T0—Configurationmatérielle&logicielle

Élément	Valeur
Machine(CPU,cœurs,RAM)	10th Gen Intel® Core™ i7-10700H (14 cœurs / 20 threads, 2.3 GHz), 16 GB RAM
OS/Kernel	Windows 11 Pro 64-bits
Javaversion	Java SE 17.0.6 (LTS), HotSpot 64-bit VM
Docker/Composeversions	Docker 27.5.1 / Docker Compose 2.21.0 (stable et compatible)
PostgreSQLversion	14.19
JMeterversion	5.6.3
Prometheus/Grafana/ InfluxDB	**v2.54.1** (2025-LTS)/ v11.1.3/ 2.7.10
JVMflags(Xms/Xmx,GC)	0.20.0
HikariCP(min/max/timeout)	5.1.0

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/DELETEsur items+categories	50→100	60s	10 min	1 KB
HEAVY-body	POST/PUTitems5KB	30→60	60s	8 min	5 KB

T2—RésultatsJMeter(parscénarioetvariante)

Scénario	Mesure	A:Jersey	C: @RestController	D:SpringData REST
READ-heavy	**RPS**	320 req/s	469.27 req/s	469.27 req/s
READ-heavy	**p50(ms)**	~21 ms	58.00	58.00
READ-heavy	**p95(ms)**	~69 ms	308.00	**308.00**

READ-heavy	**p99(ms)**	~105 ms	830.40	830.40
READ-heavy	**Err%**	0.01 %	0.000000%	**0.00 %**
JOIN-filter	**RPS**	≈ 2,0 / sec	2708 req/s	2708.30
JOIN-filter	**p50(ms)**	1	15	15.00
JOIN-filter	**p95(ms)**	45	56	56.00
JOIN-filter	**p99(ms)**	69	92	92.00
JOIN-filter	**Err%**	0,00%	9.99 %	9.991492%
MIXED(2entités)	**RPS**		110.28 req/s	110.28
MIXED(2entités)	**p50(ms)**		1.0	1.00
MIXED(2entités)	**p95(ms)**		12.0	12.00
MIXED(2entités)	**p99(ms)**		14.0	**14.00**
MIXED(2entités)	**Err%**		55.81 %	55.811985 %
HEAVY-body	**RPS**		101.84 req/s	**101.84 req**
HEAVY-body	**p50(ms)**		2.0	**2.00**
HEAVY-body	**p95(ms)**		11.0	11
HEAVY-body	**p99(ms)**		12.0	12
HEAVY-body	**Err%**		75.53 %	75.530326

T3—RessourcesJVM(Prometheus)

Variante	**CPU proc. (%)moy/pic**	**Heap(Mo) moy/pic**	**GCtime (ms/s) moy/pic**	**Threads actifs moy/pic**	**Hikari (actifs/max)**
A : Jersey	34 % / 82 %	143 / 275	----	----	----
C : @RestController	17 % / 64–76 %	300 / 900	----	----	----
D:SpringData REST	34 % / 62 %	143 / 155	----	----	----

T4—Détailsparendpoint(scénarioJOIN-filter)

Endpoint	Variante	RPS	p95 (ms)	Err %	Observations(JOIN,N+1, projection)
sGET/items?categoryId=	A	16 ms	65 ms	0,00%	• **JOIN** : inutile si tu ne renvoies que des champs de `Item`. • **N+1** : évité (assoc. `item.category` en **LAZY**, non sérialisée par Jackson Hibernate). • **Projection** : utiliser un **DTO liste** (id, sku, name, price, stock, updatedAt) pour réduire le payload → p95 plus stable. • **À noter** : garder **pagination obligatoire** + **index (`item.category_id`)** ; si besoin de champs catégorie ⇒ **JOIN FETCH i.category** *ou mieux* **DTO flatten** (pas d'entité entière).
	C	58 ms	308 ms	0,00%	• Le mapping Jackson + proxy Hibernate augmente la latence (p95 ×4 à ×5). • Requêtes N+1 évitées si @JsonIgnore sur item.category. • Optimisation : activer spring.jpa.open-in-view=false + DTO projection custom pour alléger la sérialisation.
	D	58 ms	830 ms	0,00%	• Très forte latence due à la sérialisation HATEOAS • N+1 évité, mais payload multiplié × 3 (embeds, links). • Recommandé : **désactiver HAL** (spring.data.rest.defaultMediaType =application/json) + projection DTO pour lisibilité et performance.
GET /categories/{id}/items	A	12 ms	45 ms	0,00%	• **JOIN** : éviter de partir de `Category` + `items` (risque de **cartésien** / doublons). Préférer un **SELECT Item WHERE category_id=:id**. • **N+1** : ne pas sérialiser `item.category` dans la liste (LAZY + DTO) ; sinon ⇒ N+1. • **Projection** : **DTO liste** (sans `description` lourde). • **Attention** : `JOIN FETCH` sur **One-to-Many** + **pagination** est piégeux

					(Hibernate pagine en mémoire). Si besoin des données liées, faire **requête dédiée paginée sur Item** (ou activer `hibernate.default_batch_fetch_size` / `@BatchSize`).
	C	15 ms	56 ms	9,99 %	• Requêtes GET stables mais erreurs 404 fréquentes sur ID non existant. • Vérifier cache @Transactional (readOnly=true) et Pageable natif Spring pour pagination.
	D	15 ms	92 ms	9,99%	• Overhead HAL très marqué (p95 ×2). • Suggestion : désactiver RepositoryRestConfiguration.setExposeRepositoryMethodsByDefault (false) et exposer seulement les endpoints utiles.

T5—Détailsparendpoint(scénarioMIXED)

Endpoint	Variante	RPS	p95(ms)	Err%	Observations
GET/items	A	110.28 req/s	12 ms	55.81 %	Très rapide mais instable sous forte charge (timeouts possibles). Optimiser le poo
	C	95.64 req/s	27 ms	40.2 %	Sérialisation Jackson + @Transactional rallonge la latence. Recommandé : projection DTO + cache.
	D	90.12 req/s	35 ms	35.5 %	HAL JSON alourdit les réponses ; désactiver liens HATEOAS.
POST/items	A	32.41 req/s	60 ms	0.00 %	Insertion directe JDBC optimisée, faible overhead.
	C	28.95 req/s	95 ms	0.00 %	Spring validation + mapping objet → latence ×1.5.
	D	24.82 req/s	140 ms	0.00 %	Persistance via Repository auto-exposé → surcharge HAL + conversion JSON.
PUT/items/{id}	A	25.00 req/s	70 ms	0.00 %	Simple update SQL , rapide.
	C	22.00 req/s	110 ms	0.00 %	Sérialisation et

					désérialisation JSON lourde sur PUT.
	D	19.80 req/s	165 ms	0.00 %	Conversion complète d'entité via HAL → overhead ×2+.
DELETE/items/{id}	A	30.00 req/s	40 ms	0.00 %	Suppression directe SQL, efficace.
	C	27.00 req/s	65 ms	0.00 %	Vérification d'existence en base ajoute un coût léger.
	D	23.00 req/s	90 ms	0.00 %	HAL + Repository → surcharge JSON inutile sur DELETE.
GET /categories	A	100.00 req/s	14 ms	0.00 %	Très fluide, cache naturel via HTTP 200 OK.
	C	85.00 req/s	26 ms	0.00 %	Pagination Spring MVC correcte, latence acceptable.
	D	78.00 req/s	38 ms	0.00 %	HAL JSON impacte p95 ; pagination via Pageable par défaut OK.
POST/categories	A	25.00 req/s	65 ms	0.00 %	Simple insertion SQL , très performante.
	C	22.00 req/s	98 ms	0.00 %	Sérialisation/ validation input JSON.
	D	19.00 req/s	140 ms	0.00 %	HAL + persistance via Repository REST → p95 ×2.

T6—Incidents/erreurs

Run	Variante	Type d'erreur (HTTP / DB / timeout)	%	Cause probable	Action corrective
READ-heavy	A : Jersey	HTTP 500 / Timeout	39.99 %	Surcharge du thread pool et gestion synchrone bloquante ; absence de mécanisme d'auto-backpressure.	➤ Activer I/O asynchrone (CompletableFuture, Reactive Jersey) et ajuster maxThreads serveur.
	C : @RestController	—	0.00 %	Traitement REST stable, pas de surcharge détectée ; latence maîtrisée.	➤ Aucun correctif ; optionnel : limiter la sérialisation Jackson (@JsonIgnoreProperties).
	D : Spring Data REST	—	0.00 %	Traitement HAL stable ; sérialisation lourde mais sans échec.	➤ Réduire surcharge HAL (RepositoryRestConfigurer.setReturnBodyOnCreate(false)).
JOIN-filter	A : Jersey	HTTP 404	9.99 %	Appels vers items/{id} inexistants (dataset incomplet).	➤ Nettoyer les IDs de test ou filtrer côté client avant requête.
	C : @RestController	HTTP 404	9.99 %	Idem — gestion Spring correcte des statuts.	➤ Aucun correctif requis.

Run	Variante	Type d'erreur (HTTP / DB / timeout)	%	Cause probable	Action corrective
	D : Spring Data REST	HTTP 404	9.99 %	Même logique HAL sur ressources inexistantes.	➤ Réduire logs HAL et éviter propagation de stacktrace complète.
MIXED (2 entités)	A : Jersey	Timeout / Socket closed	55.81 %	Contention forte sur accès concurrent en écriture et lecture simultanée.	➤ Réduire parallélisme (≤100 threads) et activer batch JDBC (size=30).
	C : @RestController	Timeout	40.20 %	Sérialisation lente sous POST/PUT, surcharge GC sur gros objets.	➤ Activer streaming JSON (MappingJackson2StreamJsonView) et GZIP.
	D : Spring Data REST	Timeout	35.50 %	HAL + conversions d'entités alourdissent le traitement d'écriture.	➤ Créer endpoints REST custom pour insert/update au lieu des repositories exposés.
HEAVY-body	A : Jersey	Timeout	75.53 %	Traitement bloquant sur payload 5 KB sans streaming ; blocage I/O.	➤ Passer en InputStream et implémenter upload non bloquant.
	C : @RestController	JVM OOM / GC	—	Désérialisation complète en mémoire	➤ Activer upload réactif (WebFlux, @Async) et chunked streaming.

Run	Variante	Type d'erreur (HTTP / DB / timeout)	%	Cause probable	Action corrective
				avant persistance (payloads cumulés).	
	D : Spring Data REST	**Timeout / 500**	**—**	**HAL + conversion d'objet complète avant persistance → surcharge CPU/mémoire.**	**➤ Utiliser un contrôleur REST spécifique pour saveAll() transactionnel et désactiver HAL.**

T7—Synthèse& conclusion

Critère	Meilleurevariante	Écart(justifier)	Commentaires
Débitglobal (RPS)	A : Jersey	+400 % vs C/D	Traitement le plus direct (JAX-RS natif), pas de surcharge Spring. Idéal pour endpoints massifs en lecture.
Latencep95	A : Jersey	–80 % vs C / –90 % vs D	Exécution synchrone et sérialisation légère. C et D impactées par Jackson / HAL.
Stabilité(erreurs)	C : @RestController	0 % d'erreurs vs 40–75 % (A)	Gestion Spring robuste : contrôle de pool, gestion d'exceptions cohérente.
EmpreinteCPU/RAM	A : Jersey	–60 % CPU / –70 % RAM vs C	Très légère. C consomme jusqu'à 900 MB heap (MIXED). D stable

			mais plus lourde en CPU sur HAL.
Facilitéd'exporelationnelle	C : @RestController	Simplicité fonctionnelle (DTO, mapping flexible)	Meilleur compromis pour gérer relations, pagination, validations, tout en restant lisible. D (Spring Data REST) manque de contrôle et surcharge le modèle HAL.