API REST & Microservices
(SpringBoot + Java + Flask (Python))

Objectif général

Ce TP vise à comprendre les fondements des **API REST** et leur mise en œuvre dans deux environnements distincts : **SpringBoot (Java 17)** et **Flask (Python 3.13)**.

L'objectif est de maîtriser les concepts communs aux APIs REST : exposition de routes HTTP, manipulation et sérialisation, conception de modèles (entités), opérations CRUD, persistance en base de données, et enfin structuration d'un projet backend.

Une fois ces principes compris, l'implémentation dans n'importe quel langage devient largement intuitive.

Avant de commencer, il faut savoir que SpringBoot n'a plus de version 2.x disponible et donc ne supporte plus la version Java 8. C'est donc la version Java 17 qui sera utilisé durant ce TP. Autre point : seule la version MySQL a pu aboutir car je n'ai pas réussi à faire fonctionner H2.

Enfin il faut noté que pour la partie 2, le **git@github.com:samiryoucef/apirestpython.git** ne fonctionnait pas et donc tout a été repris du cours donné en classe sur Flask en Python, ainsi d'une aide externe via StackOverflow et ChatGPT.

PARTIE 1 — API REST AVEC SPRINGBOOT

1.1 Création du projet SpringBoot

SpringBoot permet la génération rapide d'un projet Java basé sur Maven. Il propose l'inclusion automatique d'un serveur Tomcat embarqué, un point d'entrée unique (*Application.java*), et un JAR exécutable généré dans *target/* . Le fichier *application.properties* centralise la configuration (port, base H2, etc.), pour simplifier la tâche au développeur.

1.2 Premiers Services REST

Pour comprendre, on va créer une classe **MyApi**, annotée **@RestController** pour la transformer en API REST, puis on expose des routes REST via par exemple : @GetMapping("/bonjour"), avant une méthode pour un mapping de type HTTP/GET sur la méthode en passant par **http://localhost/nomdemonmapping**. SpringBoot convertit automatiquement les objets renvoyés en **JSON**, sans besoin de configuration supplémentaire.

1.3 Ajout de la classe Etudiant

Pour le TP, on va créer la classe **Etudiant** qui contient : un identifiant, un nom, une moyenne, des constructeurs, et enfin des getters/setters. On va ensuite créer une méthode getEtudiant dans notre classe MyApi, avec un mapping, qui renvoie un objet Etudiant, automatiquement traduit en JSON.

1.4 Tests avec Postman

Postman permet d'envoyer des requêtes GET, POST, PUT, et DELETE, de tester les paramètres, et d'afficher proprement les réponses JSON. Cela nous permet de simplementvalider les différentes routes REST sans passer par le navigateur, tout en sauvegardant les manipulation pour plus tard.

1.5 Simulation d'une base avec ArrayList

Pour simuler une base avant JPA, on peut utiliser une `ArrayList<Etudiant>` qui va stocker les données localement pour le test. Il faut également implémenté les routes : **GET** (liste complète), **POST** (ajout), **PUT** (modification), et enfin **DELETE** (suppression). Cela introduit les **opérations CRUD**.

1.6 Passage à une vraie base de données : H2 + JPA

On peut rajouter des dépendances manquantes soit en recréant le projet, soit en passant par le fichier pom.xml, dans la partie <**dependencies**>. Pour connaître le contenu à rajouter pour nos dépendances, il suffit d'aller sur le site Maven Repository pour récupérer le code à copier. On va donc ajouter : H2 ou MySQL (base mémoire) et Spring Data JPA (ORM).

On va également réorganiser le projet 3 packages : **entities**, **repository**, et **web**.

On va ensuite créer différent fichier suivant le modèle MVC (Model View Controller) :
- Un modèle (entité) **Adherent** annotée avec @Entity avant la classe et @Id avant le paramètre Id. Pour que l'id soit incrémenter automatiquement, on peut utiliser @GeneratedValue.
- Une interface **AdherentRepository** étend **JpaRepository** pour fournir toutes les opérations CRUD sans code supplémentaire.
- Un **CommandLineRunner** dans notre Application, qui permet de remplir la base au démarrage avec quelques données.

Il faut également rajouter les propriétés de la base de données utilisés via les différents paramètres de **spring.datasource** (comme l'url de la BD, le username...)

PARTIE 2 — API REST AVEC PYTHON & FLASK

La seconde partie consiste à **reproduire les mêmes fonctionnalités** mais avec **Flask** en Python.

2.1 Présentation de Flask (et Flask-RESTful)

Flask est un micro-framework minimaliste, qui fournit : un serveur HTTP intégré, une syntaxe simple pour définir des routes et un fonctionnement très proche de SpringBoot en termes conceptuels.Pour structurer une API REST, on peut utiliser **Flask simple** ou **Flask-RESTful** (optionnel, facilite la création de services REST)

2.2 API Flask basique

On initialise donc une api Flask. Tout comme SpringBoot, Flask convertit automatiquement les dictionnaires Python en JSON via jsonify(). L'instruction @GetMapping() équivaut à @app.get() en Flask.

2.3 Equivalent à JPA : SQLAlchemy

Pour représenter les données comme les entités, on peut utiliser **SQLAlchemy** comme ORM. On l'initialise via db = SQLAlchemy(app).

2.4 Base de données

Une configuration simple en MySQL pour Python via SQLAlchemy :
SQLALCHEMY_DATABASE_URI = 'mysql+pymysql://username:password@localhost:3306/nom_de_la_bd'

2.5 CRUD complet en Flask

@app.get("/etudiants") # Liste des étudiants

@app.get("/etudiants/<int:id>") # Étudiant précis

@app.post("/etudiants") # Ajouter un étudiant

@app.put("/etudiants/<int:id>") # Modifier un étudiant

@app.delete("/etudiants/<int:id>") # Supprimer un étudiant

2.6 Organisation du projet Flask

Structure utilisé pour cette 2ème partie de TP :

```
project/
|── app.py # L'application Flask et la connection avec la BD
|── config.py # Les paramètres comme l'URL de la BD
|── extensions.py # SQLAlchemy(), appelé 1 seule fois pour tout le monde.
|── models/
|     └── etudiant.py # La classe Étudiant et son implémentation
|── resources/
|     └── etudiant_api.py # Les routes de l'API REST
```

2.7 Ce qu'il faut retenir

Les fondamentaux REST sont les mêmes entre SpringBoot et Flask : une route → une méthode Python ou Java, chaque méthode renvoie du JSON, opérations CRUD, un modèle = une classe avec des attributs, l'ORM gère l'accès à la base, le framework lance un serveur HTTP.

Conclusion

Ce TP permet de comprendre que malgré des outils différents (SpringBoot vs Flask), la logique REST reste identique : routes, modèles, JSON, base de données, CRUD, structure du backend.
SpringBoot offre un cadre complet et productif.
Flask laisse une grande liberté et convient parfaitement aux microservices Python.
La maîtrise de ces concepts rend possible la création d'APIs professionnelles dans n'importe quel langage ou framework.