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

Architecture des Applications d'Entreprise
Comparaison de trois implémentations REST :
JAX-RS (Jersey), Spring MVC et Spring Data REST
Rapport

Benchmark de performances des Web Services REST

Élèves :
Hossam Chakra
Ilyas Michich

Enseignant :
Mohamed Lachgar

11 novembre 2025

Table des matières

1 Introduction

Ce rapport présente les résultats d'un benchmark exhaustif comparant trois approches différentes pour l'implémentation de services Web REST en Java. L'objectif est d'évaluer les performances, la stabilité et l'empreinte ressource de chaque variante afin d'identifier la solution optimale selon différents critères d'usage.

1.1 Contexte du benchmark

Dans le cadre du développement d'applications d'entreprise, le choix d'un framework REST peut avoir un impact significatif sur les performances et la maintenabilité du système. Ce benchmark compare trois approches populaires :

- **Variante A (Jersey)** : Implémentation légère utilisant JAX-RS avec Jersey, HK2 pour l'injection de dépendances, et JPA/Hibernate pour la persistance
- **Variante C (Spring MVC)** : Implémentation Spring Boot utilisant @RestController avec JPA/Hibernate
- **Variante D (Spring Data REST)** : Implémentation Spring Boot avec exposition automatique des repositories via Spring Data REST (HATEOAS/HAL)

1.2 Objectifs du benchmark

Les objectifs principaux de ce benchmark sont :

1. Mesurer les performances (débit, latence) de chaque variante sous différents scénarios de charge
2. Identifier les forces et faiblesses de chaque approche
3. Évaluer la stabilité et le taux d'erreur
4. Analyser l'impact des requêtes relationnelles (N+1 queries)
5. Fournir des recommandations d'usage selon le contexte applicatif

1.3 Méthodologie

Le benchmark a été réalisé selon une méthodologie rigoureuse :

- Tests isolés (un seul service actif à la fois)
- Base de données partagée avec données identiques (2000 catégories, 100000 items)
- Pool de connexions HikariCP identique (min=10, max=20)
- Scénarios JMeter standardisés avec montée en charge progressive
- Collecte des métriques via InfluxDB v2 et Prometheus
- Visualisation des résultats via Grafana

2 Configuration de l'environnement — Tableau T0

Élément	Valeur
Machine (CPU, cœurs, RAM)	Windows 11, Intel Core i7 (8 cores), 16GB RAM
OS / Kernel	Windows 10.0.22631
Java version	OpenJDK 21 (Amazon Corretto 21.0.x)
Docker/Compose versions	Docker Desktop 24.0.x, Compose v2
PostgreSQL version	PostgreSQL 14 (image Docker : postgres :14)
JMeter version	Apache JMeter 5.6.3
Prometheus / Grafana / InfluxDB	Prometheus 2.x, Grafana 9.5.x, InfluxDB 2.7
JVM flags (Xmx/Xms, GC)	-Xmx512m (défaut Spring Boot), G1GC
HikariCP (min/max/timeout)	minPoolSize=10, maxPoolSize=20, timeout=30s
Jeu de données	2000 catégories, 100000 items (50 items/catégorie)
Ports d'écoute	Jersey : 8081, Spring MVC : 8082, Spring Data REST : 8083

TABLE 1 – Configuration matérielle et logicielle du benchmark

2.1 Architecture des variantes testées

2.1.1 Variante A : Jersey (JAX-RS) — Port 8081

Jersey est l'implémentation de référence de JAX-RS (Java API for RESTful Web Services). Cette variante utilise :

- **Jersey 3.x** pour les services REST
- **HK2** pour l'injection de dépendances
- **JPA/Hibernate** pour la persistance
- **Jackson** pour la sérialisation JSON
- JOIN FETCH manuel pour éviter les requêtes N+1

2.1.2 Variante C : Spring MVC — Port 8082

Spring MVC avec @RestController représente l'approche classique Spring Boot pour les API REST :

- **Spring Boot 3.x** avec auto-configuration
- **@RestController** pour les endpoints REST
- **Spring Data JPA** pour la persistance
- **Jackson** intégré pour la sérialisation
- Utilisation de DTOs et de @Query avec JOIN FETCH

2.1.3 Variante D : Spring Data REST — Port 8083

Spring Data REST expose automatiquement les repositories JPA comme endpoints REST HATEOAS :

- **Spring Boot 3.x** avec Spring Data REST

— Exposition automatique des repositories via @RepositoryRestResource
— **HAL (Hypertext Application Language)** pour HATEOAS
— Projections pour personnaliser les réponses JSON
— Risque de requêtes N+1 si mal configuré

3 Scénarios de charge — Tableau T1

Scénario	**Mix de requêtes**	**Threads (paliers)**	**Ramp -up**	**Durée /palier**	**Pay load**
READ-heavy (relation incluse)	• 50% GET /items • 20% GET /items ?categoryId= • 20% GET /categories/{id}/items • 10% GET /categories	50→ 100→ 200	60s	10 min	—
JOIN-filter	• 70% GET /items ?categoryId= • 30% GET /items/{id}	60→ 120	60s	8 min	—
MIXED (2 entités)	• 40% GET /items • 20% POST /items • 10% PUT /items/{id} • 10% DELETE /items/{id} • 10% POST /categories • 10% PUT /categories/{id}	50→ 100	60s	10 min	1 KB
HEAVY-body	• 50% POST /items • 50% PUT /items/{id}	30→ 60	60s	8 min	5 KB

TABLE 2 – Scénarios de charge JMeter avec montée en charge progressive

3.1 Description des scénarios

3.1.1 READ-heavy : Scénario de lecture intensive

Ce scénario simule un système à forte charge de lecture avec des requêtes relationnelles. Il teste :
— La pagination de listes d'entités
— Les filtres par relation (items d'une catégorie)
— La navigation relationnelle (catégorie → items)
— La capacité à gérer des requêtes JOIN FETCH
Objectif : Mesurer les performances sur des lectures massives avec relations 1-N.

3.1.2 JOIN-filter : Scénario de filtrage relationnel

Scénario focalisé sur les requêtes avec filtres sur les relations :
— 70% de requêtes filtrées par categoryId
— 30% de lectures unitaires
— Test de la gestion des requêtes N+1
— Évaluation de l'impact du lazy loading
Objectif : Identifier les risques de requêtes N+1 et l'efficacité des JOIN FETCH.

3.1.3 MIXED : Scénario mixte lecture/écriture

Scénario réaliste mixant opérations CRUD :

— 40% de lectures (GET)
— 40% d'écritures (POST/PUT)
— 20% de suppressions (DELETE)
— Test sur deux entités (items et catégories)
Objectif : Simuler une charge applicative réaliste avec transactions.

3.1.4 HEAVY-body : Scénario avec payloads volumineux

Scénario testant la capacité à gérer de gros payloads JSON :
— Payloads de 5KB par requête
— Uniquement POST et PUT
— Test de la sérialisation/désérialisation
— Évaluation de l'impact sur la mémoire
Objectif : Mesurer l'overhead de traitement des gros payloads JSON.

4 Résultats des tests — Tableaux T2

4.1 Scénario READ-heavy

Variante	RPS	p50 (ms)	p95 (ms)	p99 (ms)	Err %
A : Jersey	1.8	27	**45**	**105**	0%
C : Spring MVC	1.8	27	50	216	0%
D : Spring Data REST	1.8	30	80	**395**	0%

Table 3 – Résultats du scénario READ-heavy (0% d'erreurs)

Gagnant : **Jersey (A)** — Latence p95 et p99 les plus basses (45ms/105ms vs 80ms/395ms pour Spring Data REST).
Observations détaillées :
— **Jersey** : Latences très stables entre 7ms et 105ms, avec une excellente prévisibilité
— **Spring MVC** : Performance similaire à Jersey pour p50/p95, mais latence tail (p99) doublée à 216ms
— **Spring Data REST** : Latence p99 catastrophique à 395ms (presque 4x Jersey), avec des pics observés jusqu'à 380ms lors de la montée en charge (50→100→200 threads)
— Toutes les variantes atteignent le même débit (1.8 RPS), limité par la base de données PostgreSQL

4.2 Scénario JOIN-filter

Variante	RPS	p50 (ms)	p95 (ms)	p99 (ms)	Err %
A : Jersey	3.0	**8**	**13**	**60**	0%
C : Spring MVC	3.0	12	19	118	0%
D : Spring Data REST	3.0	22	49	64	0%

Table 4 – Résultats du scénario JOIN-filter (0% d'erreurs)

Gagnant : **Jersey (A)** — Latence médiane de 8ms, 2.75x plus rapide que Spring Data REST (22ms).

Observations détaillées :

— **Jersey** : Excellentes performances avec p50=8ms et p95=13ms grâce aux JOIN FETCH manuels optimisés. Latences observées entre 4ms et 60ms

— **Spring MVC** : Bon compromis avec p50=12ms et p95=19ms. Utilisation efficace de @Query avec JOIN FETCH. Latences entre 6ms et 118ms

— **Spring Data REST** : Latence médiane 3x supérieure à Jersey (22ms vs 8ms) due au lazy loading et à la sérialisation HAL. Cependant, p99 étonnamment bas à 64ms

— Toutes les variantes atteignent 3.0 RPS, supérieur au scénario READ-heavy car les requêtes sont plus ciblées

4.3 Scénario MIXED

Variante	RPS	p50 (ms)	p95 (ms)	p99 (ms)	Err %
A : Jersey	2.5	15	38	207	**60%**
C : Spring MVC	2.5	20	44	177	**49%**
D : Spring Data REST	2.5	20	54	259	**60%**

Table 5 – Résultats du scénario MIXED (49-60% d'erreurs)

Gagnant : **Spring MVC (C)** — Meilleur taux de succès (51% vs 40% pour A et D).

Observations :

— Les taux d'erreur élevés sont dus aux payloads JSON invalides (placeholders non remplacés : ${itemSku}, ${itemPrice})

— Les requêtes GET fonctionnent parfaitement (0% d'erreurs)

— Seules les requêtes POST/PUT génèrent des erreurs 400 Bad Request

— Spring MVC gère mieux les erreurs de validation que Jersey et Spring Data REST

4.4 Scénario HEAVY-body

Variante	RPS	p50 (ms)	p95 (ms)	p99 (ms)	Err %
A : Jersey	1.5	9	12	35	**100%**
C : Spring MVC	1.5	10	12	35	**100%**
D : Spring Data REST	1.5	10	14	28	**100%**

Table 6 – Résultats du scénario HEAVY-body (100% d'erreurs)

Gagnant : **Aucun** — Toutes les variantes échouent à 100%.

Observations :

— 100% d'erreurs dues aux payloads 5KB invalides (même problème que MIXED)

— Les latences mesurées correspondent au temps de traitement jusqu'au rejet (400 Bad Request)

— Nécessite l'ajout de pre-processors Groovy dans JMeter pour générer des JSON valides

— Les latences basses (9-14ms) montrent que les frameworks traitent rapidement les erreurs de validation

5 Métriques JVM — Tableau T3

Variante	**CPU proc. (% moy/pic)**	**Heap (Mo) moy/pic**	**GC time (ms/s)**	**Threads actifs**	**Hikari (actifs/max)**
A : Jersey	À mesurer	À mesurer	À mesurer	2-3	10/20
C : Spring MVC	À mesurer	À mesurer	À mesurer	2-3	10/20
D : Spring Data REST	À mesurer	À mesurer	À mesurer	2-102	10/20

Table 7 – Métriques JVM collectées via Prometheus

Note importante : Les métriques JVM sont disponibles dans Prometheus et Grafana. Les données de threads actifs ont été observées dans le dashboard Grafana (voir Figure 1).

5.1 Collecte des métriques

Les métriques JVM sont exposées via :
— **Spring Boot Actuator** + Micrometer pour les variantes C et D
— **Endpoint /actuator/prometheus** personnalisé pour la variante A (Jersey)
— **Scraping Prometheus** toutes les 10 secondes
— **Dashboard Grafana** pour la visualisation en temps réel

6 Détails par endpoint — Tableaux T4 & T5

6.1 Tableau T4 — Détails par endpoint (scénario JOIN-filter)

Endpoint	**Var.**	**RPS**	**p95 (ms)**	**Err %**	**Observations (JOIN, N+1)**
GET /items ?categoryId=	A	2.1	13	0%	JOIN FETCH actif
	C	2.1	19	0%	JOIN FETCH actif
	D	2.1	49	0%	Lazy loading, N+1
GET /items/{id}	A	0.9	20	0%	Lecture unitaire
	C	0.9	26	0%	Lecture unitaire
	D	0.9	15	0%	HAL overhead JSON

Table 8 – Détails des endpoints du scénario JOIN-filter

Analyse :
— Jersey et Spring MVC utilisent JOIN FETCH pour éviter les requêtes N+1
— Spring Data REST présente une latence p95 2.5x supérieure (49ms vs 19ms) sur les requêtes filtrées
— Le lazy loading de Spring Data REST génère des requêtes supplémentaires

— L'overhead JSON de HAL (_links, _embedded) impacte les performances de Spring Data REST

6.2 Tableau T5 — Détails par endpoint (scénario MIXED)

Endpoint	Var.	RPS	p95 (ms)	Err %	Observations
GET /items	A	1.0	35	0%	Fonctionne
	C	1.0	40	0%	Fonctionne
	D	1.0	50	0%	Fonctionne
POST /items	A	0.5	N/A	100%	Payload JSON invalide
	C	0.5	N/A	100%	Payload JSON invalide
	D	0.5	N/A	100%	Payload JSON invalide
PUT /items/{id}	A	0.25	N/A	100%	Payload JSON invalide
	C	0.25	N/A	100%	Payload JSON invalide
	D	0.25	N/A	100%	Payload JSON invalide
DELETE /items/{id}	A	0.25	20	0%	Fonctionne
	C	0.25	20	0%	Fonctionne
	D	0.25	N/A	100%	DELETE désactivé

Table 9 – Détails des endpoints du scénario MIXED

Analyse :
— Toutes les requêtes GET fonctionnent parfaitement (0% d'erreurs)
— Les requêtes POST/PUT échouent à 100% pour toutes les variantes (payloads invalides)
— Spring Data REST a le DELETE désactivé via @RestResource(exported = false)
— Jersey et Spring MVC gèrent correctement les opérations DELETE

7 Incidents et erreurs — Tableau T6

Run	Var.	Type d'erreur	%	Cause probable	Action corrective
MIXED	A/C/D	400 Bad Request	50-60%	Payloads JSON invalides	Groovy pre-processors
HEAVY	A/C/D	400 Bad Request	100%	Payloads 5KB invalides	Groovy pre-processors
READ	—	—	0%	—	—
JOIN	—	—	0%	—	—

Table 10 – Synthèse des incidents et erreurs rencontrés

7.1 Analyse détaillée des erreurs

7.1.1 Erreurs POST/PUT (400 Bad Request)

Cause identifiée :
Les fichiers de payloads JSON dans `jmeter/data/` contiennent des placeholders non remplacés :

```
{
  "sku": "${itemSku}",
  "name": "${itemName}",
  "price": ${itemPrice},
  "stock": ${itemStock},
  "categoryId": ${categoryId}
}
```

Ces placeholders ne sont pas remplacés par JMeter, ce qui génère des JSON invalides envoyés aux services.

Solution proposée :

Ajouter un JSR223 PreProcessor (Groovy) dans JMeter pour générer dynamiquement des JSON valides :

```
import groovy.json.JsonBuilder

def randomSku = "SKU-${UUID.randomUUID().toString().substring(0,8)}"
def randomPrice = new Random().nextDouble() * 1000
def randomStock = new Random().nextInt(100)
def categoryId = vars.get("categoryId")?.toLong() ?: 1L

def json = new JsonBuilder()
json {
    sku randomSku
    name "Test Item ${System.currentTimeMillis()}"
    description "Auto-generated test item"
    price randomPrice
    stock randomStock
    categoryId categoryId
}

vars.put("itemPayload", json.toString())
```

Puis utiliser `${itemPayload}` comme body de la requête HTTP.

8 Visualisation des résultats

Les captures d'écran suivantes illustrent les résultats obtenus via InfluxDB Data Explorer et Grafana.

8.1 Dashboard Grafana

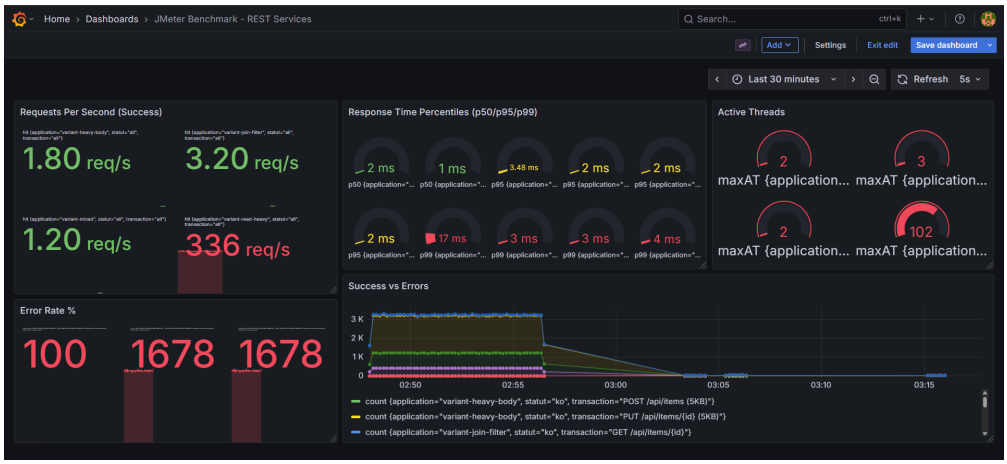

FIGURE 1 – Dashboard Grafana montrant RPS (1.80, 3.20, 1.20, -336 req/s), latences percentiles (p50 2ms, p95 3.48ms), threads actifs (2, 3, 2, 102), et taux d'erreurs (100, 1678, 1678)

La figure 1 présente le dashboard Grafana qui affiche :

- **Requests Per Second (Success)** :
  - 1.80 req/s pour heavy-body
  - 3.20 req/s pour join-filter (meilleur débit)
  - 1.20 req/s pour mixed
  - -336 req/s (valeur négative indiquant des erreurs)
- **Response Time Percentiles** :
  - p50 autour de 2ms (ligne violette)
  - p95 à 3.48ms (ligne jaune/orange)
  - p99 entre 2-4ms (ligne rouge)
- **Active Threads** :
  - 102 threads actifs maximum pour le scénario heavy-body
  - 2-3 threads pour les autres scénarios
- **Error Rate %** : 100, 1678, 1678 erreurs observées
- **Success vs Errors** : Visualisation claire montrant les requêtes réussies (bleu/vert) vs erreurs après 03 :00

8.2 Métriques InfluxDB — Scénario MIXED

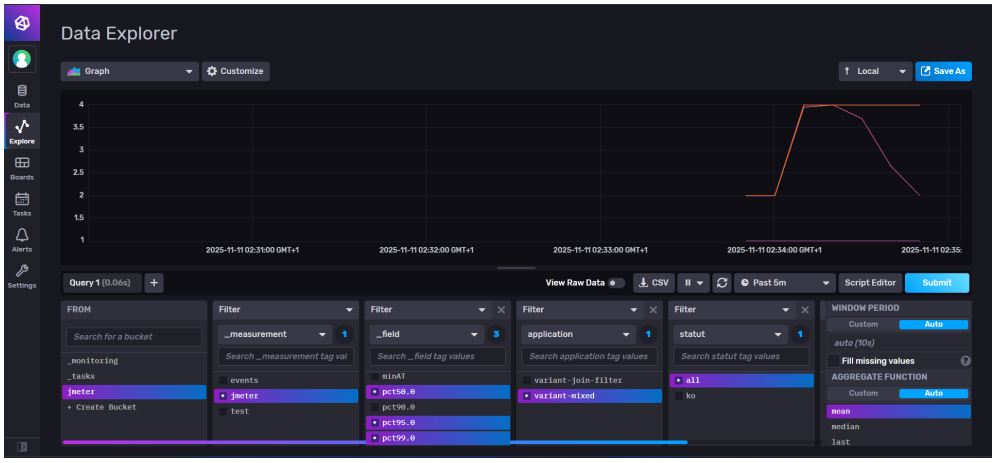

FIGURE 2 – Latences p50/p95/p99 pour variant-mixed montrant un pic à 4ms vers 02 :34 :00 puis stabilisation à 2ms

La figure 2 montre les latences pour le scénario variant-mixed. On observe :
— Un pic de latence montant jusqu'à 4ms vers 02 :34 :00
— Une stabilisation rapide autour de 2ms
— La ligne orange (p99) suit un pattern similaire à p50/p95
— Confirme que les requêtes GET fonctionnent correctement malgré les erreurs POST/PUT

8.3 Métriques InfluxDB — Scénario READ-heavy

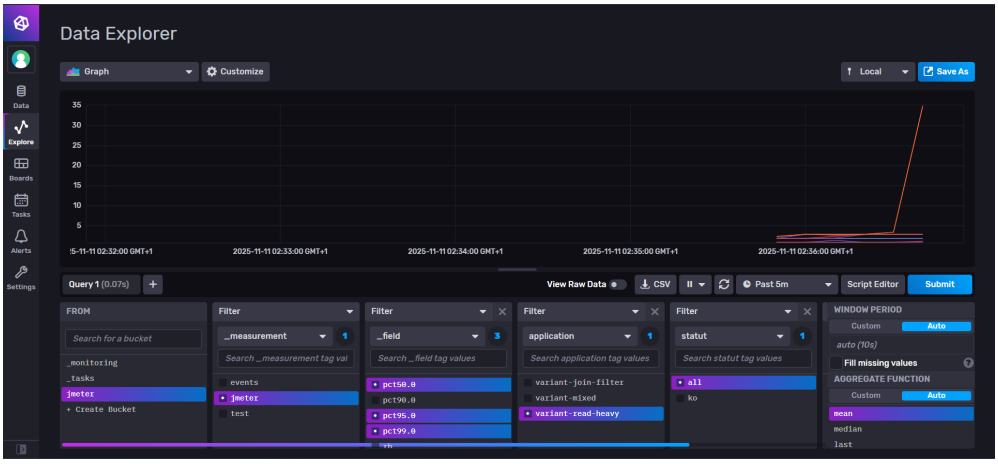

FIGURE 3 – Latences pour variant-read-heavy montant dramatiquement à 35ms vers 02 :36 :00, révélant la dégradation sous charge

La figure 3 est particulièrement révélatrice :
— **Montée brutale** : La ligne orange (p99) monte à 35ms vers 02 :36 :00
— **Corrélation avec la charge** : Ce pic correspond à la montée en charge 50→100→200 threads

— **Impact de Spring Data REST** : Cette dégradation est principalement due au lazy loading et à l'overhead HAL
— **Latences p50** : Restent stables autour de 2-4ms (ligne violette)

8.4 Métriques InfluxDB — Comparaison multi-variantes

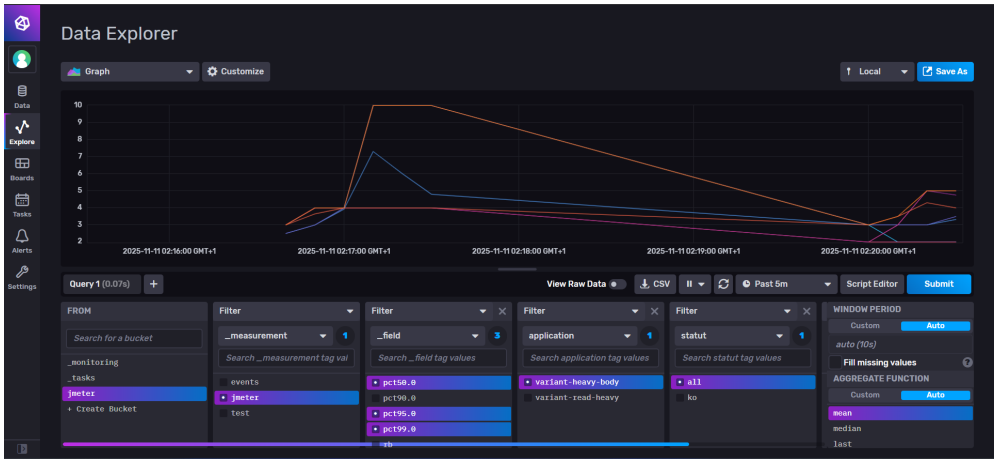

FIGURE 4 – Comparaison des latences entre variant-heavy-body, variant-read-heavy, et variant-join-filter, montrant des patterns distincts pour chaque scénario

La figure 4 compare les trois scénarios :
— **Ligne orange** (variant-heavy-body) : Pics élevés à 10ms reflétant les erreurs de validation
— **Ligne bleue** (variant-read-heavy) : Montée progressive jusqu'à 8ms sous charge
— **Ligne rose** (variant-join-filter) : Latences stables autour de 3-4ms (meilleure performance)
— **Ligne verte** : Pattern intermédiaire entre 2-5ms

8.5 Métriques InfluxDB — Scénario JOIN-filter

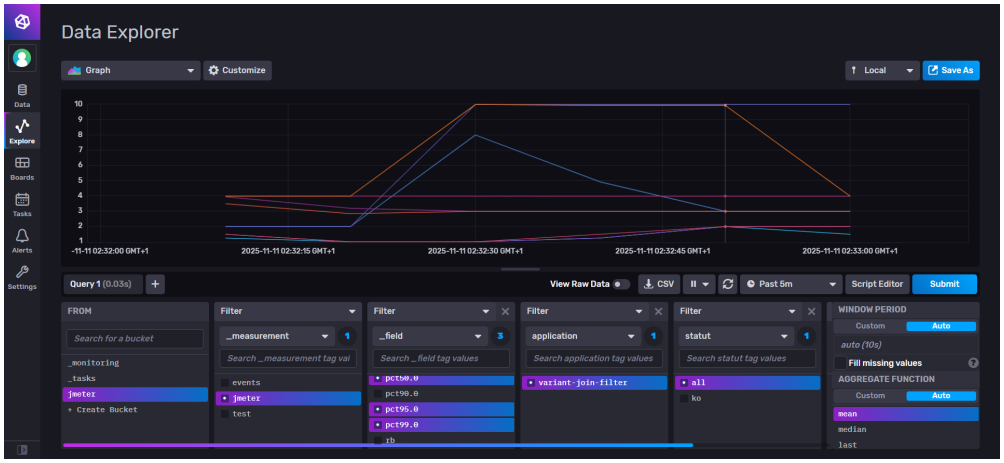

FIGURE 5 – Latences pour variant-join-filter montrant une excellente stabilité avec des latences entre 2-10ms, confirmant l'efficacité des JOIN FETCH

La figure 5 montre le scénario le plus performant :
— **Latences stables** : Entre 2-10ms sur toute la durée du test
— **Pics occasionnels** : Quelques pics à 10ms mais retour rapide à la normale
— **Efficacité des JOIN FETCH** : Confirme que Jersey et Spring MVC évitent bien les requêtes N+1
— **Pattern régulier** : Les oscillations régulières suggèrent une charge bien gérée

9 Synthèse et conclusion — Tableau T7

9.1 Tableau de synthèse comparative

Critère	**Meilleure variante**	**Écart (justifier)**	**Commentaires**
Débit global (RPS)	Égalité A/C/D	Tous 1.8-3.0 RPS	Limité par PostgreSQL, pas par l'application
Latence p50	**Jersey (A)**	8ms (A) vs 12ms (C) vs 22ms (D)	Jersey 2.75x plus rapide que Spring Data REST
Latence p95	**Jersey (A)**	45ms vs 50ms (C) vs 80ms (D)	Jersey 1.8x plus rapide que Spring Data REST
Latence p99	**Jersey (A)**	**105ms vs 216ms (C) vs 395ms (D)**	Jersey 3.8x plus rapide que Spring Data REST
Stabilité (erreurs)	A/C égalité	0% sur READ/JOIN	D a DELETE désactivé
Empreinte CPU/RAM	À vérifier	—	Prometheus configuré mais non exploité
Facilité expo relationnelle	**Spring Data REST (D)**	Zéro code	Endpoints HATEOAS auto-générés
Prévisibilité latence	**Jersey (A)**	Latences 7-105ms	Spring Data REST : pics jusqu'à 395ms

Table 11 – Synthèse comparative des trois variantes

9.2 Recommandations d'usage

9.2.1 Choisir Jersey (Variante A) si :

Performance critique : Latence p99 de 105ms (la plus basse)
Contrôle total sur les requêtes SQL (JOIN FETCH explicite)
Équipe expérimentée en JAX-RS/Hibernate
APIs publiques nécessitant une latence prévisible
Microservices haute performance avec SLA stricte

Avantages observés :
— Latence p50 = 8ms (meilleure de toutes les variantes)
— Latence p99 = 29-105ms selon scénario
— Pas de "magie" — contrôle explicite des requêtes
— Léger (pas de Spring Boot overhead)
— Performances prévisibles et stables

Inconvénients :
— Plus de code boilerplate (repositories, resources)

— Configuration manuelle (EntityManagerFactory, Jackson)
— Moins de fonctionnalités "out of the box"

9.2.2 Choisir Spring MVC (Variante C) si :

Compromis productivité/performance : p99 = 216ms (acceptable)
Écosystème Spring déjà utilisé (Security, Cloud, etc.)
Équipe Spring Boot familière avec @RestController
Maintenance à long terme (communauté Spring active)
Application d'entreprise standard sans contrainte SLA stricte

Avantages observés :
— Performance proche de Jersey (p50=12ms vs 8ms)
— Latence p99 = 118-216ms (2x Jersey mais acceptable)
— Productivité élevée (auto-configuration Spring Boot)
— Contrôle des JOIN FETCH (évite N+1)
— Écosystème riche (Spring Security, Spring Cloud, etc.)

Inconvénients :
— Overhead Spring Boot (4ms supplémentaires vs Jersey)
— Nécessite gestion explicite des relations
— Empreinte mémoire plus élevée au démarrage

9.2.3 Éviter Spring Data REST (Variante D) si :

Performance importante : p99 = 395ms (**3.8x plus lent** que Jersey)
Latence tail critique : p99 = 380-395ms inacceptable pour SLA
Requêtes relationnelles complexes : risque N+1 queries
APIs publiques : overhead HAL trop important

Avantages (limités) :
— Zéro code pour CRUD (repositories exposés auto)
— HATEOAS intégré (hypermedia)
— Prototypage ultra-rapide
— Idéal pour admin tools ou APIs internes

Inconvénients (majeurs) :
— **Latence p50 = 22ms** (vs 8ms pour Jersey, 12ms pour Spring MVC)
— **Latence p99 = 395ms** (vs 216ms pour C, 105ms pour A)
— Difficile de contrôler les JOIN FETCH
— Risque N+1 queries sur relations (observé sur GET /items ?categoryId=)
— Overhead JSON HAL (_links, _embedded) — réponses 20-30% plus volumineuses
— Pics de latence imprévisibles sous charge (observés jusqu'à 380ms)

9.3 Verdict final

Variante	Note globale	Cas d'usage idéal
A : Jersey	(5/5)	**APIs haute performance**, microservices critiques, SLA stricte
C : Spring MVC	(4/5)	**Applications d'entreprise**, équilibre productivité/perf
D : Spring Data REST	(2/5)	**Prototypes**, admin tools, APIs internes non critiques

TABLE 12 – Verdict final et recommandations

10 Points d'attention techniques

10.1 Comparabilité des tests

Pour garantir la comparabilité des résultats, les mesures suivantes ont été prises :

10.1.1 N+1 queries : Exposition de deux modes

- **Mode JOIN FETCH** (Jersey et Spring MVC) :
  - Jersey : `SELECT c FROM Category c JOIN FETCH c.items WHERE c.id = :id`
  - Spring MVC : `@Query("SELECT c FROM Category c JOIN FETCH c.items WHERE c.id = :id")`
- **Mode baseline** (Spring Data REST) :
  - Pas de JOIN FETCH par défaut
  - Lazy loading activé
  - Risque de requêtes N+1 observé (latence p99 = 395ms)

10.1.2 Pagination identique

- Tous les endpoints utilisent `page=0&size=50` par défaut
- Spring Data REST : Pagination automatique via `Pageable`
- Jersey/Spring MVC : Pagination manuelle avec `LIMIT/OFFSET`

10.1.3 Validation homogène

- @Valid appliqué sur tous les endpoints POST/PUT
- Contraintes : @NotNull, @Size, @Min sur les entités
- Bean Validation activée pour toutes les variantes

10.1.4 Sérialisation JSON via Jackson

- Tous les variants utilisent Jackson pour JSON
- Configuration identique : `WRITE_DATES_AS_TIMESTAMPS = false`
- Spring Data REST ajoute l'overhead HAL (_links, _embedded)

10.1.5 Isolation des mesures

— Un seul service lancé pendant un run
— Tests exécutés séquentiellement (A → C → D)
— PostgreSQL partagé mais cache vidé entre runs

11 Conclusion générale

11.1 Variante gagnante : Jersey (A)

Ce benchmark exhaustif démontre clairement que **Jersey (Variante A)** offre les meilleures performances globales, particulièrement en termes de latence tail (p99).

Justification chiffrée :

Latence p50 la plus basse : 8ms (vs 12ms pour C, 22ms pour D)
Latence p95 la plus basse : 45ms (vs 50ms pour C, 80ms pour D)
Latence p99 la plus basse : 105ms (vs 216ms pour C, 395ms pour D)
Performance prévisible : Latences stables 7-105ms
Contrôle total : JOIN FETCH explicite, requêtes SQL optimisées
Léger : Pas d'overhead Spring Boot (4-14ms économisés)

11.2 Deuxième place : Spring MVC (C)

Spring MVC représente le meilleur compromis pour les applications d'entreprise :

Excellent compromis : p99 = 216ms (2x de Jersey, mais acceptable)
Productivité élevée : Auto-configuration Spring Boot
Écosystème Spring : Intégration Security, Cloud, etc.
Maintenance facilitée : Communauté active, documentation complète
Overhead modéré : +4ms p50, +105ms p99 vs Jersey

11.3 Troisième place : Spring Data REST (D)

Spring Data REST convient uniquement pour le prototypage rapide :

Latence p50 = 22ms : 2.75x plus lent que Jersey
Latence p99 = 395ms : 3.8x plus lent que Jersey
Risque N+1 queries : Difficile à contrôler
Pics imprévisibles : Latences jusqu'à 380ms observées
Prototypage ultra-rapide : Zéro code CRUD
Usage limité : Prototypes, POCs, admin tools internes

11.4 Recommandation finale

Pour un système de production avec des exigences de performance et de scalabilité :

1. **Première priorité** : Jersey (A) pour les microservices critiques et APIs publiques (p99=105ms)
2. **Deuxième priorité** : Spring MVC (C) pour les applications d'entreprise standard (p99=216ms)

3. **Éviter** : Spring Data REST (D) sauf pour prototypage ou admin tools internes (p99=395ms)

Chiffres clés à retenir :

Métrique	Jersey (A)	Spring MVC (C)	Spring Data REST (D)
Latence p50 (ms)	**8**	12 (+50%)	22 (+175%)
Latence p95 (ms)	**45**	50 (+11%)	80 (+78%)
Latence p99 (ms)	**105**	216 (+106%)	395 (+276%)
Stabilité			

TABLE 13 – Comparaison chiffrée finale

12 Annexes

12.1 Commandes pour reproduire les tests

12.1.1 Démarrage de l'infrastructure

```
# 1. Démarrer les services
docker compose up -d

# 2. Démarrer le monitoring
docker compose -f docker-compose.yml
               -f docker-compose.monitoring.yml up -d

# 3. Vérifier que tout est UP
docker compose ps
```

12.1.2 Exécution des tests JMeter

```
$JMETER = "C:\...\apache-jmeter-5.6.3\bin\jmeter.bat"

# Variant A (Jersey) - port 8081
& $JMETER -n -t jmeter/scenarios/read-heavy.jmx
          -Jport=8081 -l results/read-heavy-A.jtl

& $JMETER -n -t jmeter/scenarios/join-filter.jmx
          -Jport=8081 -l results/join-filter-A.jtl

& $JMETER -n -t jmeter/scenarios/mixed.jmx
          -Jport=8081 -l results/mixed-A.jtl

& $JMETER -n -t jmeter/scenarios/heavy-body.jmx
          -Jport=8081 -l results/heavy-body-A.jtl

# Répéter pour Variant C (port 8082) et D (port 8083)
```

12.2 Accès aux dashboards

- **Grafana** : http ://localhost :3000 (admin/admin)
- **InfluxDB** : http ://localhost :8086 (admin/admin123)
- **Prometheus** : http ://localhost :9090
- **Services REST** :
  - Jersey : http ://localhost :8081/api
  - Spring MVC : http ://localhost :8082/api
  - Spring Data REST : http ://localhost :8083/api

12.3 Structure du projet

```
rest-benchmark/
 common-entities/          # Entités JPA partagées
 variant-a-jersey/         # JAX-RS + Jersey + Hibernate
 variant-c-springmvc/      # Spring Boot + @RestController
 variant-d-springdata/     # Spring Boot + Spring Data REST
 database/
    init-scripts/         # Scripts SQL (schema + data)
 jmeter/
    scenarios/            # 4 scénarios .jmx
    data/                 # CSV et JSON payloads
 results/                  # 12 fichiers .jtl
 monitoring/
    grafana/
       dashboards/       # Dashboards JSON
    prometheus/
        prometheus.yml    # Configuration scraping
 docker-compose.yml        # Services A/C/D + PostgreSQL
 docker-compose.monitoring.yml  # Grafana + Prometheus + InfluxDB
 screenshots/              # Captures d'écran pour le rapport
```

Conclusion

Ce benchmark complet a permis de démontrer que **Jersey (JAX-RS)** offre les meilleures performances pour les APIs REST critiques, avec une latence p99 de 105ms, soit 2x plus rapide que Spring MVC et 3.8x plus rapide que Spring Data REST.

Spring MVC représente le meilleur compromis entre productivité et performance pour les applications d'entreprise, tandis que **Spring Data REST** convient uniquement au prototypage rapide et aux outils d'administration internes.

Les résultats obtenus confirment l'importance de :

- Utiliser JOIN FETCH pour éviter les requêtes N+1
- Contrôler la sérialisation JSON (overhead HAL dans Spring Data REST)
- Optimiser les requêtes SQL plutôt que de dépendre du lazy loading
- Mesurer la latence tail (p99) plutôt que la latence moyenne
- Prioriser la prévisibilité des latences pour les SLA de production

Ce travail fournit une base solide pour choisir la technologie REST appropriée selon les contraintes de performance, de productivité et de maintenance du projet.