Rapport Final de TP : API REST et Microservices

De Java Spring Boot à Python Flask

RANDRIANARISOA Ambinintsoa

23 novembre 2025

Table des matières

1 Introduction et Objectifs **2**

2 Partie I : Mise en place de l'API Java (Simulation) **2**
2.1 1. Le Modèle de Données (POJO) 2
2.2 2. Le Contrôleur et la Liste Statique 2
2.3 3. Implémentation des Routes 3

3 Partie II : Persistance des Données (JPA & MySQL) **4**
3.1 1. Adaptation du Modèle (Migration Jakarta) 4
3.2 2. Création du Repository 4
3.3 3. Configuration MySQL 4

4 Partie III : Reproduction en Python (Flask) **4**
4.1 1. Installation 4
4.2 2. Code Complet (app.py) 5

5 Conclusion **5**

1 Introduction et Objectifs

Ce document sert de guide technique complet pour la réalisation d'une API RESTful de gestion d'étudiants. L'objectif est double :

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

Environnement Technique :

- Java 21 (Spring Boot 3.5.x)
- Python 3.x
- Base de données : MySQL

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

Dans cette première phase, nous créons le service sans base de données, en simulant le stockage via une liste en mémoire.

2.1 1. Le Modèle de Données (POJO)

Créez la classe `Etudiant`. Elle doit respecter l'encapsulation et fournir les accesseurs (Getters/-Setters) nécessaires à la conversion JSON.

```java
public class Etudiant {
    private int identifiant;
    private String nom;
    private double moyenne;

    public Etudiant() {} // Constructeur vide obligatoire

    public Etudiant(int identifiant, String nom, double moyenne) {
        this.identifiant = identifiant;
        this.nom = nom;
        this.moyenne = moyenne;
    }
    // Getters et Setters...
}
```

Listing 1: Etudiant.java

2.2 2. Le Contrôleur et la Liste Statique

Créez la classe `MyApi` annotée avec `@RestController`. Utilisez une `ArrayList` statique pour stocker les données temporairement.

```java
@RestController
public class MyApi {
    // Simulation de BDD en mémoire
    public static List<Etudiant> liste = new ArrayList<>();

    static {
        liste.add(new Etudiant(0, "A", 19));
```

```
        liste.add(new Etudiant(1, "A", 19));
        // ...
    }
    // ...
}
```

Listing 2: MyApi.java (Extrait)

2.3 3. Implémentation des Routes

Définissez les méthodes pour manipuler la liste :

- **GET /liste** : Retourne toute la collection.
- **GET /getEtudiant?identifiant=x** : Retourne l'élément à l'index x.
- **POST /addEtudiant** : Ajoute un étudiant à la liste.
- **PUT /modifier** : Met à jour le nom d'un étudiant.
- **DELETE /delete** : Supprime un étudiant par son index.

```
@GetMapping(value = "getEtudiant")
public Etudiant getEtudiant(int identifiant){
    return liste.get(identifiant);
}

@PostMapping(value = "addEtudiant")
public Etudiant addEtudiant(Etudiant etudiant){
    liste.add(etudiant);
    return etudiant;
}
```

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

L'étape suivante consiste à remplacer la liste statique par une véritable base de données.

3.1 1. Adaptation du Modèle (Migration Jakarta)

Modifiez la classe `Etudiant` pour en faire une Entité JPA.

Attention Java 21

Utilisez les imports `jakarta.persistence.*` et non `javax.*`.

```
@Entity
public class Etudiant {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id; // On passe en Long pour la BDD
    // ... rest of code
}
```

3.2 2. Création du Repository

Supprimez la `ArrayList` de `MyApi` et créez une interface Repository :

```
public interface EtudiantRepository extends JpaRepository<Etudiant, Long> {
    // Spring implémente automatiquement save(), findAll(), findById()...
}
```

3.3 3. Configuration MySQL

Configurez le fichier `application.properties` pour pointer vers votre base MySQL locale.

```
spring.datasource.url=jdbc:mysql://localhost:3306/db_etudiants
spring.datasource.username=root
spring.jpa.hibernate.ddl-auto=update
```

4 Partie III : Reproduction en Python (Flask)

L'objectif final est de prouver que l'architecture est transférable. Nous recréons l'API avec Flask et SQLAlchemy.

4.1 1. Installation

Installez les dépendances nécessaires :

```
pip install flask flask-sqlalchemy mysql-connector-python
```

4.2 2. Code Complet (app.py)

Voici la structure équivalente à notre application Java finale :

```python
from flask import Flask, jsonify, request
from flask_sqlalchemy import SQLAlchemy

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)

# 1. 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}

# 2. Les Routes (Equivalent @RestController)
@app.route('/etudiants', methods=['GET'])
def get_all():
    return jsonify([e.to_json() for e in Etudiant.query.all()])

@app.route('/etudiants', methods=['POST'])
def add():
    data = request.get_json()
    new_e = Etudiant(nom=data['nom'], moyenne=data['moyenne'])
    db.session.add(new_e)
    db.session.commit()
    return jsonify(new_e.to_json()), 201

if __name__ == '__main__':
    with app.app_context():
        db.create_all()
    app.run(debug=True, port=5000)
```

Listing 3: app.py : L'équivalent Python

5 Conclusion

Nous avons réussi à implémenter une API RESTful complète. Nous avons vu que :

- En **Java**, Spring Boot automatise énormément la configuration (Tomcat embarqué, Injection de dépendances, Repository).
- En **Python**, Flask offre une approche plus explicite et légère, mais les concepts d'ORM (SQLAlchemy vs Hibernate) et de routage restent identiques.