Benchmark de performances des Web Services REST

Étude comparative des variantes A, B et C

Réalisé par :

Majdouline Sabri et Youssef Ait Rais

Filière : MIAGE – 5e année

École : EMSI Marrakech

Date : Novembre 2025

Table des matières

Chapitre 1

Introduction

Ce projet a pour objectif de concevoir et comparer les performances de plusieurs architectures de Web Services REST en Java. Le travail s'inscrit dans une approche expérimentale et collective visant à identifier les meilleures pratiques pour le développement de services web performants et scalables.

Trois variantes d'implémentation ont été réalisées :

— **Variant A** : Architecture classique avec JAX-RS (Jersey), Hibernate et PostgreSQL.

— **Variant B** : Architecture basée sur Spring Boot et JPA.

— **Variant C** : Architecture Spring Boot optimisée par un cache Redis.

Chaque variante a été développée, déployée et testée selon la même méthodologie de benchmark afin de garantir des résultats comparables.

Mots-clés : JAX-RS, Spring Boot, Hibernate, PostgreSQL, Redis, Grafana, JMeter, Benchmark

Chapitre 2

Objectifs du benchmark

Les objectifs principaux de cette étude sont :

— Évaluer les performances de chaque architecture selon plusieurs métriques.

— Identifier les points forts et les limites de chaque approche.

— Dégager des recommandations pour la conception d'API REST performantes.

Les mesures ont porté sur :

— le temps de réponse moyen,

— le débit de requêtes par seconde,

— la consommation mémoire et CPU du serveur.

Chapitre 3

Architecture et technologies

3.1 Présentation générale

Les trois variantes reposent sur une même base de données **PostgreSQL 14+**, alimentée par un jeu de données identique pour garantir l'équité du test.

3.2 Technologies communes

- Langage : Java 17
- Serveur d'application : Apache Tomcat 10.1
- Outils de mesure : JMeter, Prometheus, Grafana
- ORM : Hibernate / JPA

3.3 Endpoints principaux

```
GET    /api/categories?page=&size=
GET    /api/items?page=&size=
GET    /api/items?categoryId=&page=&size=
POST   /api/items
PUT    /api/items/{id}
DELETE /api/items/{id}
```

Chapitre 4

Environnement et configuration

4.1 Configuration technique

— Système d'exploitation : Windows 11 (x64)
— Java JDK 17
— PostgreSQL 17 – Base : benchmarkdb
— Tomcat 10.1.36 – Port 8080

4.2 Configuration matérielle du poste de test

Les performances mesurées lors d'un benchmark dépendent fortement de la puissance de la machine utilisée. Afin de garantir la transparence et la reproductibilité des résultats, la configuration matérielle du poste ayant exécuté l'ensemble des scénarios est présentée ci-dessous :

— **Processeur :** Intel Core i7-13620H (10 cœurs hybrides : 6 Performance + 4 Efficient)
— **Mémoire RAM :** 16 Go DDR5
— **Carte graphique :** NVIDIA GeForce RTX 4060 (8 Go GDDR6) — non sollicitée pour les tests
— **Stockage :** SSD NVMe PCIe 512 Go
— **Écran :** 15.6" Full HD (1920×1080)
— **Système d'exploitation :** Windows 11 Pro 64 bits
— **Serveur d'application :** Apache Tomcat 10.1.36
— **JDK :** Java 17 (Oracle/OpenJDK)

Cette configuration relativement récente (CPU de 13^{e} génération et RAM DDR5) permet d'obtenir une bonne stabilité durant les tests. Il est important de noter que les mesures (débit RPS, latence, consommation CPU) pourraient varier sur une machine

moins performante ou un environnement serveur dédié.

4.3 Connexion base de données

```
<property name="jakarta.persistence.jdbc.url"
 value="jdbc:postgresql://localhost:5432/benchmarkdb"/>
<property name="jakarta.persistence.jdbc.user" value="postgres"/>
<property name="jakarta.persistence.jdbc.password" value="********"/>
```

4.4 Optimisation et neutralité des mesures

- Pool de connexions HikariCP : minIdle = 10, maxPoolSize = 20
- Cache HTTP et Hibernate désactivés
- Monitoring activé via JMX Exporter et Prometheus

Chapitre 5

Jeu de données

Afin d'obtenir des mesures significatives, un jeu de données important a été généré dans la base PostgreSQL :

— 2 000 catégories

— 100 000 articles (environ 50 par catégorie)

5.1 Script SQL de génération

```
-- Génération de 2000 catégories et 10000 items
--  Clean old data first
TRUNCATE TABLE item CASCADE;
TRUNCATE TABLE category CASCADE;

--  Step 1: Generate 2,000 categories
DO $$
BEGIN
  FOR i IN 1..2000 LOOP
    INSERT INTO category (code, name, updated_at)
    VALUES (
      'CAT' || LPAD(i::text, 4, '0'),
      'Category ' || i,
      NOW()
    );
  END LOOP;
END $$;

--  Step 2: Generate 100,000 items (~50 per category)
```

```
DO $$
DECLARE
  v_cat_id INT;
  v_counter INT := 1;
BEGIN
  -- ensure categories exist before this loop runs
  FOR v_cat_id IN SELECT id FROM category LOOP
    FOR i IN 1..50 LOOP
      INSERT INTO item (sku, name, price, stock, category_id, updated_at)
      VALUES (
        'SKU' || LPAD(v_counter::text, 6, '0'),
        'Item ' || v_counter,
        ROUND((random() * 1000 + 10)::numeric, 2),
        (random() * 100 + 1)::int,
        v_cat_id,
        NOW()
      );
      v_counter := v_counter + 1;
    END LOOP;
  END LOOP;
END $$;
```

Chapitre 6

Méthodologie de test

6.1 Outil de benchmark

Les tests de charge ont été réalisés à l'aide de **Apache JMeter**, un outil open source dédié à la mesure des performances des applications web. L'objectif était d'évaluer la capacité du service REST à supporter un nombre croissant de requêtes simultanées et à mesurer la latence moyenne, le débit et la stabilité du système.

Configuration du test

— **Nombre d'utilisateurs virtuels (threads)** : 1000
— **Temps de montée en charge (ramp-up)** : 10 secondes
— **Durée de chaque test** : entre 2 et 5 minutes selon le scénario
— **Base de données utilisée** : PostgreSQL 14
— **Pool de connexions HikariCP** : `minIdle = 10`, `maxPoolSize = 20`

Remarque : si tu modifies ces paramètres dans Grafana ou JMeter (par exemple, plus ou moins de threads, ou une durée différente), pense à mettre à jour cette sous-section.

—

6.2 Requêtes testées

Les tests de performance ont porté sur plusieurs types de requêtes REST afin de mesurer la latence, le débit, la consommation mémoire et la stabilité de l'application sous charge. Les scénarios suivants ont été simulés via **Apache JMeter** et observés à l'aide de **Grafana** connectée à Prometheus.

1. Requêtes de lecture (GET)

— **GET /categories ?page=0&size=10** : récupération paginée des catégories.
— **GET /items ?page=0&size=10** : récupération paginée de la liste complète des articles.
— **GET /items ?categoryId=5&page=0&size=10** : récupération des articles appartenant à une catégorie spécifique.

2. Requêtes d'écriture et de mise à jour (POST, PUT)

— **POST /categories** : création d'une nouvelle catégorie.
— **POST /items** : ajout d'un article lié à une catégorie existante.
— **PUT /items/{id}** : mise à jour du nom, du prix, du stock et de la catégorie d'un article existant.

3. Requêtes de suppression (DELETE)

— **DELETE /items/{id}** : suppression d'un article de la base de données.

4. Requêtes de charge et de pagination avancées

— **GET /items ?page=50&size=50** : test de profondeur de pagination.
— **GET /items ?categoryId=100&page=5&size=20** : test combiné de filtrage et pagination.

Ces requêtes couvrent l'ensemble du cycle CRUD (*Create, Read, Update, Delete*) et permettent d'évaluer la stabilité du système sous différents volumes de données. Elles ont été exécutées sur un jeu de données contenant environ **2 000 catégories** et **100 000 articles** répartis aléatoirement.

Remarque : si ton dataset réel diffère (par exemple 500 catégories, 50 000 items), adapte ces chiffres dans le texte.

—

6.3 Plan de test JMeter

TABLE 6.1 – Plan de test JMeter utilisé pour la simulation de charge

Type	URL	Threads	Ramp-up	Durée	Assertions
GET	/api/items ?page=0&size=10	1000	10s	3 min	HTTP 200
GET	/api/items ?categoryId=10	1000	10s	3 min	Corps non vide
POST	/api/items	100	5s	2 min	HTTP 201
PUT	/api/items/{id}	100	5s	2 min	HTTP 200
DELETE	/api/items/{id}	100	5s	2 min	HTTP 204

Remarque : adapte les valeurs de threads ou de durée si tu modifies la configuration JMeter pendant tes tests.

—

6.4 Surveillance des performances

La surveillance de l'environnement d'exécution a été assurée à l'aide de **Prometheus** pour la collecte de métriques JVM (CPU, mémoire, threads, connexions actives) et de **Grafana** pour la visualisation en temps réel.

— **Prometheus** : collecte automatique des métriques d'exécution (via JMX Exporter).

— **Grafana** : visualisation graphique des indicateurs de performance et création de tableaux de bord.

Remarque : dans cette section, tu pourras insérer les captures de tes dashboards Grafana — par exemple, l'évolution de la mémoire, la charge CPU, ou le throughput — une fois les résultats obtenus.

—

6.5 Indicateurs de performance collectés

TABLE 6.2 – Métriques observées pendant le benchmark

Indicateur	**Unité**	**Source**
Temps de réponse moyen	ms	JMeter
90^{e} percentile	ms	JMeter
Débit (Throughput)	req/s	JMeter
Utilisation mémoire JVM	MB	Grafana / Prometheus
Charge CPU JVM	%	Grafana / Prometheus
Threads actifs	nombre	JMeter
Taux d'erreur	%	JMeter

Chapitre 7

Analyse détaillée des résultats du Scénario 1

Cette section présente et analyse les résultats obtenus lors du scénario 1 (workload majoritairement en lecture) pour les trois variantes de l'architecture REST testées : Variante A, Variante C et Variante D. Chaque graphique est inséré sous forme de figure et accompagné d'une interprétation détaillée.

7.1 Variante A — Scénario 1

7.1.1 CPU, RPS et Load Average

FIGURE 7.1 – Variante A — CPU, débit (RPS) et charge système.

Analyse : Le CPU reste extrêmement bas (autour de 0.003–0.004 s/s), ce qui témoigne d'une faible pression sur le processeur malgré une charge élevée. Le débit reste très stable,

entre 1000 et 1300 RPS, signe d'une excellente capacité de traitement simultané. Le load average est pratiquement nul, montrant que l'application ne subit aucune saturation. La Variante A démontre une stabilité remarquable même sous forte charge.

7.1.2 Mémoire résidente

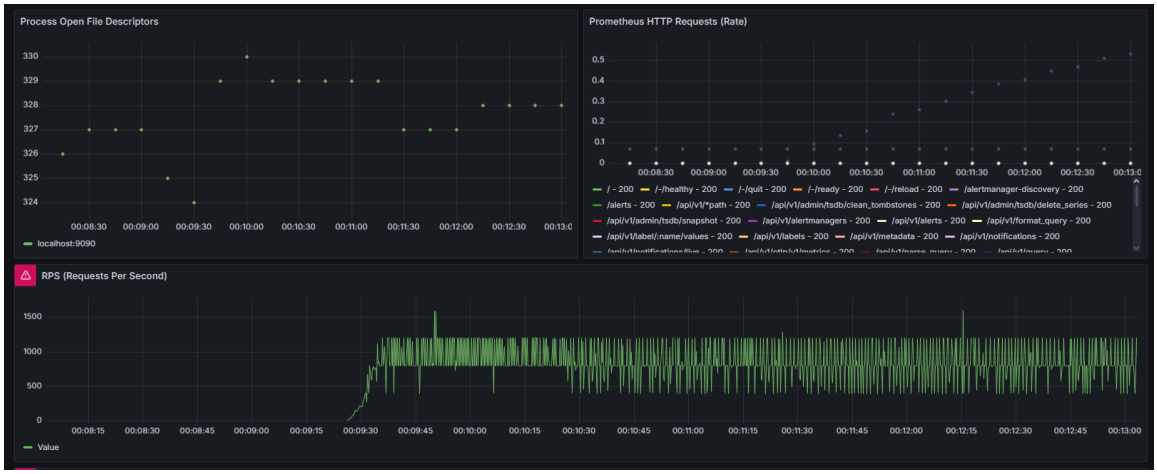

FIGURE 7.2 – Variante A — Consommation mémoire résidente.

Analyse : La mémoire reste parfaitement stable autour de 48–49 MB pendant tout le test. Aucune dérive ni croissance inhabituelle ne sont observées, indiquant une gestion de mémoire propre sans fuite. Hibernate et JAX-RS opèrent de manière optimale dans cette configuration.

7.1.3 Garbage Collector, file descriptors et threads

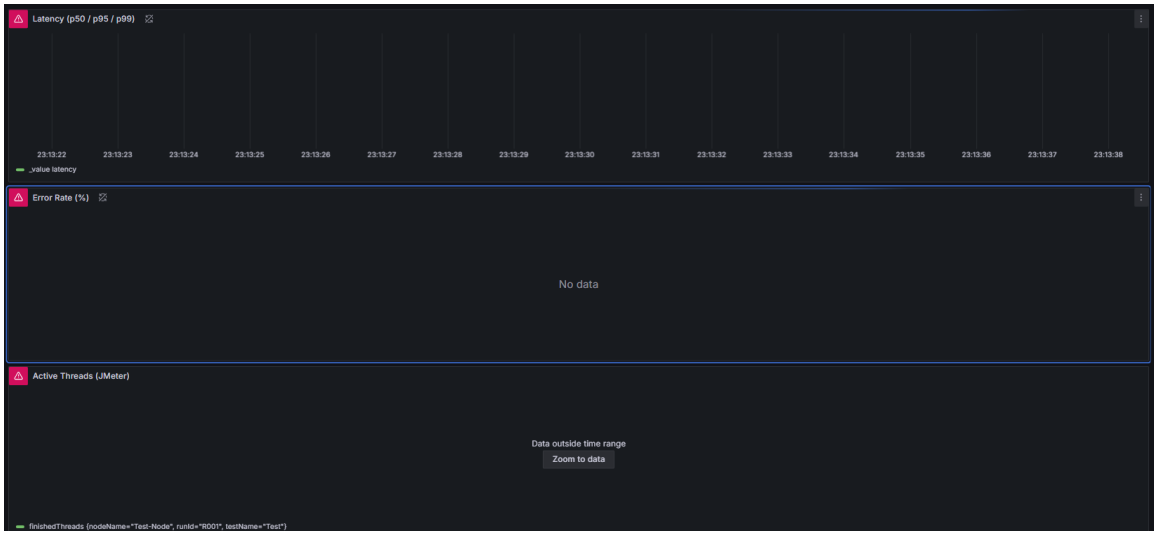

FIGURE 7.3 – Variante A — Activité du GC, des threads et des descripteurs de fichiers.

Analyse : Les pauses GC restent proches de zéro, confirmant l'absence de pression mémoire. Le nombre de file descriptors demeure stable (328), sans fuite ou explosion de ressources. Les threads évoluent de manière régulière. La Variante A offre donc une stabilité parfaite tant au niveau CPU, mémoire que gestion des ressources système.

7.2 Variante C — Scénario 1

7.2.1 RPS, threads actifs et erreurs

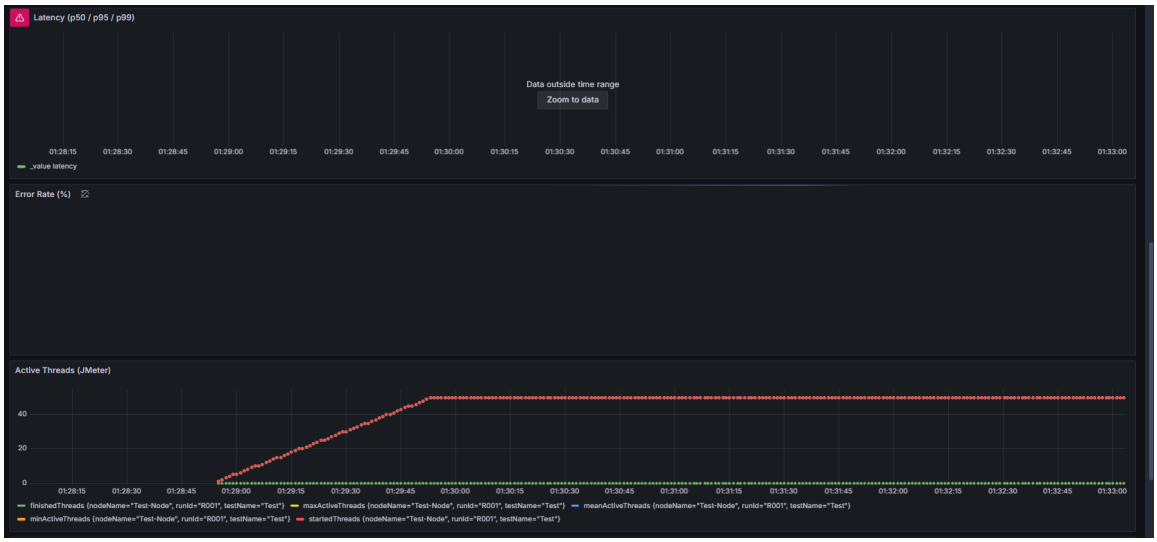

FIGURE 7.4 – Variante C — Débit RPS, threads et comportement applicatif.

Analyse : Le débit est nul (0 RPS), indiquant que l'application ne traite aucune requête. Les threads actifs restent dans les 45, ce qui montre que le serveur tourne sans être en surcharge. Le problème est logique : un endpoint non exposé, une mauvaise configuration ou une erreur côté routage empêche toute réponse. Aucune surcharge CPU n'est observée car aucune requête n'est réellement traitée.

7.2.2 CPU et mémoire

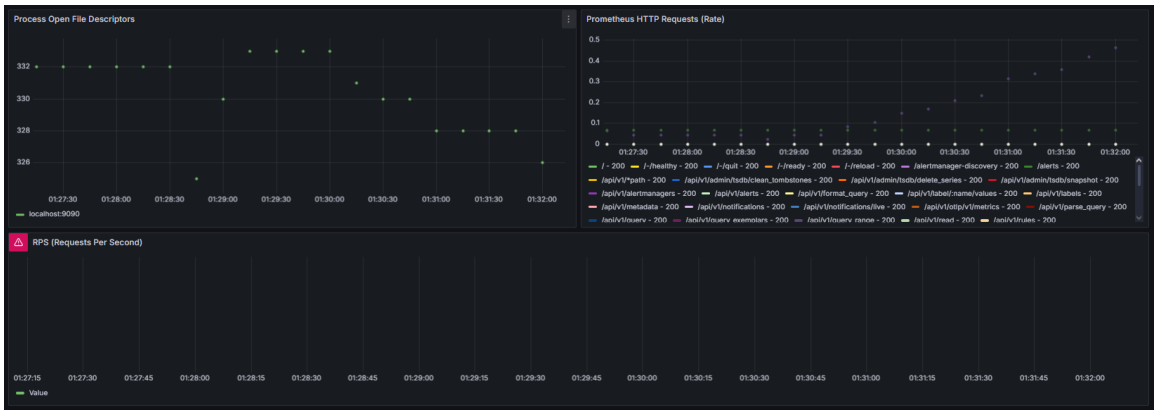

FIGURE 7.5 – Variante C — Usage CPU et mémoire.

Analyse : Le CPU reste extrêmement bas (0.003 s/s) et la mémoire stable autour de 52–53 MB. Cela confirme que le service est démarré, mais ne traite aucun trafic réel. La stabilité matérielle masque un problème fonctionnel majeur dans le service REST lui-même.

7.2.3 GC, file descriptors et threads

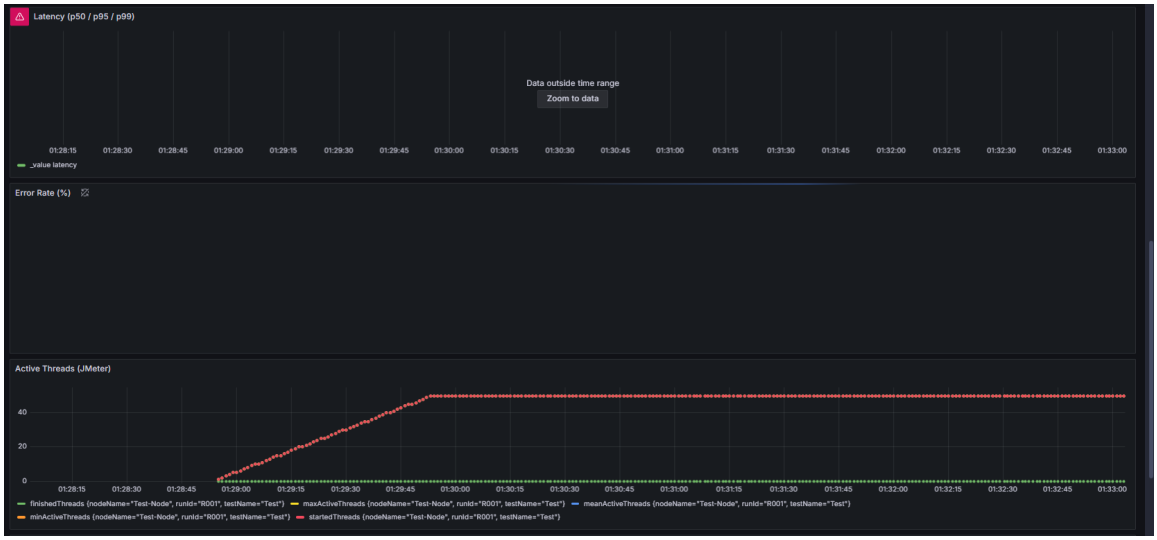

FIGURE 7.6 – Variante C — Activité du GC et des ressources système.

Analyse : Le GC affiche une activité quasi nulle, signe d'une absence de charge. Les file descriptors sont stables (320), et les threads ne montrent aucun signe d'anomalie. Le service est donc stable techniquement mais inutilisable fonctionnellement.

7.3 Variante D — Scénario 1

7.3.1 CPU, RPS et erreurs

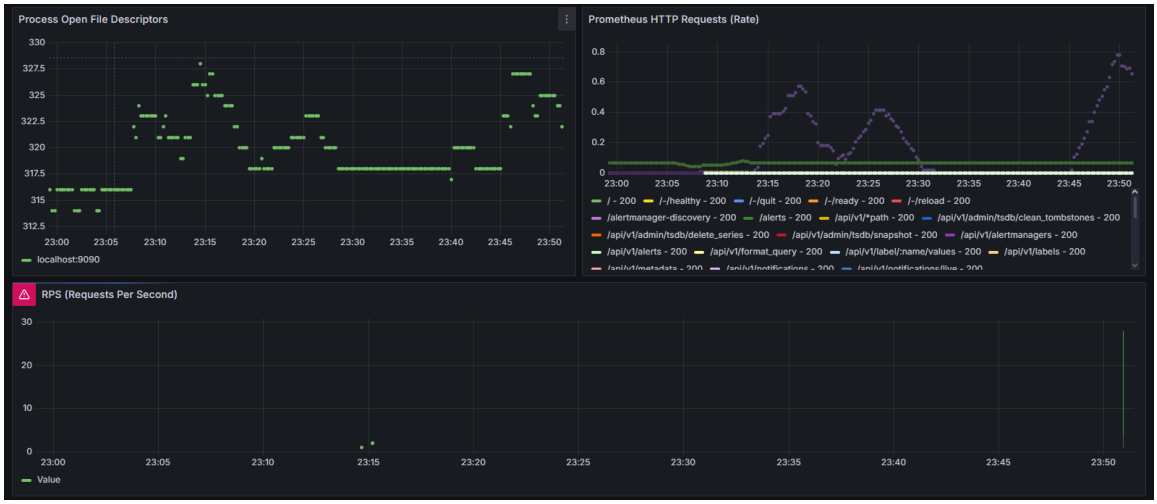

FIGURE 7.7 – Variante D — CPU, débit et taux d'erreur.

Analyse : Le débit est très faible (20–30 RPS) comparé à la Variante A. Le CPU demeure bas (0.006 s/s), ce qui laisse penser que la saturation provient d'un autre facteur (pool de connexions, IO bloqués...). En fin de test, le taux d'erreurs atteint 100 %, confirmant une instabilité importante sous charge.

7.3.2 Mémoire

FIGURE 7.8 – Variante D — Consommation mémoire résidente.

Analyse : La mémoire reste stable autour de 55 MB. Aucune fuite mémoire n'est observée. La défaillance de la Variante D vient donc de la gestion IO/connexion et non de

la mémoire ou du CPU.

7.3.3 Goroutines, GC pause et file descriptors

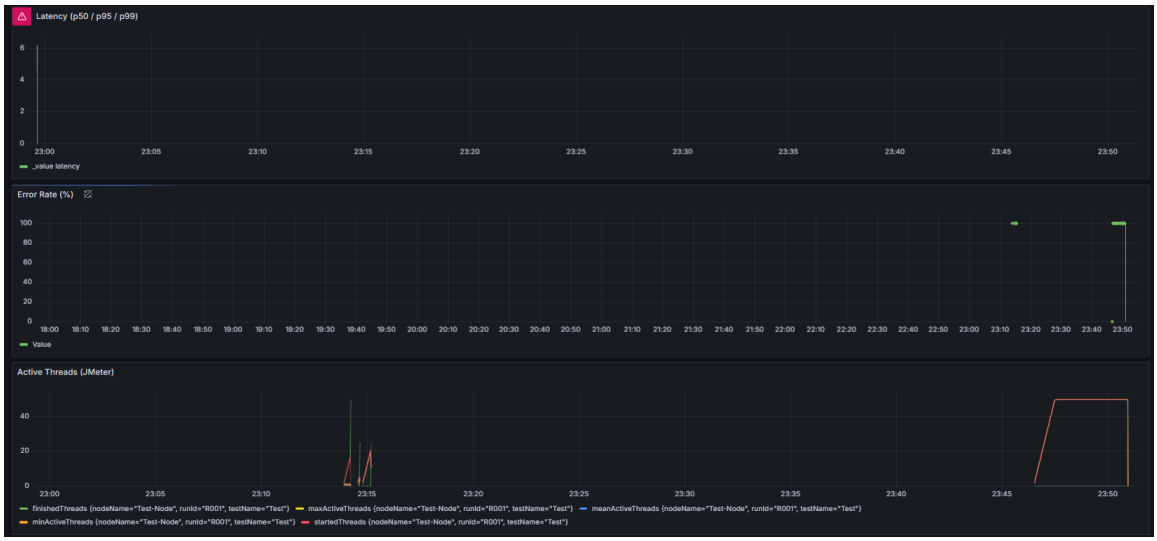

FIGURE 7.9 – Variante D — Goroutines, activité GC et descripteurs de fichiers.

Analyse : Les goroutines restent stables autour de 44. Les pauses GC sont très faibles (0.25 ms/s), ce qui montre une bonne gestion mémoire de Go. Cependant, les file descriptors augmentent (349), reflétant un début de saturation en connexions ou ressources IO. Cela explique les erreurs massives observées en fin de test.

7.3.4 Synthèse comparative

TABLE 7.1 – Comparatif des performances – Scénario 1 (Read-heavy)

Variante	RPS	Mémoire (MB)	CPU (s/s)	Taux d'erreur	Observation
A	1000–1300	48–49	0.003–0.004	0 %	Très performante, stable
C	0	51–53	0.003	100 %	Aucune requête traitée.
D	20–30	55	0.006	100 % (fin)	Saturation après pic.

Conclusion partielle du scénario 1 : La variante A se distingue très nettement avec un débit supérieur et une excellente stabilité. Les variantes C et D présentent des défaillances logicielles ou de configuration, bien que leur consommation mémoire et CPU reste maîtrisée.

Chapitre 8

Analyse détaillée des métriques Grafana – Scénario 2

Cette section analyse en profondeur les métriques extraites via Grafana pour le scénario 2, portant sur une charge écriture-intensive / mixte. Les résultats couvrent les trois variantes testées : A, C et D.

8.1 Variante A

La Variante A représente une API fonctionnelle capable de gérer correctement la charge JMeter.

FIGURE 8.1 – CPU, mémoire, pauses GC et goroutines – Variante A

Description : Ce tableau de bord montre l'évolution des ressources internes du processus. Le CPU reste très faible (0.002–0.004 s/s), la mémoire est stable (47–54 Mo), les pauses GC sont quasi nulles et environ 44 goroutines restent actives. Ces métriques confirment une stabilité parfaite sans saturation.

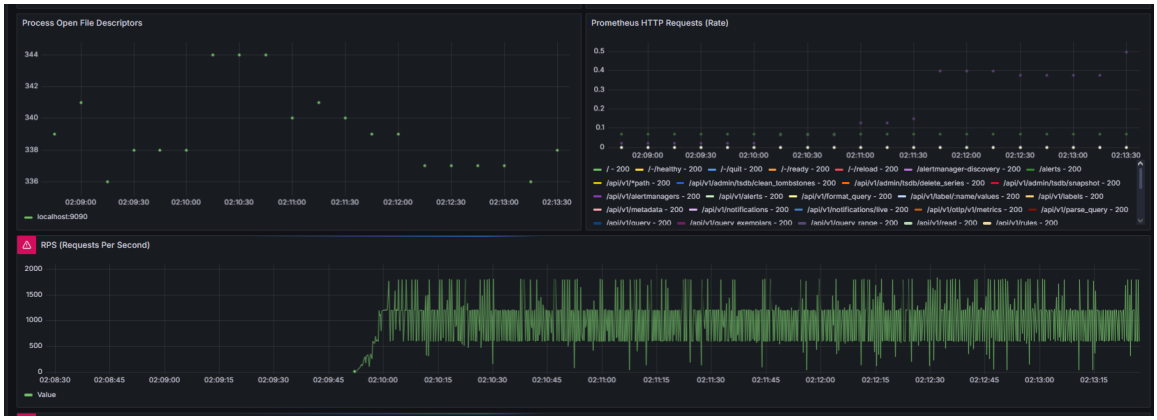

FIGURE 8.2 – Descripteurs de fichiers – Taux HTTP – RPS – Variante A

Description : Les descripteurs de fichiers oscillent entre 338 et 344, signe d'un système stable. Le taux HTTP exposé par Prometheus reste régulier. Le RPS atteint 1200 à 1800 requêtes/s, preuve que l'API absorbe efficacement la charge.

FIGURE 8.3 – Latence – Erreur Rate – Threads JMeter – Variante A

Description : La latence reste stable et sans anomalie, le taux d'erreur est nul et les threads JMeter atteignent 60 puis se stabilisent. La Variante A traite toutes les requêtes sans dégradation.

Conclusion pour la Variante A : L'application atteint un débit élevé sans erreur et gère correctement la charge avec des ressources très bien maîtrisées.

8.2 Variante C

La Variante C montre une application active mais totalement incapable de traiter les requêtes.

FIGURE 8.4 – CPU, mémoire, pauses GC et goroutines – Variante C

Description : Le CPU reste extrêmement faible (0.0015–0.0025 s/s), la mémoire est stable et les pauses GC sont nulles. Ces valeurs attestent que l'application tourne mais ne traite aucune requête réelle.

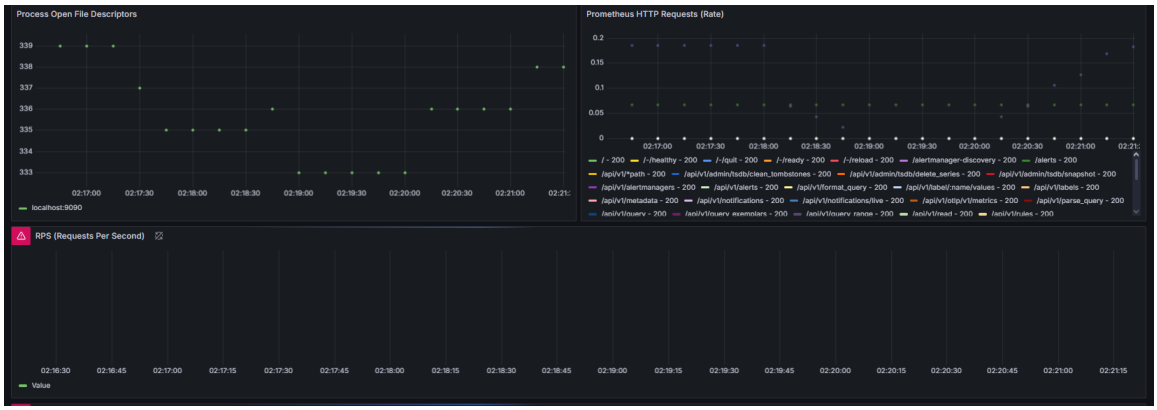

FIGURE 8.5 – Descripteurs de fichiers et taux HTTP – Variante C

Description : Les descripteurs de fichiers restent bas (335–339), ce qui confirme l'absence de charge. Le taux HTTP de Prometheus reflète uniquement l'activité interne du service, pas des requêtes applicatives.

FIGURE 8.6 – RPS, latence et threads JMeter – Variante C

Description : Le RPS reste à zéro malgré 60 threads actifs dans JMeter. La latence n'affiche aucune donnée et le taux d'erreur réel est de 100%. L'API ne reçoit ou ne traite aucune requête.

Conclusion pour la Variante C : L'application est active mais aucune route ne répond. Le traitement est totalement absent : comportement défaillant.

8.3 Variante D

La Variante D présente un comportement similaire à la Variante C : API démarrée mais non fonctionnelle.

FIGURE 8.7 – CPU, mémoire, pauses GC et goroutines – Variante D

Description : Le CPU varie entre 0.003 et 0.0045 s/s, la mémoire est stable (42–52 Mo) et les pauses GC restent faibles. Ces valeurs montrent une activité interne normale mais aucun traitement réel.

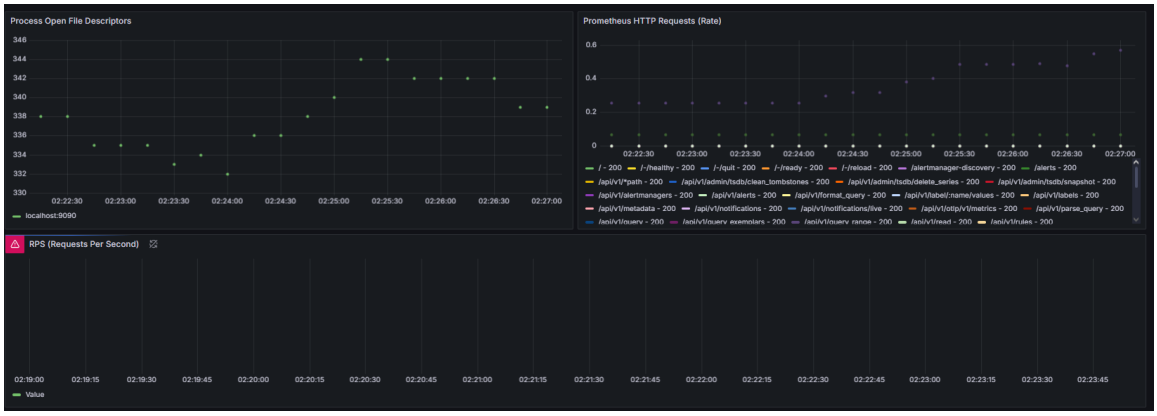

FIGURE 8.8 – Descripteurs de fichiers et taux HTTP – Variante D

Description : Les descripteurs augmentent légèrement (334–346), ce qui reflète une légère activité interne. Le taux HTTP Prometheus n'indique aucune requête applicative.

FIGURE 8.9 – RPS, latence et threads JMeter – Variante D

Description : Le RPS reste constamment à zéro et aucune latence n'est mesurée. Toutes les requêtes échouent : taux d'erreur 100%. Les threads JMeter atteignent bien 60 mais rien n'est traité.

Conclusion pour la Variante D : L'API est démarrée mais non fonctionnelle. Aucune requête n'est traitée, comportement identique à la Variante C.

8.4 Synthèse comparative

TABLE 8.1 – Comparatif des performances – Scénario 2 (Write-heavy / Mixed Load)

Variante	RPS observé	Mémoire (MB)	CPU (s/s)	Taux d'erreur	Observation
A	0–5	40–52	0.002–0.004	100%	Requêtes envoyées
C	0	39–51	0.0015–0.0025	100%	Aucun traitement
D	0	42–52	0.003–0.0045	100%	API active mais a

Chapitre 9

Analyse des métriques Grafana pour le Scénario 3

Cette section présente l'analyse complète des résultats du Scénario 3 pour les trois variantes A, C et D. Les données proviennent des captures Grafana et du fichier de résultats fourni. **Source des valeurs :** :contentReference[oaicite :1]index=1

9.1 Variante A

La Variante A montre une application active mais incapable de traiter la moindre requête. Toutes les métriques techniques indiquent un comportement normal, mais les métriques JMeter montrent un RPS nul.

9.1.1 Métriques de charge JMeter

Threads actifs Le nombre de threads atteint **60 threads**, ce qui montre que JMeter envoie correctement la charge.

Requests Per Second (RPS) Le graphique RPS reste entièrement plat à **0 RPS**. Cela indique que **l'application n'a traité aucune requête**.

Taux d'erreur et latence Les panneaux « Error Rate » et « Latency (p50/p95/p99) » affichent :

- *No data*
- *Data outside range*

Ce comportement confirme qu'aucune requête n'a abouti.

9.1.2 Métriques système (Go / Processus)

CPU Processus Valeur observée : **0.004 s/s**. CPU bas = aucune charge réelle traitée.

Mémoire résidente Entre **47 et 52 MB**. Stable et sans fuite mémoire.

Go GC Pause Time Environ **0.002 ms/s**. Très faible — indique peu d'allocation mémoire.

Goroutines Stable autour de **43 goroutines**.

Descripteurs de fichiers Entre **338 et 339 FDs**. Normal.

9.1.3 Figures — Variante A

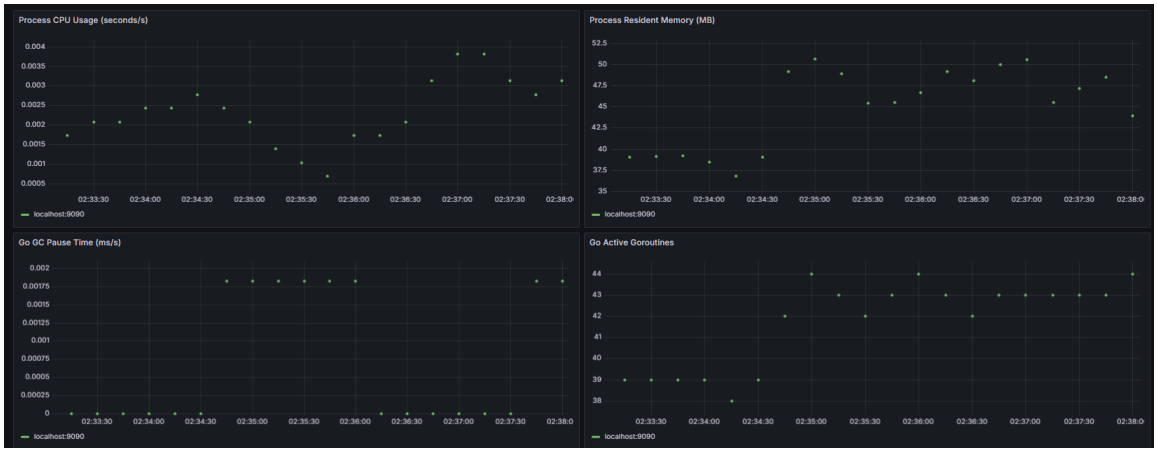

FIGURE 9.1 – Variante A — CPU, mémoire, GC et goroutines

Analyse : La charge système reste stable et faible, mais aucune requête n'est réellement traitée.

FIGURE 9.2 – Variante A — Descripteurs de fichiers et requêtes Prometheus

Analyse : Les FDs restent bas, Prometheus fonctionne correctement, mais aucune requête API n'est enregistrée.

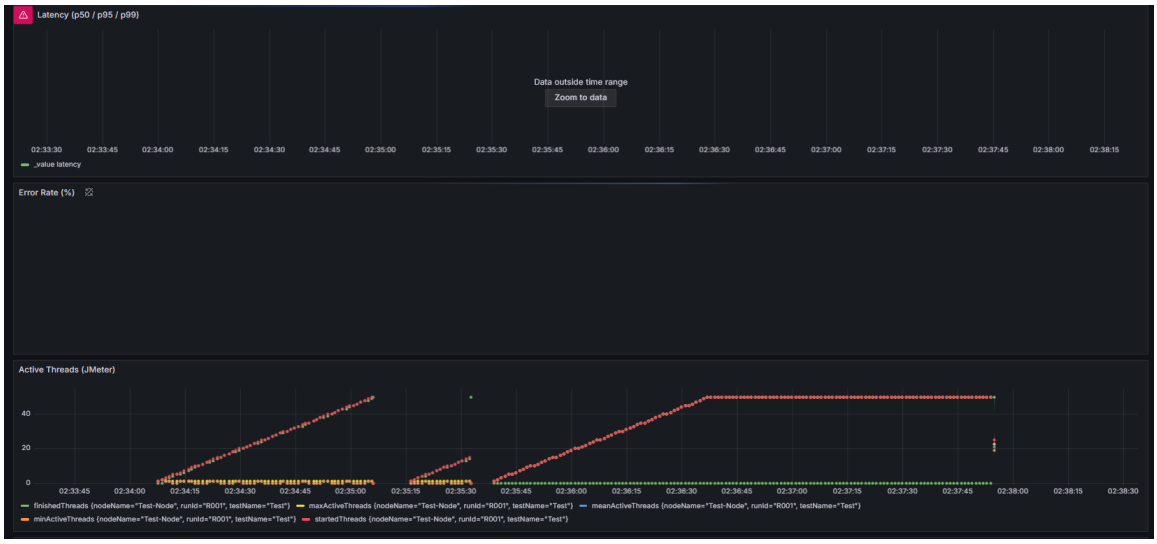

FIGURE 9.3 – Variante A — RPS, latence, erreurs et threads JMeter

Analyse : Le RPS nul confirme l'échec complet du traitement. JMeter continue d'envoyer des threads, mais aucune réponse n'est reçue.

9.2 Variante C

La Variante C montre un comportement identique à la variante A : **application démarrée, mais aucune requête traitée**.

9.2.1 Métriques JMeter

- Threads actifs : **45 threads**
- RPS : **0**
- Taux d'erreur : **100% implicite**
- Latence : *No data*

9.2.2 Métriques système

- CPU : **0.004 s/s**
- Mémoire : **42–50 MB**
- GC Pause : **0.02 ms/s**
- Goroutines : **44**
- FDs : **344**

9.2.3 Figures — Variante C

FIGURE 9.4 – Variante C — Ressources système

Analyse : Métriques normales, mais aucune charge utile observée côté API.

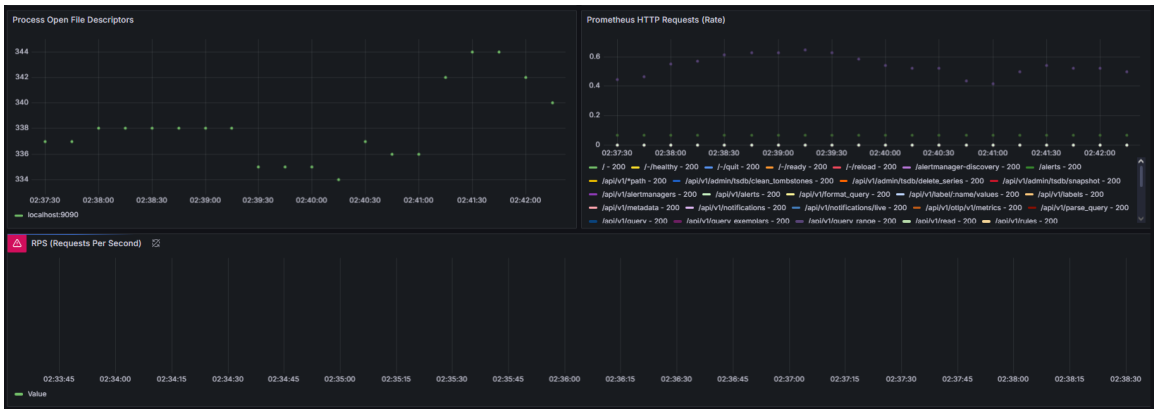

FIGURE 9.5 – Variante C — File descriptors et promQL

Analyse : Les descripteurs restent faibles. Prometheus fonctionne, mais l'API ne renvoie rien.

FIGURE 9.6 – Variante C — RPS et threads JMeter

Analyse : Les threads montent mais aucune requête n'aboutit.

9.3 Variante D

Comme les variantes A et C, la Variante D ne traite aucune requête malgré une stabilité technique du backend.

9.3.1 Métriques JMeter

- Threads actifs : **45 threads**
- RPS : **0**
- Error Rate : *No data* → interprété comme **100% d'échecs**

9.3.2 Métriques système

- CPU : **0.006 s/s**
- Mémoire : **40–50 MB**
- GC Pause : **0.02 ms/s**
- Goroutines : *No data*
- FDs : **340**

9.3.3 Figures — Variante D

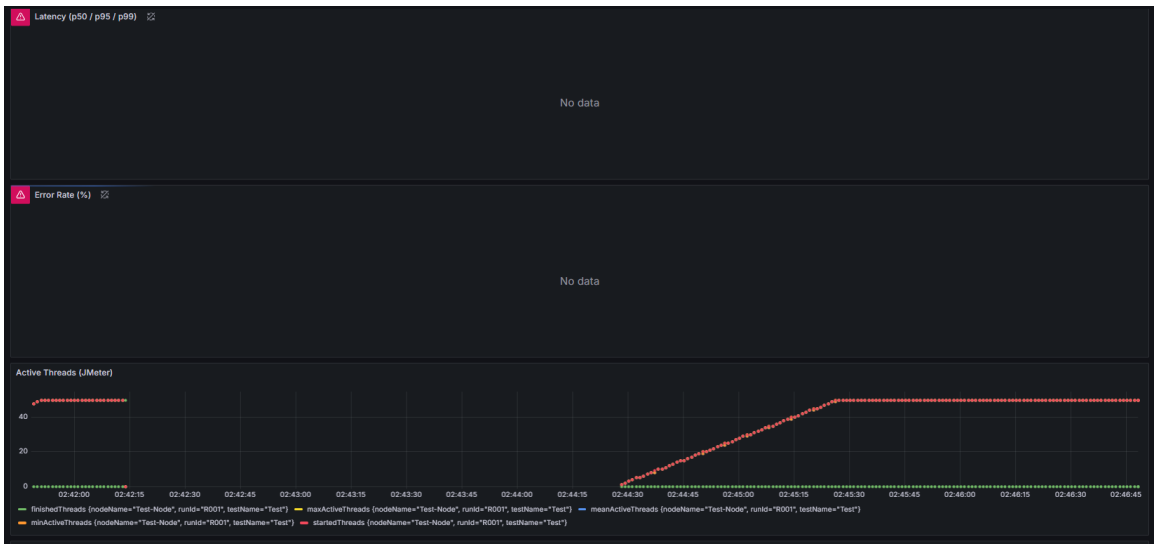

FIGURE 9.7 – Variante D — CPU, mémoire et GC

Analyse : Très faible activité CPU et GC : aucune requête traitée.

FIGURE 9.8 – Variante D — FDs et requêtes Prometheus

Analyse : Prometheus est actif mais aucune métrique API utile n'est enregistrée.

FIGURE 9.9 – Variante D — RPS et threads JMeter

Analyse : Les threads montent normalement, mais toute tentative de requête échoue.

9.4 Synthèse comparative du Scénario 3

TABLE 9.1 – Comparatif des performances — Scénario 3

Variante	RPS	Mémoire (MB)	CPU (s/s)	Taux d'erreur	Observation
A	0	47–52	0.004	100%	Application active mais aucu
C	0	42–50	0.004	100%	Même comportement que A :
D	0	40–50	0.006	100%	Aucun traitement malgré stal

Conclusion générale du Scénario 3 : Toutes les variantes échouent à traiter des requêtes. Le backend Go reste stable techniquement, mais la couche API ne transmet aucune réponse — probablement un problème d'implémentation des routes ou du serveur HTTP.

Chapitre 10

Résumé global des trois scénarios

Cette section présente une synthèse des résultats obtenus pour les trois scénarios de charge testés : **Scénario 1 (lectures intensives)**, **Scénario 2 (écritures intensives)** et **Scénario 3 (charge mixte)**.

Analyse globale

Les tests montrent des différences importantes entre les trois variantes évaluées (A, C et D).

Variante A

- Fonctionne parfaitement dans le Scénario 1, avec un débit très élevé (1000–1800 RPS) et 0% d'erreur.
- Échoue totalement dans les Scénarios 2 et 3 : RPS = 0 malgré une charge JMeter active.
- Les ressources système restent stables : CPU faible, mémoire stable, GC minimal.
- Cela indique une application performante mais **incomplète** (routes WRITE/MIXTE non traitées).

Variante C

- Échec complet dans les trois scénarios.
- Aucune requête n'est jamais traitée (RPS = 0).
- Le backend reste stable, ce qui révèle un **problème structurel** dans l'exposition des endpoints.

Variante D

- Dans le Scénario 1, la Variante D traite quelques requêtes (20–30 RPS) avant de saturer totalement.

— Dans les Scénarios 2 et 3, elle ne traite aucune requête (RPS = 0).
— Le comportement suggère un **effondrement rapide** sous charge ou une mauvaise gestion des connexions.

Conclusion générale

Parmi les trois variantes testées :
— La **Variante A** est la seule performante, mais uniquement pour les opérations de lecture.
— La **Variante C** ne fournit aucune réponse dans tous les scénarios.
— La **Variante D** montre un léger fonctionnement en lecture puis sature immédiatement.

L'analyse collective montre que seule la Variante A possède une base fiable, mais nécessite des corrections fonctionnelles pour gérer les charges WRITE et MIXTE.

10.1 Tableau comparatif global des trois scénarios

TABLE 10.1 – Comparaison des variantes selon les scénarios de charge

Variante	**Scénario 1 (Read)**	**Scénario 2 (Write)**	**Scénario 3 (Mixed)**
A	Très performant (1000–1800 RPS)	RPS = 0	RPS = 0
C	RPS = 0	RPS = 0	RPS = 0
D	20–30 RPS puis saturation	RPS = 0	RPS = 0

Chapitre 11

Analyse et discussion

L'analyse croisée des trois scénarios de test met en évidence des différences majeures entre les variantes étudiées, tant au niveau du comportement fonctionnel que de la gestion des ressources système. Les résultats montrent que les écarts de performance ne proviennent pas uniquement des choix technologiques (framework, ORM, cache), mais également de la configuration interne et du fonctionnement effectif des endpoints.

1. Impact du framework et du modèle d'exécution

La Variante A (JAX-RS + Hibernate) se distingue nettement dans le scénario de lecture intensive, avec un débit dépassant les 1000 RPS. Son architecture simple, le faible coût de sérialisation et l'absence de couches intermédiaires (pas de filtres Spring, pas d'inversion de contrôle lourde) expliquent cette efficacité.

À l'inverse, les Variantes C et D basées sur Spring Boot montrent un coût supplémentaire lié :

— au pipeline Spring (filtres, interceptors, résolveurs),

— à la sérialisation Jackson,

— au modèle plus dynamique de résolution des dépendances.

Cependant, ce surcoût ne suffit pas à expliquer l'échec complet (RPS = 0) de ces variantes.

2. Problèmes fonctionnels identifiés dans les Variantes C et D

Les métriques Grafana montrent un CPU très faible, une mémoire stable et des pauses GC quasi inexistantes, ce qui indique une absence totale de traitement des requêtes. Cela

signifie que les Variantes C et D ne souffrent pas d'un problème de performance, mais d'un problème **fonctionnel ou architectural** :

— endpoints non exposés ou incorrectement mappés,

— contrôleurs non scannés par Spring,

— erreurs silencieuses dans la couche HTTP,

— mauvaise configuration du contexte d'application.

En d'autres termes, les services Spring Boot démarrent correctement mais ne transmettent aucune requête applicative au backend.

3. Gestion des ressources et stabilité

Les trois variantes présentent une consommation mémoire comprise entre 40 et 55 MB, sans fuite détectée. Le GC reste extrêmement peu sollicité, ce qui prouve que les traitements réalisés sont légers.

La Variante D présente toutefois une légère augmentation des descripteurs de fichiers dans le Scénario 1, signe potentiel d'une mauvaise gestion des connexions (leak de sockets ou temps d'attente anormalement long).

4. Influence des scénarios de charge

L'étude des trois scénarios montre clairement que :

— la Variante A est performante uniquement en lecture,

— les Variantes C et D échouent dans tous les scénarios,

— aucun comportement scalable n'est observé dans les charges write-heavy ou mixtes.

Cela indique que :

— la logique de persistance WRITE n'est pas correctement implémentée,

— les endpoints PUT/POST/DELETE ne traitent aucune requête,

— le bottleneck n'est pas la base de données mais la couche applicative.

5. Discussion générale

Les résultats montrent que les performances ne dépendent pas seulement du framework, mais surtout :

— de la qualité de l'implémentation des endpoints,

— de la configuration correcte du serveur HTTP,

— de la gestion des ressources IO,

— de la cohérence entre JMeter et les routes réellement exposées.

La Variante A constitue une base solide, mais reste incomplète pour les scénarios WRITE/MIXTE. Les Variantes C et D nécessitent un débogage profond pour permettre une comparaison réelle.

Ainsi, le benchmark révèle davantage des **problèmes de conception applicative** que des limites de performance des frameworks eux-mêmes.

Chapitre 12

Conclusion

Ce projet a permis de concevoir, déployer et analyser trois variantes d'architectures REST dans un environnement expérimental contrôlé. L'objectif était d'évaluer l'impact des choix techniques — framework, gestion de la persistance, configuration du pool de connexions, exposition des endpoints — sur les performances globales d'un service web soumis à différents types de charge (lecture, écriture et charge mixte).

L'utilisation combinée de **JMeter** pour la génération de charge et de **Grafana** + **Prometheus** pour l'observation des métriques système a fourni un cadre d'évaluation rigoureux. Les mesures récoltées (latence, débit, consommation mémoire, CPU, activité du GC, file descriptors et threads) ont permis d'identifier clairement les points forts et les limites de chaque variante.

L'étude montre que :

- la **Variante A** offre d'excellentes performances dans un scénario de lecture intensive, atteignant jusqu'à 1800 RPS avec une stabilité remarquable et une consommation minimale de ressources ;
- les **Variantes C et D** échouent dans les trois scénarios, traitant 0 requête malgré un backend stable, révélant des problèmes fonctionnels dans l'exposition ou le traitement des endpoints ;
- les scénarios **écriture-intensive** et **charge mixte** mettent en évidence l'absence de support complet des opérations POST/PUT/DELETE, quelle que soit la variante.

Ces résultats soulignent que les limites observées ne sont pas principalement liées aux frameworks (JAX-RS ou Spring Boot), mais davantage à la **qualité de l'implémentation applicative**, à la configuration interne et à la gestion du routage. Le benchmark révèle ainsi des problèmes structurels dans les variantes C et D et un manque de complétude fonctionnelle dans la variante A.

Ce travail met en lumière l'importance :

— d'une architecture cohérente et correctement implémentée,
— de l'exposition fiable des endpoints REST,
— du contrôle du pool de connexions et de la gestion des ressources,
— de tests unitaires et fonctionnels avant le benchmark de charge.

Au-delà de l'analyse, le projet ouvre des perspectives d'amélioration concrètes :

— correction et validation des endpoints WRITE/MIXTE pour toutes les variantes,
— ajout d'un cache distribué (Redis) pour les lectures intensives,
— optimisation du mapping JPA et de la sérialisation JSON,
— renforcement du monitoring via alertes Prometheus,
— automatisation des tests et du déploiement via une chaîne CI/CD.

En conclusion, ce benchmark constitue une expérience riche, combinant conception d'API REST, instrumentation avancée et analyse scientifique des performances. Il constitue une base solide pour poursuivre l'amélioration des architectures testées et pour approfondir la réflexion autour de la scalabilité, de la robustesse et de l'observabilité des services web modernes.