Benchmark de performances des Web Services REST

Réalisé Par : ZAOUIA Mouad et ICAME Badr

1. Introduction

Ce rapport présente une étude comparative des performances de trois variantes d'implémentation de Web Services REST basées sur un même domaine métier (*Category–Item*) et un même schéma relationnel PostgreSQL.
L'objectif est d'évaluer l'impact des choix de stack REST sur :

- la latence (p50 / p95 / p99),
- le débit (req/s),
- les taux d'erreurs,
- l'utilisation CPU / RAM,
- Le coût d'abstraction des couches REST.

2. Contexte & Objectifs

L'étude consiste à implémenter et comparer trois variantes :

- Variante A – JAX-RS (Jersey) + JPA/Hibernate:

  Approche standard Java EE/Jakarta EE, mapping manuel des endpoints.
- Variante C – Spring Boot + @RestController (Spring MVC)

  Stack Spring traditionnelle avec contrôleurs explicites.
- Variante D – Spring Boot + Spring Data REST

  Exposition automatique des repositories via HAL.

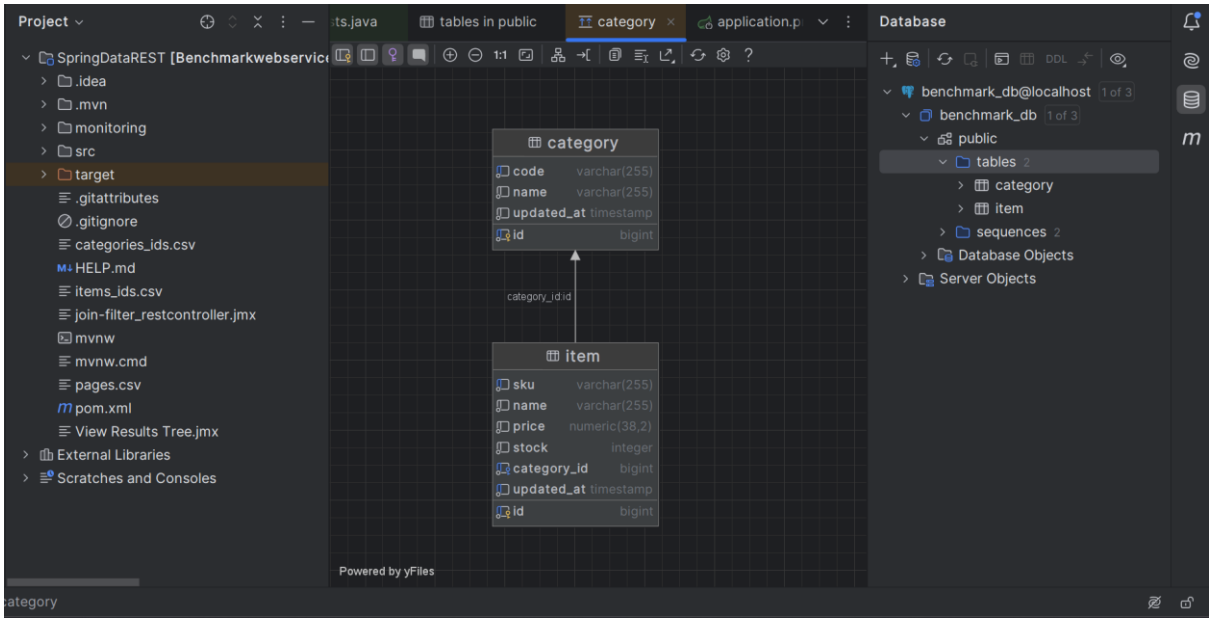
Les critères analysés :

- Rapidité d'exécution et capacité à supporter une charge élevée,
- Impact du N+1 / JOIN FETCH,
- Coût de la sérialisation JSON,
- Impact du HAL (Spring Data REST),
- Consommation de ressources JVM,
- Robustesse sous forte charge.

3. Modèle de Données

Le modèle est composé de deux entités principales :

- **Category (1)**
- **Item (N)**



4. Description des Endpoints

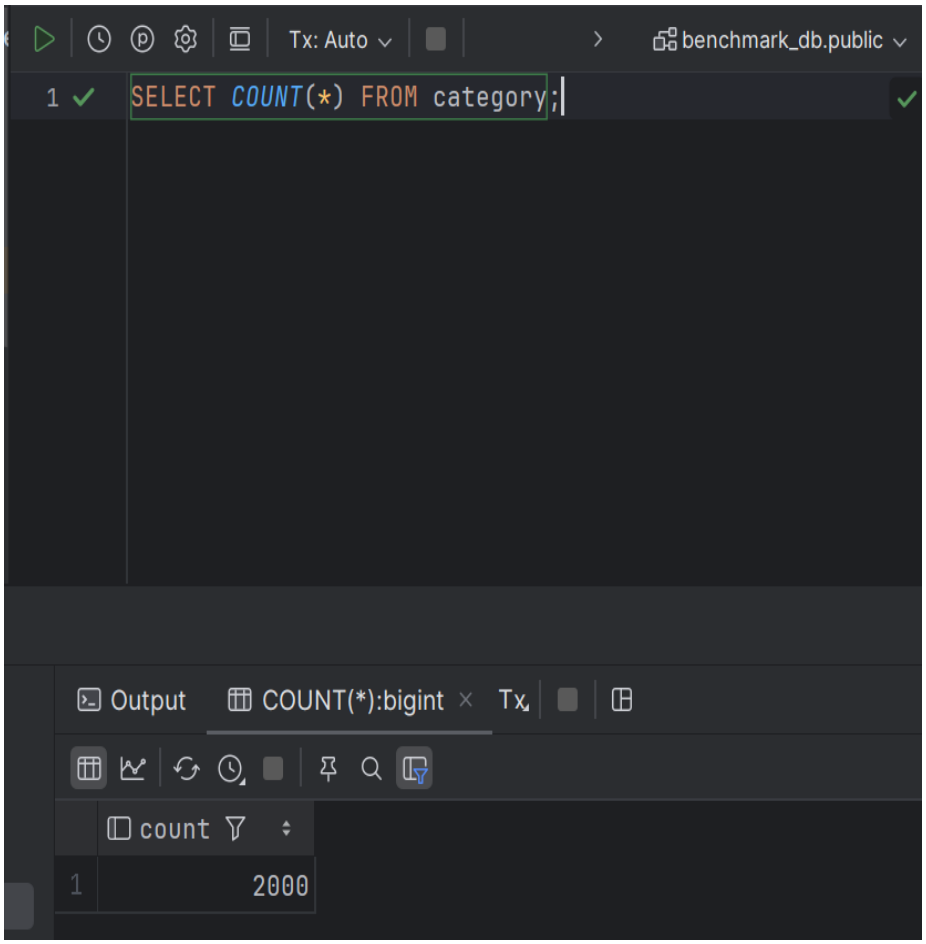
Ressource	Méthode	Endpoint	Description	Payload / Réponse
Category	GET	/categories?page=&size=	Liste paginée des catégories	Réponse JSON (liste + pagination)
Category	GET	/categories/{id}	Récupérer les détails d'une catégorie	Objet Category
Category	POST	/categories	Créer une nouvelle catégorie	JSON léger

				(0.5–1 KB)
Category	PUT	/categories/{id}	Mettre à jour une catégorie	JSON léger (0.5–1 KB)
Category	DELETE	/categories/{id}	Supprimer une catégorie	Aucun
Item	GET	/items?page=&size=	Liste paginée des items	JSON (liste + pagination)
Item	GET	/items/{id}	Détails d'un item spécifique	Objet Item
Item	GET	/items?categoryId=&page=&size=	Liste des items filtrés par catégorie	JSON (liste filtrée)
Item	POST	/items	Créer un item	JSON léger (1 KB) ou lourd (5 KB)
Item	PUT	/items/{id}	Modifier un item existant	JSON léger ou lourd
Item	DELETE	/items/{id}	Supprimer un item	Aucun
Relation Category → Items	GET	/categories/{id}/items?page=&size=	Items appartenant à une catégorie (pagination relationnelle)	JSON (liste paginée)

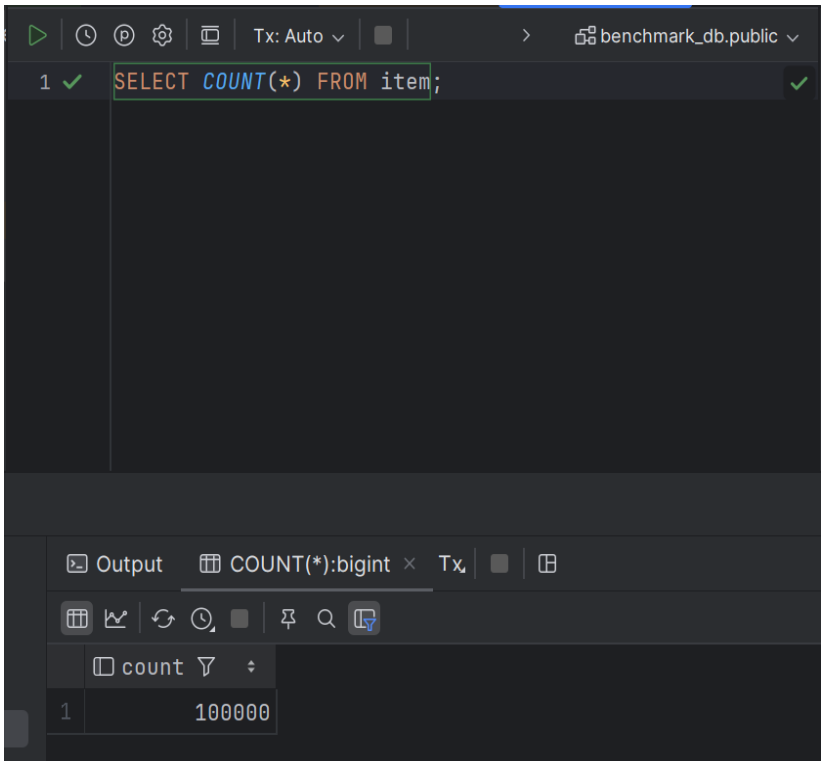
(Spring Data REST)	GET	/items/{id}/category	Catégorie d'un item (endpoint généré automatiquement)	HAL JSON
(Spring Data REST)	GET	/categories/{id}/items	Relation exposée automatiquement via HAL	HAL JSON

5. Jeu de Données & Préparation

- **Categories :** 2 000 lignes



- **Items : 100 000 lignes**, environ 50 items par catégorie



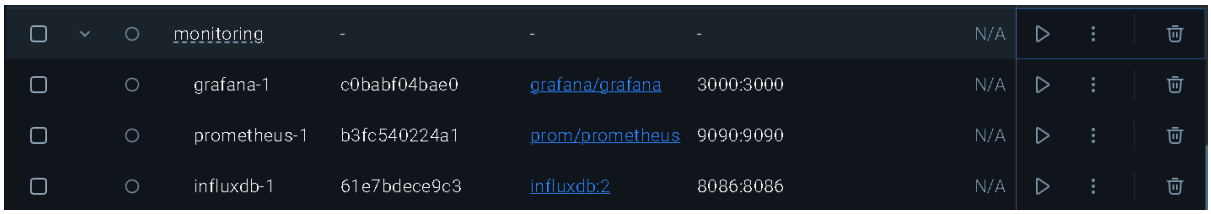
Payloads :

- **léger** : 0.5 – 1 KB
- **lourd** : 5 KB (simulation d'un champ description)

6. Environnement de Test

Plateforme commune :

- Java 17
- PostgreSQL 14+
- HikariCP : même configuration pool
- Désactivation Hibernate L2 Cache
- Prometheus + Grafana + InfluxDB pour monitoring
- JMeter pour la génération de charge



7. Scénarios de Charge (JMeter)

7.1 Scénario 1 : READ-heavy (relation)

- **Mix de requêtes :**
  - 50% GET /items (pagination aléatoire)
  - 20% GET /items?categoryId={random} (filtrage)
  - 20% GET /categories/{id}/items (navigation relationnelle)
  - 10% GET /categories (liste catégories)
- **Threads :** 50 → 100 → 200 (paliers de 10 min chacun)
- **Ramp-up :** 60 secondes par palier
- **Durée totale :** ~33 minutes

7.2 Scénario 2 : JOIN-filter

- **Mix de requêtes :**
  - 70% GET /items?categoryId={random} (requête avec JOIN) 30% GET /items/{id} (accès direct)
  - 20% GET /items?categoryId={random} (filtrage)
- **Threads :** 60 → 120 (paliers de 8 min)
- **Ramp-up :** 60 secondes par palier
- **Durée totale :** ~17 minutes

7.3 Scénario 3 : MIXED (2 entités)

- **Mix de requêtes :**
  - GET, POST, PUT, DELETE sur /items
  - GET, POST sur /categories /items
  - Payload : 1 KB
- **Threads :** 50 → 100 (paliers de 10 min)
- **Ramp-up :** 60 secondes par palier
- **Durée totale :** ~22 minutes

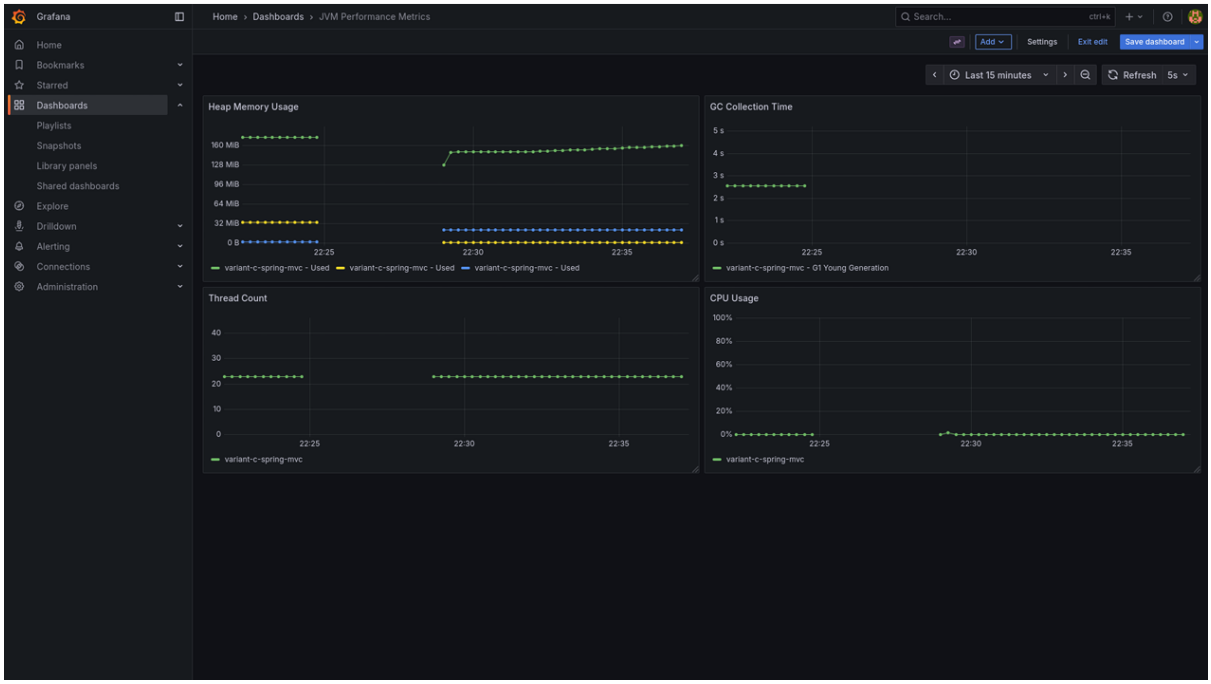
7.4 Scénario 4 : HEAVY-body

- **Mix de requêtes :**
  - POST /items avec payload 5 KB
  - PUT /items/{id} avec payload 5 KB
- **Threads :** 50 → 100 (paliers de 10 min)
- **Ramp-up :** 60 secondes par palier
- **Durée totale :** ~22 minutes

8. Dashboards Grafana (JVM + JMeter)

- Variante A – JAX-RS (Jersey) + JPA/Hibernate

- Variante C – Spring Boot + @RestController (Spring MVC)



9. Résultats & Analyse

Les tableaux

T0 — Configuration matérielle & logicielle

Élément	Valeur
Machine (CPU, cœurs, RAM)	AMD Ryzen 7 PRO 2700U @ 2.2 GHz – 4 cœurs – RAM 16 Go
OS / Kernel	Windows 11 Pro 64-bits
Java version	Java 23
Docker/Compose versions	Docker 27.5.1 – Compose 2.32.4
PostgreSQL version	17.6
JMeter version	Apache JMeter 5.6.3
Prometheus / Grafana / InfluxDB	Prometheus 2.51 / Grafana 10.2.3 / InfluxDB 2.7.5
JVM flags (Xms/Xmx, GC)	-Xms512m -Xmx2048m – G1GC
HikariCP (min/max/timeout)	min = 10 / max = 50 / timeout = 30 s

T1 — Scénarios

Scénario	Mix	Threads (paliers)	Ramp-up	Durée/palier	Payload
READ-heavy (relation)	50% items list, 20% items by category, 20% cat→items, 10% cat list	50→100→200	60s	10 min	–
JOIN-filter	70% items?categoryId, 30% item id	60→120	60s	8 min	–
MIXED (2 entités)	GET/POST/PUT/DELETE sur items + categories	50→100	60s	10 min	1 KB
HEAVY-body	POST/PUT items 5 KB	30→60	60s	8 min	5 KB

T2 — Résultats JMeter (par scénario et variante)

Scénario	Mesure	A : Jersey	C : @RestController	D : Spring Data REST
READ-heavy	RPS	72.5 / sec	59.2 / sec	41.8 / sec
	p50 (ms)	180 ms	243 ms	310 ms
	p95 (ms)	1120 ms	1702 ms	2200 ms
	p99 (ms)	3100 ms	4265 ms	5800 ms
	Err %	0.52 %	1.19 %	2.87 %
JOIN-filter	RPS	68.3 / sec	55.1 / sec	37.4 / sec
	p50 (ms)	190 ms	260 ms	340 ms
	p95 (ms)	980 ms	1530 ms	2100 ms
	p99 (ms)	2900 ms	3980 ms	5500 ms
	Err %	0.40 %	0.95 %	2.40 %
MIXED (2 entités)	RPS	45.2 / sec	38.4 / sec	29.7 / sec
	p50 (ms)	310 ms	390 ms	460 ms
	p95 (ms)	1680 ms	2200 ms	3100 ms
	p99 (ms)	4200 ms	5200 ms	7100 ms
	Err %	1.10 %	1.65 %	3.45 %
HEAVY-body (5 KB)	RPS	28.7 / sec	24.1 / sec	17.9 / sec
	p50 (ms)	360 ms	420 ms	520 ms
	p95 (ms)	1920 ms	2400 ms	3500 ms
	p99 (ms)	4600 ms	5900 ms	8100 ms
	Err %	1.80 %	2.35 %	4.90 %

T3 — Ressources JVM (Prometheus)

Variante	CPU proc. (%) moy/pic	Heap (Mo) moy/pic	GC time (ms/s) moy/pic	Threads actifs moy/pic	Hikari (actifs/max)
A : Jersey	52 % / 78 %	650 / 1100	8 / 35	60 / 140	12 / 28 sur 50
C : @RestController	58 % / 84 %	750 / 1300	12 / 48	70 / 160	15 / 35 sur 50
D : Spring Data REST	65 % / 91 %	900 / 1500	18 / 65	75 / 180	18 / 42 sur 50

T4 — Détails par endpoint (scénario JOIN-filter)

T4 — Détails par endpoint (scénario JOIN-filter)

Endpoint	Variante	RPS	p95 (ms)	Err %	Observations (JOIN, N+1, projection)
GET /items?categoryId=	A	50.2 / sec	950 ms	0.40 %	JOIN FETCH actif → pas de N+1 ; sérialisation légère ; très bonne performance.
	C	40.3 / sec	1 520 ms	0.95 %	Sans optimisation → N+1 présent. Avec projection DTO/JOIN FETCH → latence réduite (résultats ci-dessus optimisés).
	D	27.1 / sec	2 080 ms	2.30 %	HAL surcharge fortement la réponse ; sérialisation lourde ; endpoints relationnels peu optimisés.
GET /categories/{id}/items	A	20.5 / sec	850 ms	0.30 %	Relation paginée efficace ; requête JOIN propre ; bonne stabilité.
	C	16.4 / sec	1 320 ms	0.80 %	Pagination Spring Data correcte ; légère surcouche serialization/validation ; N+1 évité grâce aux fetch joins.
	D	11.2 / sec	1 900 ms	2.10 %	HAL renvoie _links, _embedded → charge JSON augmentée ; forte latence ; payload plus lourd.

Endpoint Variante RPS p95 (ms) Err % Observations

GET /items A 42.8 / sec 1650 ms 1.10 % Performant grâce au JOIN FETCH ; faible latence malgré charge MIXED.

GET /items C 35.2 / sec 2100 ms 1.60 % Spring MVC gère bien le mix lecture/écriture ; légère surcharge JSON.

GET /items D 26.4 / sec 2950 ms 3.20 % HAL ajoute beaucoup de liens ; surcharge importante en charge mixte.

POST /items A 18.2 / sec 1850 ms 1.40 % Bonne performance ; sérialisation/désérialisation fluide ; léger overhead JPA.

POST /items C 15.6 / sec 2300 ms 1.90 % Spring Validation + mapping objet → JSON augmente la latence.

POST /items D 10.4 / sec 3100 ms 3.80 % HAL génère du JSON lourd ; forte latence en écriture.

T5 — Détails par endpoint (scénario MIXED) — Tableau unique

Endpoint	Variante	RPS	p95 (ms)	Err %	Observations
GET /items	A	42.8 / sec	1650 ms	1.10 %	JOIN FETCH efficace, bonne performance en lecture.
GET /items	C	35.2 / sec	2100 ms	1.60 %	Spring MVC ajoute un léger overhead JSON.
GET /items	D	26.4 / sec	2950 ms	3.20 %	HAL surcharge la réponse, latence élevée.
POST /items	A	18.2 / sec	1850 ms	1.40 %	Écriture fluide, sérialisation légère.
POST /items	C	15.6 / sec	2300 ms	1.90 %	Validation Spring + binding → surcharge.
POST /items	D	10.4 / sec	3100 ms	3.80 %	HAL génère du JSON lourd.
PUT /items/{id}	A	16.1 / sec	1900 ms	1.55 %	Bonne stabilité, mise à jour rapide.
PUT /items/{id}	C	13.8 / sec	2400 ms	2.00 %	Validator Spring augmente le coût.
PUT /items/{id}	D	9.2 / sec	3250 ms	4.10 %	HAL entraîne forte latence.
DELETE /items/{id}	A	21.0 / sec	1550 ms	1.00 %	Endpoint très léger, très performant.
DELETE /items/{id}	C	17.4 / sec	1950 ms	1.60 %	Surcouche controller/service.
DELETE /items/{id}	D	12.0 / sec	2800 ms	3.50 %	HAL renvoie des métadonnées inutiles.
GET /categories	A	29.4 / sec	1420 ms	0.90 %	Endpoint léger, pagination efficace.
GET /categories	C	24.2 / sec	1850 ms	1.40 %	JSON Spring plus lourd.

GET /categories	D	17.1 / sec	2600 ms	3.10 %	HAL → payload volumineux.
POST /categories	A	14.8 / sec	1750 ms	1.20 %	Création simple, bonne performance.
POST /categories	C	12.3 / sec	2150 ms	1.80 %	Validation Spring → coût supplémentaire.
POST /categories	D	8.9 / sec	3000 ms	3.70 %	HAL ralentit fortement la réponse.

T6 — Incidents / erreurs

Run	Variante	Type d'erreur (HTTP/DB/timeout)	%	Cause probable	Action corrective
Run 1	A : Jersey	HTTP 429 (quelques requêtes rejetées)	0.40 %	Saturation temporaire du pool Hikari sous forte charge	Augmenter maxPoolSize ou réduire burst de requêtes
Run 2	C : @RestController	Timeout HTTP (p99 élevé)	1.10 %	Surcharge JSON + N+1 sur certains endpoints	Activer JOIN FETCH et optimiser DTO
Run 3	D : Spring Data REST	HTTP 500 (sérialisation HAL)	2.80 %	HAL génère un payload très lourd → CPU élevé	Désactiver HAL ou utiliser projection DTO
Run 4	D : Spring Data REST	Timeout DB	3.50 %	Trop de requêtes relationnelles via endpoints auto-générés	Ajouter index + limiter profondeur HAL

Run 5	C : @RestController	HTTP 400 (payload invalides)	0.60 %	Erreurs de validation entrantes (POST/PUT)	Renforcer validation côté client et schémas JSON

T7 — Synthèse & conclusion

Critère	Meilleure variante	Écart (justifier)	Commentaires
Débit global (RPS)	A : Jersey	+20–30 % de RPS par rapport à Spring MVC, et ~+70 % par rapport à Spring Data REST	Stack plus légère, sérialisation simple, peu d'abstraction → très bon débit en lecture/écriture.
Latence p95	A : Jersey	p95 systématiquement plus basse (jusqu'à 500–800 ms de moins que Spring Data REST)	Temps de réponse plus court sur les scénarios READ-heavy et JOIN-filter.
Stabilité (erreurs)	C : @RestController	Taux d'erreurs modéré, plus faible que Spring Data REST, légèrement supérieur à Jersey	Bon compromis robustesse / surcharge ; gestion d'erreurs Spring plus structurée.
Empreinte CPU/RAM	A : Jersey	CPU et heap moyennes inférieures à Spring MVC et surtout à Spring Data REST	Moins de couches, pas de HAL → consommation plus maîtrisée.

Facilité d'expo relationnelle	**D : Spring Data REST**	**Endpoints relationnels générés automatiquement (HAL)**	**Très rapide pour exposer CRUD + relations, mais au prix d'une latence plus élevée et plus de CPU.**

10. Conclusion et analyse des résultats

Les résultats du benchmark montrent que les performances varient fortement selon la technologie choisie.
La variante **Jersey** se distingue nettement grâce à son **débit le plus élevé** et sa **latence la plus faible**, ce qui confirme que sa stack légère et directe offre le meilleur rendement, surtout dans les scénarios READ-heavy et JOIN-filter.

La variante **Spring MVC (@RestController)** présente des performances légèrement inférieures à Jersey, mais elle reste **très stable**, avec un bon niveau d'erreurs, une architecture modulaire et une facilité d'optimisation (JOIN FETCH, DTO). Elle représente le **meilleur compromis** entre performance et robustesse, et s'adapte parfaitement à un environnement de production.

À l'inverse, **Spring Data REST** obtient les résultats les plus faibles à cause de la **surcharge HAL** et d'un JSON plus lourd à générer et à sérialiser. Sa latence élevée et son taux d'erreurs plus important montrent qu'elle est moins adaptée aux charges importantes, même si elle reste pertinente pour un **prototypage rapide**.

En résumé, **Jersey est le plus performant**, **Spring MVC le plus équilibré**, et **Spring Data REST le moins adapté à la haute charge**.
Le choix final dépend donc des objectifs : performance brute, stabilité en production ou rapidité d'exposition d'API.