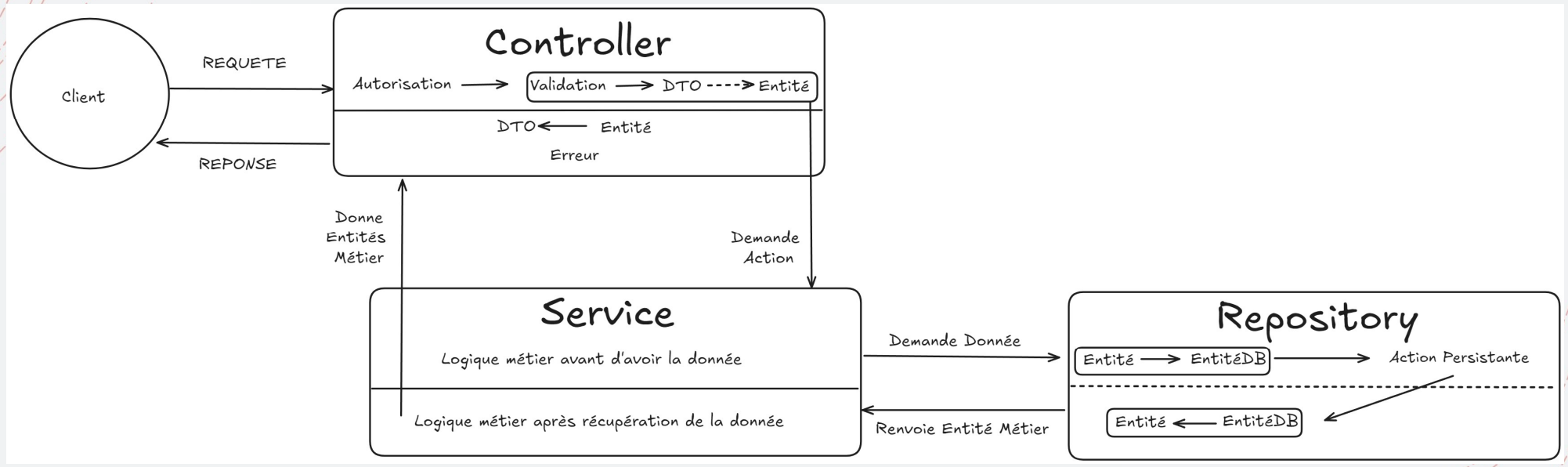
- flaks_expects_json:
  - Basé sur des dictionnaires
  - Définition des propriétés
  - Définition de champs obligatoires
  - Choix de renvoyer une erreur si trop de champs
  - Types acceptés:
    - object / array / string / integer / number / null / boolean /
  - Erreur 400 automatique si rejet du DTO
  - **Le décorateur doit être placé sous la route**

dto > user-schema.py > login_schema

```python
login_schema = {
    'type': 'object',
    'properties': {
        'email': {'type': 'string', 'format': 'email'},
        'password': {'type': 'string'}
    },
    'required': ['email', 'password'], # si email ou password manquants, erreur
    'additionalProperties': False # si autre propriété, erreur
}
```

```python
@auth_bp.post('/auth/login')
@expects_json(login_schema)
def login():
    return "Logged in successfully", 200
```

F. Architecture – Schéma Récapitulatif



Ressources & Bibliographie

+ Documentation Flask: https://flask.palletsprojects.com/en/3.0.x/
+ Liste des codes HTTP: https://developer.mozilla.org/fr/docs/Web/HTTP/Status
+ **Higginbotham**, James « Principles of WEB API Design ». Pearson – Chapitre 7 en particulier « REST-based API Design »
+ Pour aller plus loin dans l'architecture:
    + **Martin,** Robert C. « Clean Architecture ». Pearson (Existe traduit en français)