Université Sorbonne Paris Nord

École d'Ingénieurs Sup Galilée

API REST et Microservices

Rapport de Travaux Pratiques – TP n°1

Matière : Du développement au déploiement d'applications web

Étudiant : BENHAMMADA Ahmed Amine

Promotion : INGINFOA2 (École d'Ingénieurs Sup Galilée)

Encadrant : Samir Youcef

Date de rendu : 23 novembre 2025

Table des matières

1 Introduction et Objectifs

Ce TP a pour but de guider la réalisation d'une **API RESTful** de gestion d'étudiants en utilisant l'architecture des **microservices**. L'objectif principal est double :

- Maîtriser la création de microservices avec **Java Spring Boot**.
- Reproduire cette architecture avec **Python Flask** pour démontrer l'interopérabilité des concepts.

L'environnement technique utilisé est **Java 21 (Spring Boot)** et **Python**, avec **MySQL** comme base de données de persistance.

1.1 Principes REST et Microservices

Une API REST (Representational State Transfer) repose sur plusieurs principes fondamentaux :

- **Sans état (Stateless)** : Chaque requête est indépendante et contient toutes les informations nécessaires.
- **Ressources identifiées par URL** : Chaque ressource est accessible via une URL unique.
- **Méthodes HTTP standardisées** : Utilisation cohérente des méthodes `GET`, `POST`, `PUT`, `DELETE` pour les opérations **CRUD** (Create, Read, Update, Delete).
- **Format JSON** : Les données sont échangées au format JSON, léger et universel.

Les **microservices** sont des applications REST autonomes et légères, conçues pour être déployées indépendamment, ce que Spring Boot facilite grâce à son environnement d'exécution et sa configuration automatique.

2 Partie I : Mise en place de l'API Java (Simulation)

Cette première phase consiste à simuler le stockage des données via une **liste statique en mémoire** sans dépendance à une base de données, pour tester rapidement l'API.

2.1 Le Modèle de Données (POJO)

La classe `Etudiant` est un **POJO** (Plain Old Java Object) qui respecte l'encapsulation et fournit les accesseurs (Getters/Setters) pour permettre la conversion en JSON.

2.1.1 Extrait du code Etudiant.java

```
package com.example.spring1;

public class Etudiant {
    private int id;
    private String nom;
    private double moyenne;

    public Etudiant() {
    }

    public Etudiant(int id, String nom, double moyenne) {
        this.id = id;
        this.nom = nom;
        this.moyenne = moyenne;
    }

    public int getId() {
        return id;
    }
// ... Getters et Setters
}
```

2.2 Le Contrôleur et la Liste Statique

La classe `MyApi` est annotée avec `@RestController`. Une `ArrayList` statique est utilisée pour simuler une base de données en mémoire.

2.2.1 Extrait du code MyApi.java

```
package com.example.spring1;

import java.util.ArrayList;
import org.springframework.web.bind.annotation.*;
```

```
@RestController
public class MyApi {
    public static ArrayList<Etudiant> liste = new ArrayList<>();

    static {
        liste.add(new Etudiant(1, "AMINE", 17));
        liste.add(new Etudiant(2, "BOUCHER", 12));
        liste.add(new Etudiant(3, "DUPONT", 14));
        liste.add(new Etudiant(4, "MARTIN", 16));
    }

    @GetMapping(value = "/liste")
    public ArrayList<Etudiant> getAllEtudiant() {
        return (ArrayList<Etudiant>) liste;
    }

    @PostMapping(value = "/ajouterEtudiant")
    public Etudiant addEtudiant(Etudiant e) {
        liste.add(e);
        return e;
    }

    // ... Autres méthodes CRUD (GET, PUT, DELETE)
}
```

3 Partie II : Persistance des Données (JPA & H2/MySQL)

Cette phase remplace le stockage en mémoire par une gestion de la persistance via **Spring Data JPA**.

3.1 Adaptation du Modèle (Entité JPA)

Pour la persistance, la classe devient une entité JPA. L'exemple `Adherent.java` illustre cette transformation en utilisant les annotations `jakarta.persistence.*`.

3.1.1 Extrait du code Adherent.java (Exemple d'Entité JPA)

```
package com.polytech.h2demo.entities;

import jakarta.persistence.Entity;
```

```
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;

@Entity
public class Adherent {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String nom;
    private String ville;
    private int age;
// ... Constructeurs et accesseurs (Getters/Setters)
}
```

3.2 Création du Repository

L'interface `AdherentRepository` hérite de `JpaRepository`, permettant à Spring Data JPA de générer automatiquement les implémentations des méthodes CRUD.

3.2.1 Extrait du code AdherentRepository.java

```
package com.polytech.h2demo.repository;

import com.polytech.h2demo.entities.Adherent;
import org.springframework.data.jpa.repository.JpaRepository;

public interface AdherentRepository extends JpaRepository<Adherent, Long> {

}
```

3.3 Configuration des Bases de Données

Le fichier de configuration (`application.properties`) permet de basculer facilement entre une base de données en mémoire (**H2**) pour le développement ou une base relationnelle (**MySQL**) pour la production.

- **Configuration H2** : Active la base en mémoire et la console web H2 pour la visualisation.
- **Configuration MySQL** : Définit les paramètres de connexion et la stratégie Hibernate (`ddl-auto=update` ou `create`).

4 Partie III : Reproduction en Python (Flask)

L'API a été reproduite en Python en deux étapes : simulation en mémoire, puis intégration d'une base de données via SQLAlchemy.

4.1 Simulation en Mémoire avec Flask

Le code Flask utilise une liste Python simple pour simuler le stockage, reproduisant les opérations CRUD à l'aide des décorateurs de routes (`@app.route`) et des méthodes HTTP (`methods=['GET', 'POST', 'PUT', 'DELETE']`).

4.1.1 Code Flask (Simulation en mémoire)

```
from flask import Flask, jsonify, request

# Créer l'application flask
app = Flask(__name__)

# Sauvegarder les étudiants dans une liste :
students = [
    {"id":1, "prenom":"Samir", "age":31},
    {"id":2, "prenom":"Safa", "age":22},
]

# Définir la racine de l'API et autres routes
@app.route('/')
def home():
    return "C'est cool REST !"

@app.route('/students', methods=['GET'])
def get_students():
    return jsonify(students)

# Ajouter un étudiant, c'est une nouvelle ressource (via POST) :
@app.route('/students', methods=['POST'])
def add_student():
    new_student = request.get_json() # récupérer l'object
    new_student['id'] = len(students)+1 # Assigner un ID automatique
    students.append(new_student)
    return jsonify(new_student), 201
```

```
# Afficher un étudiant sachant son identifiant :
@app.route('/students/<int:id>', methods=['GET'])
def get_student(id):
    student = next((s for s in students if s['id']==id), None)
    if student:
        return jsonify(student)
    return jsonify({"erreur": "L'étudiant n'est pas trouvé"}), 404

# Mise à jour d'un étudiant (PUT)
@app.route('/students/<int:id>', methods=['PUT'])
def update_student(id):
    student = next((s for s in students if s['id']==id), None)
    if not student:
        return jsonify({"erreur": "not exist"}), 404

    data = request.get_json()
    student.update(data)
    return jsonify(student)

# Suppression d'un étudiant (DELETE)
@app.route('/students/<int:id>', methods=['DELETE'])
def delete_student(id):
    global students
    students = [s for s in students if s['id'] != id]
    return jsonify({"message": "L'étudiant a été supprimé"}), 200

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

4.2 Persistance des Données avec Flask-SQLAlchemy

Pour la seconde partie (persistance), Flask utilise **SQLAlchemy** comme ORM (équivalent à Hibernate/JPA en Java) pour se connecter à MySQL (`db_etudiants`).

4.2.1 Extrait du Code Flask (Persistance MySQL)

```
from flask import Flask, jsonify, request
from flask_sqlalchemy import SQLAlchemy
# ... autres imports

app = Flask (__name__)
```

```
# Connexion à la même base MySQL que Java
app.config['SQLALCHEMY_DATABASE_URI'] = 'mysql+mysqlconnector://root:root@
localhost/db_etudiants'
db = SQLAlchemy (app)

# Le Modèle (Equivalent @Entity)
class Etudiant (db.Model):
    __tablename__ = 'etudiant'
    identifiant = db.Column (db.Integer, primary_key=True, autoincrement=True)
    nom = db.Column(db.String (100), nullable=False)
    moyenne = db.Column (db.Float, nullable=False)

    def to_json(self):
        return {'identifiant': self.identifiant, 'nom': self.nom,

        'moyenne': self.moyenne}

# Exemple de Route GET utilisant l'ORM
@app.route('/etudiants', methods=['GET'])
def get_all():
    return jsonify ([e.to_json() for e in Etudiant.query.all()])
```

5 Conclusion

L'implémentation de l'API RESTful a été réussie sur les deux plateformes.

- **Java Spring Boot** offre une forte **automatisation** (serveur embarqué, injection de dépendances, JPA pour l'ORM).
- **Python Flask** propose une approche plus **modulaire et légère**.

Les deux démonstrations confirment que les concepts fondamentaux de l'architecture REST et CRUD sont universels, avec des mises en œuvre équivalentes entre Java (JPA/Hibernate) et Python (SQLAlchemy).