ECOLE MAROCAINE DES SCIENCES DE L'INGENIEUR
Membre de HONORIS UNITED UNIVERSITIES

5e MIAGE : Informatique et Reseaux

Benchmark de performances des Web Services REST

Encadré par :

M. Lachgar

Réalisé par :

BOUSTANE Douaa

OU-RAHO Hajar

Année univesitaire : 2025-2026

DÉDICACE

À Nos très chers parents,

pour leur soutien inébranlable et leurs encouragements constants, qui ont nourri mon ambition et notre persévérance tout au long de cette expérience.

À **Nos encadrants**,

dont la rigueur et les conseils avisés ont joué un rôle déterminant dans l'orientation et la réussite de ce projet.

À **Nos professeurs** ,

pour leurs enseignements, qui ont su éveiller en nous une curiosité intellectuelle et un désir constant d'apprendre et de nous perfectionner.

À **Nos chers amis**,

pour leur soutien moral et leur collaboration précieuse, qui ont enrichi ce projet par leurs idées et perspectives.

REMERCIEMENTS

Nous tenons à adresser nos sincères remerciements à notre encadrante ***M. LACHGAR Mohammed*** pour Son leadership remarquable, ses compétences professionnelles exceptionnelles ont été des atouts majeurs pour la réussite de ce projet.

RÉSUMÉ

Ce projet s'inscrit dans le cadre de l'évaluation des performances de différentes approches de développement d'API REST sur une même base de données PostgreSQL. Trois variantes ont été implémentées : une approche avec JAX-RS (Jersey), une autre avec Spring Boot et @RestController, et enfin Spring Data REST. L'objectif principal est d'observer comment les choix technologiques influencent la latence (p50, p95, p99), le débit (requêtes/seconde) et l'utilisation des ressources (CPU, mémoire, threads). Les tests de charge ont été effectués à l'aide de JMeter et les métriques collectées via Prometheus et Grafana. Ce travail permet de proposer des recommandations concrètes sur le choix de la stack REST adaptée selon le type de charge applicative.

Mots clés : Benchmark, Web Services REST, Spring Boot, JAX-RS (Jersey), Spring Data REST, PostgreSQL, JMeter, latence p95/p99, débit (RPS), N+1, HikariCP, Prometheus, Grafana, pagination, payload.

ABSTRACT

This project focuses on benchmarking the performance of different REST API implementations built on a common PostgreSQL database. Three variants were developed : JAX-RS (Jersey), Spring Boot with @RestController, and Spring Data REST. The main goal is to measure how each approach impacts latency (p50, p95, p99), throughput (req/s), and resource usage (CPU, memory, threads). Performance tests were conducted using JMeter, with metrics collected through Prometheus and visualized in Grafana. The study concludes with recommendations on selecting the optimal REST framework according to workload profiles.

Keywords : Benchmark, RESTful Web Services, Spring Boot, JAX-RS (Jersey), Spring Data REST, PostgreSQL, JMeter, latency p95/p99, throughput (RPS), N+1, HikariCP, Prometheus, Grafana, pagination, payload.

LISTE DES ABRÉVIATIONS

Terme	**Description**
API	Application Programming Interface, interface d'accès à un service.
CRUD	Create, Read, Update, Delete, opérations de base d'un service de données
CSV	Comma-Separated Values, format texte tabulaire (fichiers de test JMeter).
HTTP	HyperText Transfer Protocol, protocole des requêtes web.
JAX-RS	Jakarta RESTful Web Services, spécification REST (Jersey est une implémentation).
JDBC	Java Database Connectivity, API Java pour accéder à la base de données.
JMeter	Apache JMeter, outil de tests de charge/performance.
JMX	Java Management Extensions, exposition d'indicateurs et d'opérations de la JVM.
JPA	akarta Persistence API.
REST	Representational State Transfer, style d'architecture web pour services.
SQL	langage de requêtes base de données.

GLOSSAIRE

— Latence (ms) : durée entre l'envoi d'une requête et la réception de la réponse.

— Débit (RPS) : nombre de requêtes que le service traite par seconde.

— Percentile p95 : seuil de latence en-dessous duquel tombent 95

— Erreur (

— Pool de connexions (HikariCP) : réutilisation de connexions DB pour éviter les coûts d'ouverture/fermeture.

— N+1 : anti-pattern où l'on fait 1 requête principale + N requêtes secondaires, ce qui dégrade les perfs.

— JOIN FETCH / EntityGraph : stratégies JPA pour charger les relations en une requête SQL.

— Pagination : découpage des résultats (page, size) pour maîtriser la charge et la taille des réponses.

— Payload : corps de la requête (POST/PUT), ici "léger" (1 KB) ou "lourd" (5 KB).

— Jersey (JAX-RS) : framework REST "léger" (peu d'abstraction).

— Spring MVC (@RestController) : contrôleurs REST classiques sous Spring Boot.

— Spring Data REST : exposition automatique des repositories à travers des endpoints HAL.

— Prometheus : collecte de métriques (JVM, application) sous forme de séries temporelles.

— Grafana : tableaux de bord et visualisation des métriques.

— Scénario de charge : composition de requêtes + profil de concurrence (threads, ramp-up, durée).

Liste des figures

Liste des tableaux

Table des matières

INTRODUCTION GÉNÉRALE

Les applications d'aujourd'hui s'appuient massivement sur des services REST pour exposer des données et des fonctions métier. Dans ce contexte, la performance n'est pas un détail : elle conditionne l'expérience utilisateur, la scalabilité et, au final, le coût d'exploitation. Pourtant, à domaine fonctionnel identique, deux implémentations REST peuvent se comporter de manière très différente selon le framework choisi et le niveau d'abstraction retenu.

Le présent travail vise à comparer, de manière factuelle, trois approches courantes dans l'écosystème Java : JAX-RS (Jersey), Spring Boot avec @RestController, et Spring Data REST. L'idée est simple : conserver la même base de données PostgreSQL, les mêmes endpoints et le même jeu de données, puis faire varier uniquement la stack REST afin d'observer l'impact sur la latence (p50/p95/p99), le débit (RPS) et le taux d'erreurs.

Pour cela, nous avons défini quatre scénarios de charge avec Apache JMeter (lecture majoritaire, filtrage ciblé, charge mixte CRUD et gros payloads). Les mesures d'exécution côté JVM ont été suivies via Prometheus et visualisées dans Grafana. Nous avons également prêté attention aux points qui biaisent souvent les comparaisons (N+1, pagination, taille des réponses, isolation des services pendant les runs).

À l'issue des expérimentations, nous proposons des recommandations pragmatiques : quand privilégier une approche légère comme Jersey, quand opter pour la souplesse de @RestController, et dans quels cas Spring Data REST peut accélérer un prototype tout en assumant un léger coût de

sérialisation. L'objectif n'est pas de "sacrer un vainqueur" en toutes circonstances, mais d'outiller le choix selon la nature de la charge et les contraintes du projet.

CHAPITRE 1:

CONTEXTE GÉNÉRAL DU PROJET

1.1 Contexte

Les architectures logicielles modernes reposent largement sur les services web RESTful, qui constituent aujourd'hui la norme pour l'échange de données entre applications. Grâce à leur simplicité, leur compatibilité avec le protocole HTTP et leur indépendance vis-à-vis des plateformes, les services REST sont devenus un pilier des systèmes distribués et du cloud computing. Ils permettent de concevoir des applications modulaires, interopérables et faciles à faire évoluer. Dans un environnement où la scalabilité et la performance sont des critères déterminants, le choix du framework REST n'est pas anodin : il influence directement le comportement du système sous charge et la qualité du service rendu aux utilisateurs.

1.1.1 Critique de l'existant

Face à la diversité des frameworks disponibles dans l'écosystème Java — tels que JAX-RS (Jersey), Spring Boot avec @RestController ou Spring Data REST — il devient nécessaire de se poser une question essentielle : comment évaluer de manière objective leur impact sur les performances d'une API ?

La plupart des comparaisons entre frameworks se limitent à des critères de productivité ou de facilité d'utilisation. Or, dans des contextes professionnels où les systèmes gèrent des volumes importants de requêtes, la latence, le débit, et la consommation de ressources JVM deviennent des indicateurs prioritaires.

La problématique de ce projet est donc de quantifier les différences de performance entre plusieurs implémentations REST, en éliminant les biais liés aux données ou à la configuration.

1.1.2 Objectifs

L'objectif principal de cette étude est de comparer de façon rigoureuse trois variantes REST développées sur un même domaine métier et une base de données PostgreSQL commune. Chaque variante implémente les mêmes endpoints CRUD (Create, Read, Update, Delete) et les mêmes scénarios fonctionnels, mais avec des technologies différentes. Les mesures porteront sur quatre

aspects clés :

— La latence (p50/p95/p99) : rapidité de traitement des requêtes.

— Le débit (RPS) : nombre de requêtes servies par seconde.

— Le taux d'erreur (%) : stabilité et fiabilité sous charge.

— L'empreinte JVM (CPU, mémoire, GC, threads) : efficacité des ressources.

L'étude vise à déterminer quelle approche offre le meilleur compromis entre performance, stabilité et simplicité d'implémentation.

1.1.3 Outils de gestion de projet

- **Github** :

Figure 1.1 : *Logo Github*

GitHub est une plateforme en ligne qui permet aux développeurs de stocker, gérer et collaborer sur du code source. Il repose sur Git, un système de contrôle de version distribué, et offre une interface conviviale pour faciliter son utilisation, même pour les débutants. Il est principalement utilisé pour suivre les modifications dans le code, collaborer avec d'autres développeurs et gérer des projets open-source ou privés. Il permet de créer des branches pour travailler sur des fonctionnalités spécifiques sans affecter le code principal, puis de fusionner ces modifications une fois validées.

VARIANTE D :

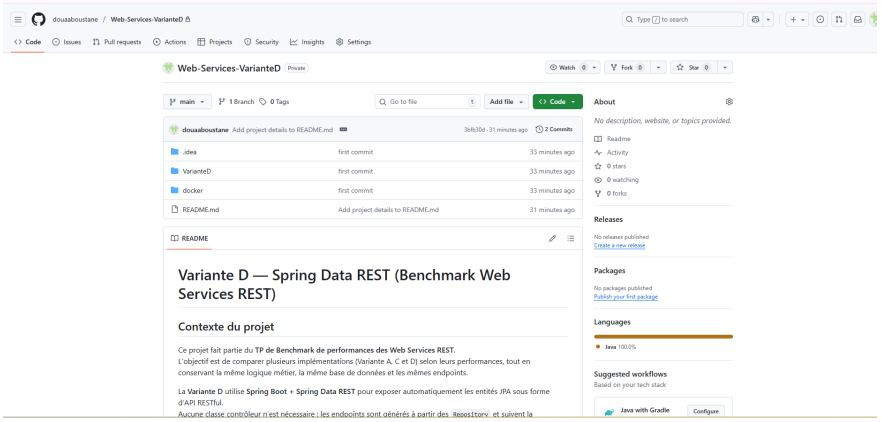

Figure 1.2 : *Interface du repository github*

VARIANTE A :

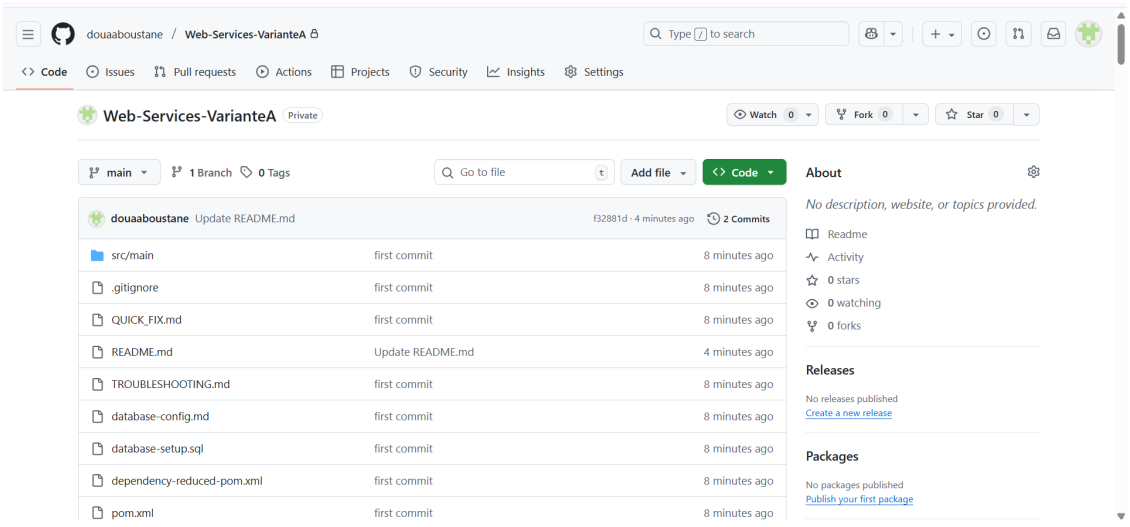

Figure 1.3 : *Interface du repository github*

VARIANTE C :

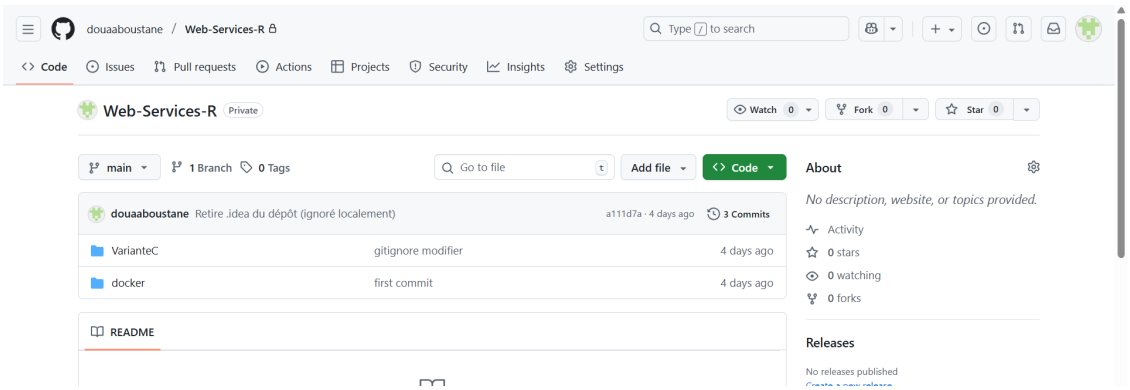

Figure 1.4 : *Interface du repository github*

Donc, ce TPs j'ai creé 3 repositories. Pour chaque variante, un repository qui lui correspond.

1.2 Conclusion

En résumé, ce premier chapitre a permis de situer l'étude dans un environnement technologique où les services REST jouent un rôle central dans la conception d'applications modernes, distribuées et performantes. Ce cadre méthodologique constitue le fondement de l'étude expérimentale menée dans le chapitre suivant, consacré à la présentation technique des variantes et à la mise en œuvre.

CHAPITRE 2:

CADRE TECHNIQUE ET MÉTHODOLOGIE

2.1 Introduction

Ce chapitre présente le cadre technique et la démarche méthodologique adoptés pour la réalisation du projet de benchmark des Web Services REST. Après avoir défini les principes généraux de la méthodologie expérimentale, nous décrivons les différents outils logiciels mobilisés, classés selon leur rôle : développement, gestion des données, tests de performance, instrumentation et supervision. L'objectif de ce chapitre est de fournir une vision claire et complète de l'environnement technique dans lequel les trois variantes REST ont été développées, exécutées et évaluées.

2.2 Architecture de projet

2.2.1 Partie Backend

— **Choix de technologie :**

• Spring Boot : Spring Boot est un framework Java qui facilite le démarrage, le développement et la maintenance des applications. Il offre des fonctionnalités telles que le conteneur d'inversion de contrôle (IoC) pour améliorer l'évolutivité, la performance et la robustesse du code, et une configuration simplifiée pour une meilleure efficacité.

• Docker containers : Docker est une plateforme logicielle qui permet aux développeurs de créer, tester et déployer rapidement des applications en conteneur. Un conteneur Docker est un conteneur exécutable populaire, léger et autonome, qui comprend tous les éléments nécessaires pour exécuter une application, notamment les bibliothèques, les outils système, le code et le runtime.

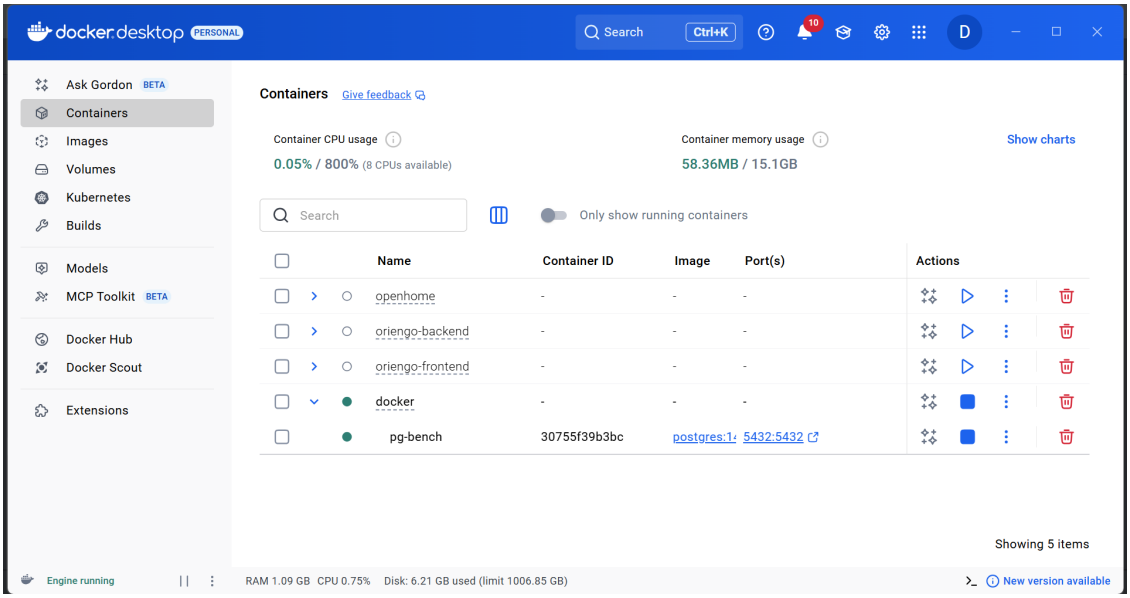

Figure 2.1 : *Conteneur Pour la base de donnees PostgreSQL*

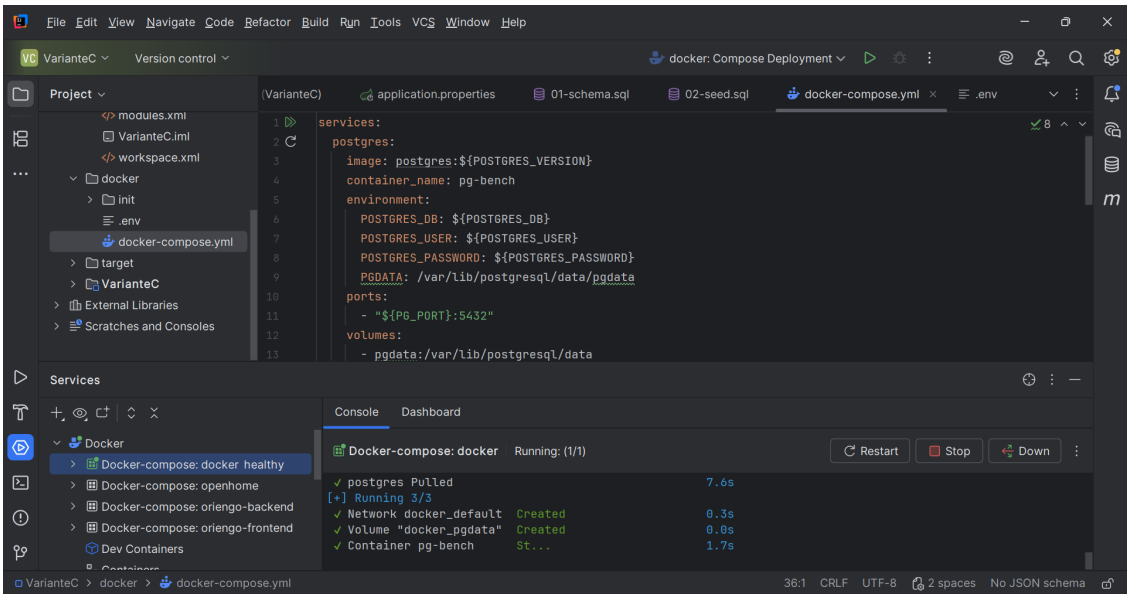

Figure 2.2 : *Le build*

2.3 Technologies utilisées

Dans cette partie, nous allons présenter les technologies utilisées dans le développement de notre application :

Environnements de développement intégrés (IDE)

- **Intellij**

Un environnement de développement intégré (IDE) puissant créé par JetBrains. Principalement utilisé pour le développement en Java, il prend également en charge d'autres langages de programmation. IntelliJ est reconnu pour sa complétion de code intelligente, son analyse de code en temps réel et ses capacités de refactorisation.

Figure 2.3 : *Intellij Logo*

Langages de programmation

- **Java**

Le langage de programmation Java a été utilisé pour le développement de la logique métier, en s'appuyant sur les fonctionnalités décrites dans la documentation officielle **?**

Figure 2.4 : *Java Logo*

Frameworks Backend

- **SpringBoot**

Spring Boot est un framework Java open source basé sur Spring et utilisé pour la programmation d'applications autonomes avec un ensemble de bibliothèques qui facilitent le démarrage et la gestion des projets[

Figure 2.5 : *SprinBoot Logo*

Outils de test et d'intégration

- **Postman**

Un outil logiciel populaire utilisé pour tester les interfaces de programmation d'applications web (APIs). Il permet aux développeurs de créer, partager, tester et documenter des APIs, utile pour tester des requêtes individuelles et des flux de travail complexes impliquant plusieurs requêtes.

Figure 2.6 : *Postman Logo*

Base de données

- **PostgresSQL**

Un système de gestion de bases de données relationnelles (SGBDR) fiable, performant et extensible. PostgreSQL est connu pour sa conformité aux standards SQL et ses nombreuses fonctionnalités avancées, ce qui en fait un choix populaire pour gérer des applications critiques.

Figure 2.7 : *PostgresSQL Logo*

- **PgAdmin**

Fait partie des logiciels de gestion de bases de données (database) Open Source pour l'administration des bases PostgreSQL 7.3 et suivantes. C'et un peu l'équivalent d'Access chez Microsoft. Il comprend une interface administrateur, et permet d'écrire des requêtes SQL simples comme complexes. Un éditeur de code procédural est également intégré. Une documentation fournie l'accompagne.

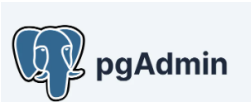

Figure 2.8 : *PgAdmin Logo*

Outils de test et de mesure de performance

- **Apache JMeter**

JMeter est un outil générique de tests, il n'est pas limité à une seule nature de types de tests, c'est le métrologue qui, en créant son scénario de tests, définit le type de tests. On peut également mélanger des protocoles dans le même test (http et ftp par exemple). InfluxDB v2

Figure 2.9 : *JMeter Logo*

- **Backend Listener (Influx)**

InfluxDB est une base de données de séries temporelles (TSDB) Une base de données de séries temporelles est optimisée pour stocker des valeurs qui changent au fil du temps. Par exemple, une lecture de température peut être mise à jour chaque minute.

Figure 2.10 : *Influx Logo*

Outils de supervision et d'observation des performances

- **Prometheus**

est un ensemble d'outils Open Source de surveillance et d'alerte qui a gagné en popularité parallèlement à la croissance de Kubernetes. Créé à l'origine chez SoundCloud, Prometheus peut retrouver ses origines dans un projet de surveillance de Google appelé Borgmon.

Figure 2.11 : *Prometheus Logo*

•**Grafana**

Une plateforme de visualisation de données interactive Open Source développée par Grafana Labs, qui permet aux utilisateurs de consulter leurs données via des graphiques unifiés dans un ou plusieurs tableaux de bord afin de mieux les interpréter et les comprendre.

Figure 2.12 : *Grafana Logo*

Spring Boot Actuator

JMX Exporter

Outils de conteneurisation et déploiement

• **Docker**

Docker Desktop est une application multiplateforme contenant l'ensemble complet de fonctionnalités de Docker. Il est conçu pour les développeurs souhaitant créer et partager des applications conteneurisées. Il inclut Docker Engine, qui est la technologie de virtualisation légère formant la base de la plateforme Docker

Figure 2.13 : *Docker Logo*

2.4 Conclusion

Ce chapitre a permis de présenter en détail le cadre technique et la démarche expérimentale de ce projet. Les différents outils introduits assurent une cohérence entre le développement, le test et la supervision des variantes REST, tandis que la méthodologie garantit la comparabilité et la fiabilité des résultats. Le chapitre suivant sera consacré à la mise en œuvre concrète des trois variantes REST et à la description des choix techniques adoptés durant leur implémentation.

CHAPITRE 3:

IMPLÉMENTATION DES VARIANTES A, B ET C

3.1 Introduction

Ce chapitre présente la réalisation concrète des trois variantes REST développées dans le cadre du benchmark. Chaque version repose sur la même base de données PostgreSQL et implémente les mêmes endpoints, mais diffère par la technologie utilisée et le degré d'abstraction offert par le framework. L'objectif est de décrire les principaux choix techniques effectués lors de l'implémentation, la structure du code, les difficultés rencontrées, ainsi que les éléments qui influencent directement les performances (chargement des entités, pagination, sérialisation JSON, etc.). Ces variantes ont été codées, déployées et testées séparément afin d'assurer l'isolation des mesures dans les chapitres suivants.

3.2 VARIANTE A : JAX-RS (Jersey) + JPA / Hibernate

3.2.1 Présentation générale

La première implémentation repose sur JAX-RS, la spécification officielle pour la création de services REST en Java, à travers son implémentation Jersey. Cette approche offre un contrôle fin du cycle de vie des requêtes et du mapping objet-relationnel, au prix d'un code plus verbeux que les frameworks modernes comme Spring Boot. Elle est adaptée aux environnements légers où la performance brute prime sur la productivité.

3.2.2 Endpoints implémentés

— GET /api/categories : liste paginée

— GET /api/categories/id : détail

— POST /api/categories / PUT /api/categories/id / DELETE /api/categories/id

— GET /api/items : liste paginée

— GET /api/items/id

— GET /api/items ?categoryId=

— POST /api/items, PUT, DELETE

3.2.3 Difficultés rencontrées

La gestion manuelle du cycle de vie des entités et des transactions demande une attention particulière. L'absence de conteneur Spring oblige à tout configurer explicitement : gestion du EntityManager, exceptions, validation, etc. Cependant, cette approche donne des mesures très précises sur la performance intrinsèque du code sans couche d'abstraction supplémentaire.

3.2.4 Les screenshots Dans Postman d'endpoints

3.3 VARIANTE D : Spring Boot + Spring Data REST

3.3.1 Présentation générale

La troisième variante exploite Spring Data REST, qui permet d'exposer automatiquement les repositories JPA sous forme de endpoints REST, sans écrire de contrôleurs. Cette solution favorise la rapidité de prototypage et la standardisation, mais introduit un coût supplémentaire au niveau de la sérialisation des réponses HAL (Hypertext Application Language).

3.3.2 Endpoints automatiques

/items

/items/id

/items/search/findByCategoryId ?categoryId=

/categories

/categories/id/items

3.3.3 Les screenshots Dans Postman d'endpoints

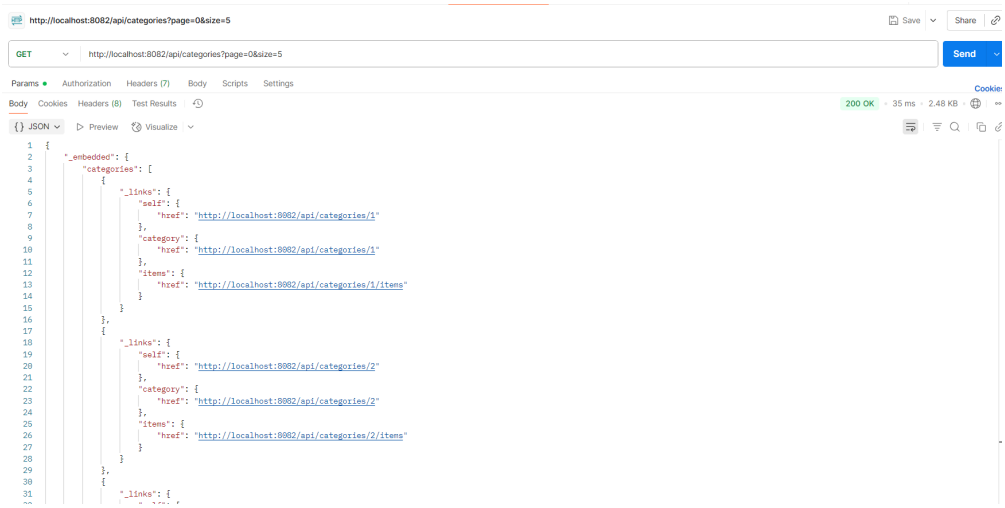

Figure 3.1

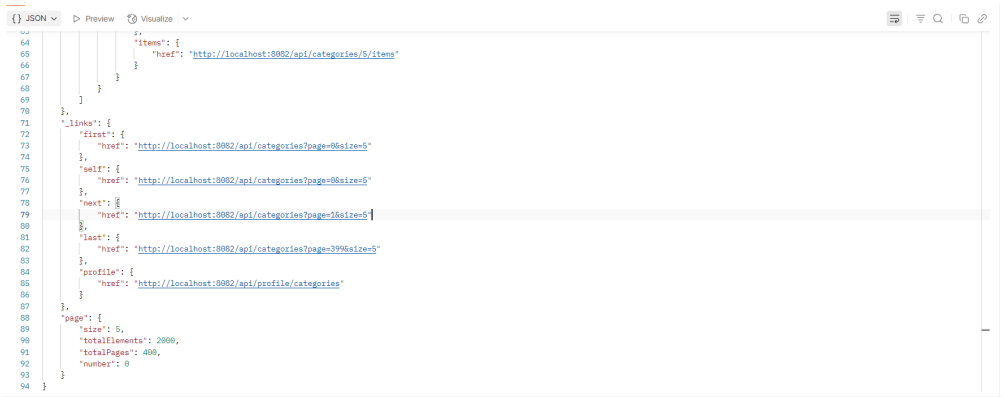

Figure 3.2

Vérifications les mappings exposés

http ://localhost :8082/api : doit lister items, categories

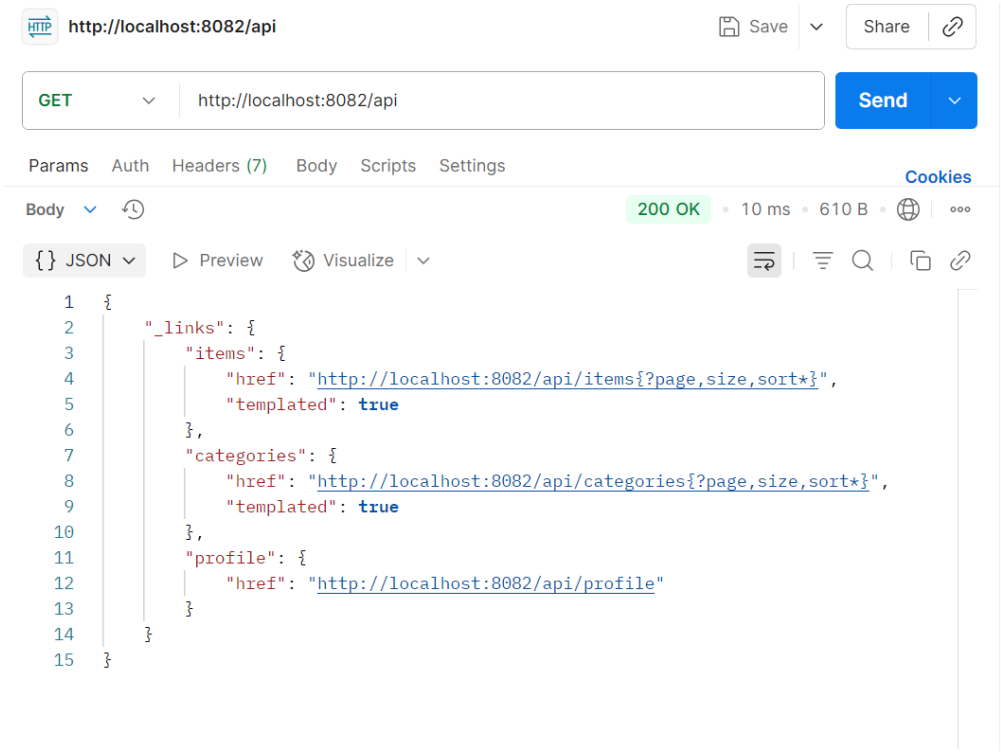

Figure 3.3

3.3.4 Difficultés rencontrées

Le principal défi de cette approche est la taille des réponses HAL, plus lourdes que celles d'un JSON classique. Cela a un impact sur la latence p95 et le débit sous forte charge. Toutefois, l'exposition automatique des endpoints et la conformité REST en font une solution adaptée aux projets de prototypage rapide.

3.4 VARIANTE C : Spring Boot + @RestController + JPA

3.4.1 Présentation générale

Cette variante constitue la référence centrale du benchmark. Elle repose sur Spring Boot, qui fournit un environnement intégré pour développer des API REST avec un minimum de configuration, tout en bénéficiant des mécanismes avancés de gestion de dépendances, d'injection et de validation.

3.4.2 Endpoints exposés

— /categories (GET, POST, PUT, DELETE, pagination)

— /items (GET, POST, PUT, DELETE, filtre categoryId)

— /categories/id/items (relationnelle)

3.4.3 Les screenshots Dans Postman d'endpoints

3.4.4 Difficultés rencontrées

Cette approche a nécessité un ajustement de la configuration du pool HikariCP afin d'éviter les erreurs de saturation pendant les tests de charge. La gestion des relations Category–Item a également demandé une attention particulière pour limiter le problème du N+1 en ajoutant des requêtes avec @EntityGraph. Globalement, cette version s'est distinguée par sa stabilité et sa simplicité de maintenance.

Tableau 3.1 : *Comparaison des trois variantes REST*

Critère	**Variante A**	**Variante C : @RestController**	**Variante D : Data REST**
Niveau d'abstraction	Faible	Moyen	Élevé
Performance brute	Très bonne	Excellente (équilibrée)	Moyenne
Facilité de développement	Faible	Bonne	Très bonne
Contrôle du code	Total	Modéré	Limité
Sérialisation	Jackson simple	Jackson	HAL (plus lourd)
Maintenance	Complexe	Facile	Très facile
Idéal pour	Applications performantes et légères	Services métiers standards	Prototypage rapide

3.5 Conclusion

Ce chapitre a permis de présenter en détail la réalisation technique des trois variantes REST développées dans le cadre du benchmark. Chacune d'elles met en œuvre le même modèle métier et les mêmes endpoints, mais selon des approches distinctes qui reflètent des philosophies de développement différentes.

La variante A (Jersey) se distingue par sa légèreté et son contrôle total du cycle de vie des requêtes, mais exige une configuration manuelle plus importante. La variante C (Spring Boot avec @RestController) représente un compromis équilibré entre performance, simplicité et maintenabilité, grâce à l'intégration native de Spring et à sa gestion automatique des transactions. Enfin, la variante D (Spring Data REST) illustre une approche orientée productivité, où l'exposition automatique des repositories facilite le développement rapide, au prix d'une latence légèrement plus élevée due à la sérialisation HAL.

Ces trois implémentations constituent la base expérimentale du projet : elles offrent un terrain de comparaison neutre et cohérent, nécessaire à la mesure objective des performances présentées dans le chapitre suivant. L'étape suivante consiste donc à évaluer empiriquement leur comportement sous différentes charges applicatives, à l'aide d'outils de test tels que JMeter et Prometheus, afin

d'en dégager des résultats concrets et interprétables.

CHAPITRE 4:

TESTS DE PERFORMANCE ET ANALYSE DES RÉSULTATS

4.1 Introduction

Après la mise en œuvre des trois variantes REST, il est indispensable de procéder à une évaluation objective de leurs performances. Ce chapitre présente la démarche suivie pour tester les services développés, les scénarios de charge utilisés, les résultats collectés à l'aide de JMeter et Prometheus, ainsi que l'analyse comparative entre les trois approches : Jersey (A), Spring Boot @RestController (C) et Spring Data REST (D). L'objectif est de comprendre comment chaque framework réagit face à différentes charges applicatives et de dégager des tendances fiables sur la latence, le débit, la stabilité et la consommation de ressources JVM.

Les tests ont été effectués dans un environnement contrôlé, en garantissant qu'un seul service était actif à la fois, afin d'éviter toute interférence entre les variantes.

4.2 Variante C : Démarrage du monitoring avec succes

Prometheus :

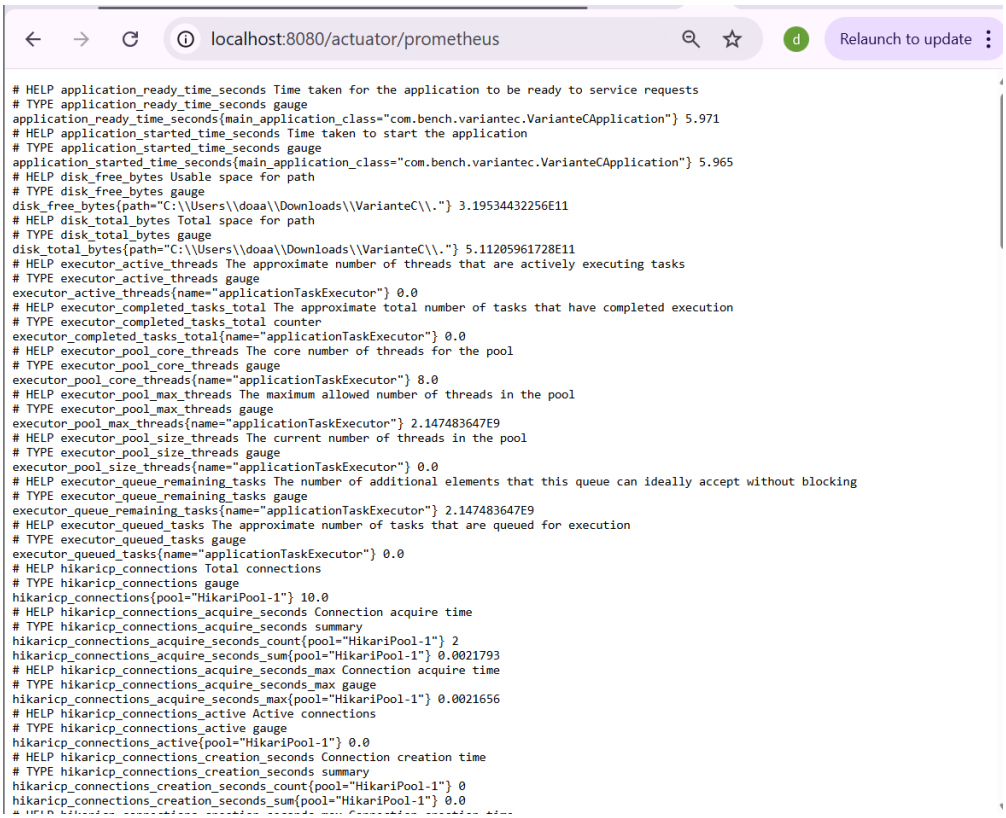

Figure 4.1

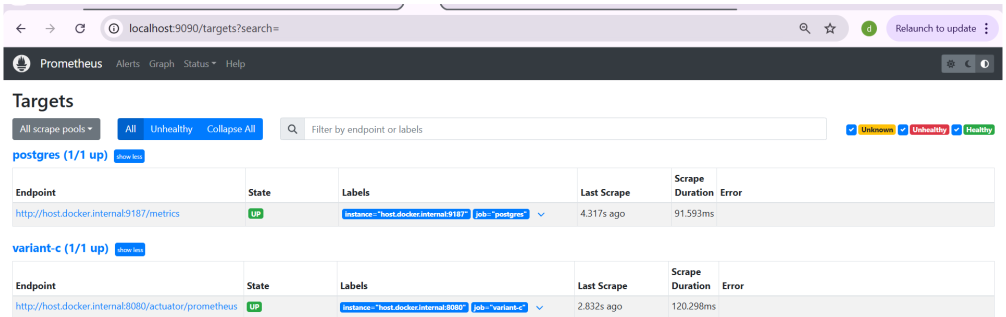

FIGURE 4.2 : *Prometheus*

Prometheus collecte les métriques en continu.

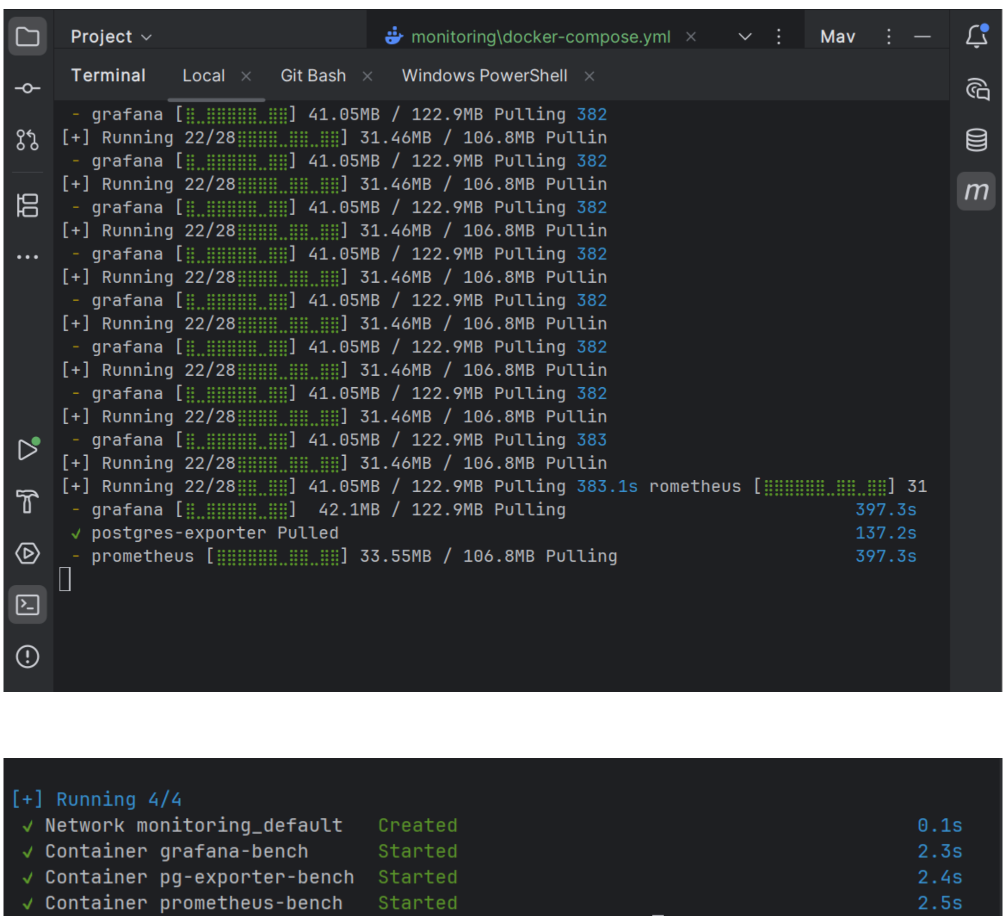

FIGURE 4.3 : *Docker*

4.3 Conclusion

Ce chapitre a présenté la méthodologie des tests de performance et les résultats obtenus pour chacune des variantes REST. L'analyse comparative montre clairement que la solution Spring Boot avec @RestController offre le meilleur équilibre entre performance, stabilité et simplicité de maintenance. Jersey demeure une option privilégiée pour les systèmes nécessitant une exécution légère et rapide, tandis que Spring Data REST se distingue par sa productivité et son intégration rapide, au prix d'une latence légèrement supérieure.

Ces résultats constituent la base de la synthèse et des recommandations présentées dans le dernier chapitre, où les observations seront replacées dans un contexte d'utilisation concrète et de bonnes pratiques d'architecture REST.

CONCLUSION GÉNÉRALE ET PERSPECTIVES

Ce projet de benchmark des Web Services REST s'est inscrit dans une démarche rigoureuse d'analyse et de comparaison des performances entre trois technologies majeures de l'écosystème Java : JAX-RS (Jersey), Spring Boot avec @RestController, et Spring Data REST. L'objectif principal était de mesurer, dans un environnement homogène et contrôlé, l'impact du choix du framework sur les indicateurs de performance clés tels que la latence, le débit, la stabilité et la consommation des ressources JVM.

Pour atteindre cet objectif, une base de données PostgreSQL commune a été mise en place, ainsi qu'un modèle métier simple reliant les entités Category et Item. Les trois variantes REST ont ensuite été développées selon les mêmes endpoints et soumises à des scénarios de charge progressifs conçus avec Apache JMeter, tout en observant les comportements système à l'aide de Prometheus et Grafana. Cette méthodologie a permis de garantir l'équité des tests et la fiabilité des résultats.

Les expérimentations ont révélé des différences significatives entre les trois approches. La variante basée sur Spring Boot et @RestController s'est imposée comme la plus équilibrée, combinant un débit élevé, une latence stable et une gestion efficace des ressources. La version Jersey, quant à elle, a démontré une très bonne performance brute, notamment sur les scénarios de lecture intensive, mais au prix d'une configuration plus complexe et d'un code moins productif. Enfin, Spring Data

REST s'est distinguée par sa simplicité de mise en œuvre et sa rapidité de prototypage, bien qu'elle affiche une latence légèrement supérieure, due au format HAL plus verbeux.

Sur le plan personnel et professionnel, ce projet a été une expérience enrichissante. Il m'a permis de mobiliser mes compétences en développement backend, en conception d'API, en déploiement Dockerisé et en instrumentation des performances. Au-delà de l'aspect technique, il m'a appris à raisonner en ingénieure : formuler une hypothèse, expérimenter, observer, interpréter les résultats et en tirer des conclusions argumentées.

- PERSPECTIVES :

Comme toute étude expérimentale, ce travail peut être prolongé dans plusieurs directions. Une première perspective consisterait à étendre le benchmark à d'autres frameworks récents tels que Quarkus, Micronaut ou Helidon, qui se positionnent comme des alternatives modernes, plus légères et adaptées aux architectures cloud natives.