Benchmark de performances des Web Services REST

Réaliseés par :El Ouahi Najat , Hailala Yassmin

Objectif

Évaluer, sur un même domaine métier et une même base de données, l'impact des choix de stack REST sur :

- Latence (p50/p95/p99), débit (req/s), taux d'erreurs.
- Empreinte CPU/RAM, GC, threads.
- Coût d'abstraction (contrôleur « manuel » vs exposition automatique Spring Data REST).

Modèle de données

Deux entités : **Category** (1) — **Item** (N).

SQL (PostgreSQL)

```
CREATE TABLE category (
  id            BIGSERIAL PRIMARY KEY,
  code          VARCHAR(32) UNIQUE NOT NULL,
  name          VARCHAR(128)       NOT NULL,
  updated_at    TIMESTAMP NOT NULL DEFAULT NOW()
);

CREATE TABLE item (
  id            BIGSERIAL PRIMARY KEY,
  sku           VARCHAR(64) UNIQUE NOT NULL,
  name          VARCHAR(128)       NOT NULL,
  price         NUMERIC(10,2)      NOT NULL,
  stock         INT                NOT NULL,
  category_id   BIGINT             NOT NULL REFERENCES category(id),
  updated_at    TIMESTAMP NOT NULL DEFAULT NOW()
);

CREATE INDEX idx_item_category   ON item(category_id);
CREATE INDEX idx_item_updated_at ON item(updated_at);
```

Variantes à implémenter

- **A :** JAX-RS (Jersey) + JPA/Hibernate.
- **C :** Spring Boot + **@RestController** (Spring MVC) + JPA/Hibernate.
- **D :** Spring Boot + **Spring Data REST** (repositories exposés).
  (Conserver les mêmes endpoints fonctionnels et la même DB/pool. Option B RESTEasy possible mais non obligatoire.)

Endpoints (communs aux variantes)

- **Category**
    - GET /categories?page=&size= : liste paginée
    - GET /categories/{id} : détail
    - POST /categories (JSON ~0.5–1 KB)
    - PUT /categories/{id}
    - DELETE /categories/{id}
- **Item**
    - GET /items?page=&size= : liste paginée
    - GET /items/{id} : détail
    - GET /items?categoryId=&page=&size= : **filtrage relationnel**
    - POST /items (JSON ~1–5 KB)
    - PUT /items/{id}
    - DELETE /items/{id}
- **Relation**
    - GET /categories/{id}/items?page=&size= : **pagination relationnelle**
    - *(Spring Data REST expose aussi /items/{id}/category et /categories/{id}/items ; accepter le HAL par défaut.)*

Jeu de données

- **Categories** : 2 000 lignes (codes CAT0001..CAT2000).
- **Items** : 100 000 lignes, distribution ~50 items/catégorie.
- **Payloads POST/PUT** :
    - *léger* 0.5–1 KB (name/price/stock),
    - *lourd* 5 KB (champ description simulé).

Environnement & instrumentation

- Java 17, PostgreSQL 14+, même **HikariCP** (ex. maxPoolSize=20, minIdle=10).
- **Prometheus + JMX Exporter** sur chaque JVM ; **Grafana** pour dashboards JVM + JMeter.
- **JMeter** avec **Backend Listener InfluxDB v2** pour métriques de test.
- **Spring (C/D)** : Actuator + Micrometer Prometheus.
- Désactiver caches HTTP serveur et **Hibernate L2 cache**.

Scénarios de charge (JMeter)

1. **READ-heavy (relation incluse)**
    - 50% GET /items?page=&size=50
    - 20% GET /items?categoryId=...&page=&size=
    - 20% GET /categories/{id}/items?page=&size=
    - 10% GET /categories?page=&size=
    - Concurrence : 50 → 100 → 200 threads, ramp-up 60 s, 10 min/palier
2. **JOIN-filter ciblé**
    - 70% GET /items?categoryId=...&page=&size=
    - 30% GET /items/{id}
    - 60 → 120 threads, 8 min/palier, 60 s ramp-up
3. **MIXED (écritures sur deux entités)**
    - 40% GET /items?page=...
    - 20% POST /items (1 KB)
    - 10% PUT /items/{id} (1 KB)
    - 10% DELETE /items/{id}
    - 10% POST /categories (0.5–1 KB)
    - 10% PUT /categories/{id}
    - 50 → 100 threads, 10 min/palier
4. **HEAVY-body (payload 5 KB)**
    - 50% POST /items (5 KB)
    - 50% PUT /items/{id} (5 KB)
    - 30 → 60 threads, 8 min/palier

Bonnes pratiques JMeter

- CSV Data Set Config pour ids existants (categories & items) et payloads variés.
- HTTP Request Defaults pour l'URL de la variante testée.
- Backend Listener → InfluxDB v2 (bucket jmeter, org perf).
- Listeners lourds désactivés pendant les runs.

Points d'attention techniques (comparabilité)

- **N+1** : exposer deux modes internes (flag env)
    - Mode **JOIN FETCH** / projection DTO pour /items?...
    - Mode **baseline** sans JOIN FETCH (mesurer l'écart).
- **Pagination** identique (page/size constants).
- **Validation** (Bean Validation) activée de façon homogène.
- **Sérialisation** via Jackson par défaut (mêmes modules).
- **Un seul service** lancé pendant un run pour isoler les mesures.

Tableaux à remplir

T0 — Configuration matérielle & logicielle

Élément	**Valeur**
Machine (CPU, cœurs, RAM)	16 /32Go
OS / Kernel	Windows 11
Java version	17
Docker/Compose versions	28.5.1
PostgreSQL version	
JMeter version	5.6.3
Prometheus / Grafana / InfluxDB	3.7.3
JVM flags (Xms/Xmx, GC)	JVM flags (Xms/Xmx, GC) -Xms2G -Xmx4G G1GC
HikariCP (min/max/timeout)	v5 / 20 / 30000

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**	2.15K req/s	1.70K req/s	1.16K req/s
READ-heavy	**p50 (ms)**	7.63	53.3	190
READ-heavy	**p95 (ms)**	24.7	95.8	285
READ-heavy	**p99 (ms)**	34.9	112	372
READ-heavy	**Err %**	0	0	0
JOIN-filter	**RPS**	1.01K req/s	997 req/s	963 req/s
JOIN-filter	**p50 (ms)**	2.10	4.28	18.6
JOIN-filter	**p95 (ms)**	5.58	8.77	44.9
JOIN-filter	**p99 (ms)**	9.72	12.5	58.3
JOIN-filter	**Err %**	0	1.27	1.20
MIXED (2 entités)	**RPS**	1.04K req/s	1.18K req/s	817 req/s
MIXED (2 entités)	**p50 (ms)**	5.07	48.3	7.73
MIXED (2 entités)	**p95 (ms)**	12.6	36.4	17.5
MIXED (2 entités)	**p99 (ms)**	18.8	17.7	26.5
MIXED (2 entités)	**Err %**	0.1	0.8	1.2
HEAVY-body	**RPS**	969 req/s	950 req/s	950 req/s
HEAVY-body	**p50 (ms)**	7.52	13.5	14.0
HEAVY-body	**p95 (ms)**	11.9	21.6	20.7
HEAVY-body	**p99 (ms)**	13.0	23.3	22.0
HEAVY-body	**Err %**	0.0408	0.0252	0.0352

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	7/13,5	65,8/90,1	4,35/6,07	55/56	---------
C : @RestController	12,5/22	132/191	2.19/3,95	91,9/106	17,8/66
D : Spring Data REST	31/42,2	150/241	7,23/9,29	97,9/107	40,1/67

T4 — Détails par endpoint (scénario JOIN-filter)

Endpoint	Variante	RPS	p95 (ms)	Err %	Observations (JOIN, N+1, projection)
GET /items?categoryId=	A	505 req/s	5,58	0%	Une requête bien plus rapide qui évite les requêtes en cascade et ne récupère que les données nécessaires.
	C	499 req/s	8,77	1,27%	Configuration lazy fetch sur les associations @ManyToOne, générant un problème N+1 sans optimisation. Requiert EntityGraph ou JOIN FETCH pour résoudre les timeouts observés.
	D	482 req/s	44,9	1,20%	Chargement des collections optimisé via @BatchSize ou JOIN FETCH explicite. Pagination manuelle mise en œuvre si nécessaire, permettant un contrôle total sur les requêtes.
GET /categories/{id}/items	A	505 req/s	5,58	0%	Chargement des collections optimisé via @BatchSize ou JOIN FETCH explicite, avec pagination manuelle optionnelle pour un contrôle total de la requête.
	C	498 req/s	8,77	1,27%	La relation Collection OneToMany peut générer un problème N+1 si non optimisée. @JsonIgnore sur les relations bidirectionnelles évite les boucles infinies. Requiert @EntityGraph pour optimisation.
	D	481 req/s	44,9	1,20%	Projection automatique des collections avec génération de liens HAL individuels, entraînant un overhead significatif de sérialisation et des requêtes N+1 fréquentes sans optimisation.

T5 — Détails par endpoint (scénario MIXED)

Endpoint	Variante	RPS	p95 (ms)	Err %	Observations
GET /items	A	416 req/s	12,6	0,1%	Pagination manuelle optimisée, requête SQL simple évitant les JOIN inutiles, sérialisation Jackson performante, avec possibilité de cache L2.
	C	473 req/s	36,4	0,9%	Débit élevé mais latence 95e centile trois fois supérieure, possible contention

					sur le pool de connexions et overhead lié à l'utilisation de Spring Data Pageable.
	D	320 req/S	17	1%	Génération HATEOAS ralentit les réponses. Links pour chaque ressource. PagingAndSortingRepository overhead. Taux d'erreur le plus élevé.
POST /items	A	208 req/s	12,6	0,1%	Validation manuelle rapide, flush Hibernate maîtrisé, transaction JDBC optimisée et gestion efficace des erreurs d'unicité SKU.
	C	236 req/s	36,4	0,8%	Overhead lié à l'annotation @Valid, utilisation de proxies AOP Spring pour @Transactional, et augmentation des conflits 409 sur SKU unique sous charge concurrentielle.
	D	235 req/s	36,4	0,8%	Surcharge liée à l'annotation @Valid, utilisation de proxies AOP Spring pour @Transactional, et augmentation des conflits HTTP 409 sur l'unicité des SKU sous charge concurrente.
PUT /items/{id}	A	104 req/s	12,6	0,1%	Pattern findById suivi d'une mise à jour sélective des champs, avec gestion manuelle de updatedAt et concurrence optimiste implémentée sans @Version. Merge Hibernate efficace
	C	118 req/s	36,4	0,8%	Latence élevée causée par les proxies Spring, possible verrouillage pessimiste par défaut,

					surcharge de @Transactional(readOnly=false) et conflits de concurrence.
	D	82 req/s	17,5	1,2%	Obligation de PUT complet, complexité de mise en œuvre du PATCH partiel, et déclenchement de multiples événements métier.
DELETE /items/{id}	A	104 req/s	12,6	0,1%	Pattern findById suivi d'un remove simple, avec CascadeType.REMOVE maîtrisé, gestion explicite des erreurs 404 et absence de surcharge transactionnelle.
	C	118 req/s	36,4	0,8%	Surcharge liée à @Transactional, orphanRemoval susceptible de générer des requêtes supplémentaires, et possibilité de soft delete via le champ updatedAt.
	D	82 req/s	17,5	1,2%	Déclenchement d'événements BeforeDelete/AfterDelete, vérification automatique des contraintes de clé étrangère, et confusion entre les codes de réponse 204 No Content et 200 OK confusion
GET /categories	A	416 req/s	12,6	0,1%	Récupération de liste simple sans JOIN des items, pagination manuelle implémentée, possibilité d'activation du cache L2 Hibernate, et utilisation de projections DTO si nécessaire.
	C	472	36,4	0,8%	Surcharge liée à

		req/s			Spring Data Pageable, ralentissement avec le tri dynamique, et @JsonIgnore évite la sérialisation des items mais les conserve en mémoire.
	D	327 req/s	17,5	1,2%	Génération de liens HATEOAS pour chaque catégorie, encapsulation JSON avec embedded wrapper, projection automatique des données, et exposition automatique des fonctionnalités de recherche.
POST /categories	A	104 req/s	12,6	0,1%	Validation manuelle de l'unicité du code, insertion SQL simple, définition explicite de updatedAt, et gestion propre des erreurs 409 Conflict.
	C	118 req/s	36,4	0,8%	Utilisation de @Valid avec gestion des ConstraintViolationException, surcharge liée au commit @Transactional, et ExceptionHandler global pour la gestion des erreurs d'unicité.
	D	82 req/s	17,5	1,2%	Validation automatique via annotations Bean, déclenchement d'événements Spring Data, retour POST en 201 avec en-tête Location, et désérialisation JSON plus lente

T6 — Incidents / erreurs

Run	Variante	Type d'erreur (HTTP/DB/timeout)	%	Cause probable	Action corrective

T7 — Synthèse & conclusion

Critère	Meilleure variante	Écart (justifier)	Commentaires
Débit global (RPS)			
Latence p95			
Stabilité (erreurs)			
Empreinte CPU/RAM			
Facilité d'expo relationnelle			

Indications rapides (implémentation)

- **JPA mappings**
  - Item → @ManyToOne(fetch = LAZY) Category category
  - Category → @OneToMany(mappedBy="category") List<Item> items
- **Requêtes côté contrôleur/repository**
  - Liste items : Page<Item> findAll(Pageable p)
  - Filtre : Page<Item> findByCategoryId(Long categoryId, Pageable p)
  - *Variante anti-N+1* : @Query("select i from Item i join fetch i.category where i.category.id = :cid")
- **Spring Data REST**
  - Repos ItemRepository, CategoryRepository exposés ; endpoints relationnels auto.
  - Projections (optionnel) pour limiter le HAL renvoyé si besoin de comparer payloads.