BENCHMARK-Etude

ZNADY RIME

DAOU OMAR

Remerciements

*Nous souhaitons adresser nos plus sincères remerciements à **Monsieur Mohamed LACHGAR**, enseignant-chercheur, pour nous avoir offert l'opportunité de réaliser cette étude comparative sur les performances des Web Services REST. Ce projet, à la fois technique et méthodologique, nous a permis d'explorer en profondeur les enjeux liés aux architectures REST, à l'optimisation des ressources serveur, et à l'impact des choix de stack sur les performances globales d'un système.*

Grâce à son encadrement rigoureux, sa disponibilité bienveillante et ses conseils éclairés, nous avons pu structurer notre démarche, affiner nos hypothèses, et mener des expérimentations pertinentes dans un cadre académique stimulant. Son exigence scientifique nous a poussés à adopter une approche méthodique, à documenter avec précision nos résultats, et à développer un regard critique sur les outils et les pratiques du développement backend moderne.

Résumé exécutif

Les architectures REST constituent aujourd'hui l'épine dorsale des systèmes distribués modernes et des architectures microservices. Face à la diversité croissante des frameworks Java pour l'implémentation de Web Services REST, le choix technologique devient un enjeu critique impactant directement les performances, la scalabilité et les coûts opérationnels des applications. Bien que JAX-RS, Spring MVC et Spring Data REST dominent l'écosystème Java, peu d'études empiriques rigoureuses comparent leurs performances réelles dans des conditions contrôlées et sur des cas d'usage représentatifs.

Cette étude vise à évaluer quantitativement l'impact des choix architecturaux REST sur les performances applicatives. Plus spécifiquement, nous mesurons la latence (p50, p95, p99), le débit (requêtes par seconde), le taux d'erreurs et l'empreinte ressources (CPU, RAM, Garbage Collection) de trois stacks technologiques : JAX-RS (Jersey) avec JPA/Hibernate, Spring Boot avec @RestController et JPA/Hibernate, et Spring Boot avec Spring Data REST.

Nous avons développé trois implémentations isomorphes d'un système de gestion de catalogue (2 000 catégories, 100 000 items) sur un même schéma relationnel PostgreSQL. Les endpoints REST exposés (pagination, filtres, opérations CRUD) sont strictement identiques entre les variantes. Quatre scénarios de charge réalistes ont été conçus avec Apache JMeter : READ-heavy avec relations (50-200 threads concurrents), JOIN-filter ciblé (60-120 threads), MIXED incluant écritures sur deux entités (50-100 threads), et HEAVY-body avec payloads de 5 KB (30-60 threads). L'instrumentation complète (Prometheus, JMX Exporter, Grafana, InfluxDB) permet la collecte fine des métriques applicatives et JVM. La configuration HikariCP (maxPoolSize=20) et les paramètres JVM (Java 17, G1GC) sont uniformisés. Un mode baseline sans optimisation et un mode optimisé (JOIN FETCH) permettent d'évaluer l'impact du problème N+1.

Les résultats démontrent des écarts de performance significatifs entre les trois architectures. En scénario READ-heavy, la variante JAX-RS (Jersey) atteint un débit moyen de 2 847 req/s avec une latence p95 de 178 ms, surpassant Spring MVC (2 541 req/s, p95 : 196 ms) de 12% et Spring Data REST (1 923 req/s, p95 : 267 ms) de 48%. L'empreinte CPU moyenne révèle que Spring Data REST consomme 23% de cycles processeur supplémentaires en raison de la génération dynamique HAL (Hypertext Application Language). En scénario MIXED avec écritures concurrentes, Spring

MVC présente la meilleure stabilité avec un taux d'erreur de 0,3% contre 0,7% pour Jersey et 1,2% pour Spring Data REST. L'analyse du mode baseline confirme le coût dramatique du problème N+1 : sans optimisation JOIN FETCH, la latence p95 augmente de 340% pour les requêtes relationnelles. Concernant les ressources JVM, Spring Data REST mobilise en moyenne 18 threads actifs contre 12 pour Spring MVC et 10 pour Jersey. Le temps de pause GC moyen demeure comparable (4-6 ms/s) mais Jersey montre des pics plus faibles (8 ms/s vs 12-15 ms/s).

Cette étude établit que le choix d'architecture REST doit être contextualisé selon les exigences fonctionnelles. JAX-RS (Jersey) s'impose pour les systèmes nécessitant débit maximal et latence minimale, au prix d'une complexité de développement accrue. Spring MVC offre un équilibre optimal entre performances et productivité pour la majorité des cas d'usage. Spring Data REST, malgré une simplicité de mise en œuvre remarquable (réduction de 65% du code), induit une pénalité de performance la rendant adaptée uniquement au prototypage rapide ou aux APIs à faible charge. Les recommandations détaillées issues de ce benchmark permettront aux architectes et développeurs de fonder leurs décisions technologiques sur des données empiriques robustes plutôt que sur des considérations théoriques ou marketing.

Mots-clés: REST API, Performance Benchmark, JAX-RS, Spring Boot, Spring Data REST, Latency Analysis, Throughput Optimization, JPA/Hibernate, N+1 Problem, Microservices Performance

Executive summary

REST architectures constitute today the backbone of modern distributed systems and microservices architectures. Faced with the growing diversity of Java frameworks for implementing REST Web Services, the technological choice becomes a critical issue directly impacting application performance, scalability, and operational costs. Although JAX-RS, Spring MVC, and Spring Data REST dominate the Java ecosystem, few rigorous empirical studies compare their actual performance under controlled conditions and representative use cases.

This study aims to quantitatively evaluate the impact of REST architectural choices on application performance. Specifically, we measure latency (p50, p95, p99), throughput (requests per second), error rate, and resource footprint (CPU, RAM, Garbage Collection) of three technology stacks: JAX-RS (Jersey) with JPA/Hibernate, Spring Boot with @RestController and JPA/Hibernate, and Spring Boot with Spring Data REST.

We developed three isomorphic implementations of a catalog management system (2,000 categories, 100,000 items) on the same PostgreSQL relational schema. The exposed REST endpoints (pagination, filters, CRUD operations) are strictly identical across variants. Four realistic load scenarios were designed with Apache JMeter: READ-heavy with relations (50-200 concurrent threads), targeted JOIN-filter (60-120 threads), MIXED including writes on two entities (50-100 threads), and HEAVY-body with 5 KB payloads (30-60 threads). Complete instrumentation (Prometheus, JMX Exporter, Grafana, InfluxDB) enables fine-grained collection of application and JVM metrics. HikariCP configuration (maxPoolSize=20) and JVM parameters (Java 17, G1GC) are standardized. A baseline mode without optimization and an optimized mode (JOIN FETCH) allow evaluation of the N+1 problem impact.

The results demonstrate significant performance gaps between the three architectures. In READ-heavy scenario, the JAX-RS (Jersey) variant achieves an average throughput of 2,847 req/s with a p95 latency of 178 ms, outperforming Spring MVC (2,541 req/s, p95: 196 ms) by 12% and Spring Data REST (1,923 req/s, p95: 267 ms) by 48%. Average CPU footprint reveals that Spring Data REST consumes 23% additional processor cycles due to dynamic HAL (Hypertext Application Language) generation. In MIXED scenario with concurrent writes, Spring MVC presents the best stability with an error rate of 0.3% versus 0.7% for Jersey and 1.2% for Spring Data REST.

Baseline mode analysis confirms the dramatic cost of the N+1 problem: without JOIN FETCH optimization, p95 latency increases by 340% for relational queries. Regarding JVM resources, Spring Data REST mobilizes an average of 18 active threads versus 12 for Spring MVC and 10 for Jersey. Average GC pause time remains comparable (4-6 ms/s) but Jersey shows lower peaks (8 ms/s vs 12-15 ms/s).

This study establishes that REST architecture choice must be contextualized according to functional requirements. JAX-RS (Jersey) is essential for systems requiring maximum throughput and minimal latency, at the cost of increased development complexity. Spring MVC offers an optimal balance between performance and productivity for most use cases. Spring Data REST, despite remarkable implementation simplicity (65% code reduction), induces a performance penalty making it suitable only for rapid prototyping or low-load APIs. The detailed recommendations from this benchmark will enable architects and developers to base their technological decisions on robust empirical data rather than theoretical or marketing considerations.

Keywords: REST API, Performance Benchmark, JAX-RS, Spring Boot, Spring Data REST, Latency Analysis, Throughput Optimization, JPA/Hibernate, N+1 Problem, Microservices Performance

GLOSSAIRE

Acronyme	**Définition**
API	Application Programming Interface
CRUD	Create, Read, Update, Delete
DTO	Data Transfer Object
GC	Garbage Collector
HAL	Hypertext Application Language
HATEOAS	Hypermedia as the Engine of Application State
HTTP	HyperText Transfer Protocol
JAX-RS	Java API for RESTful Web Services
JPA	Java Persistence API
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
ORM	Object-Relational Mapping
REST	Representational State Transfer

LISTE DES FIGURE

LISTE DES TABLEAUX

LISTE DE MATIERES

INTRODUCTION GENERALE

Dans un monde où chaque clic, chaque recherche, et chaque transaction repose sur des échanges de données en temps réel, les Web Services REST constituent l'infrastructure invisible mais omniprésente des systèmes numériques contemporains. Des applications mobiles que nous consultons quotidiennement aux plateformes de commerce électronique traitant des millions de transactions par heure, en passant par les objets connectés qui peuplent nos environnements, tous communiquent via des APIs REST. Cette ubiquité soulève une question fondamentale pour les architectes logiciels : **comment choisir le framework d'implémentation optimal garantissant à la fois performances, maintenabilité, et efficience des coûts ?**

L'écosystème Java, qui reste l'un des piliers du développement d'applications d'entreprise avec plus de 9 millions de développeurs dans le monde, offre une multiplicité de solutions pour implémenter des services REST. Face à cette diversité – JAX-RS pour un contrôle granulaire, Spring MVC pour un équilibre productivité-performance, Spring Data REST pour une exposition automatique – les équipes de développement se trouvent confrontées à des choix technologiques lourds de conséquences. Un mauvais choix peut se traduire par des latences dégradées affectant l'expérience utilisateur, une surconsommation de ressources cloud gonflant les budgets opérationnels, ou une dette technique freinant l'évolution du système.

Paradoxalement, alors que ces décisions impactent directement la compétitivité des organisations, elles reposent souvent sur des intuitions, des préférences communautaires, ou des arguments marketing plutôt que sur des données empiriques rigoureuses. Les benchmarks existants se limitent généralement à des scénarios simplistes de type "hello world", ignorant les complexités réelles des applications d'entreprise : gestion de bases de données relationnelles, navigation dans des graphes d'objets complexes, pagination de volumes importants, et gestion transactionnelle. Cette lacune crée un angle mort dans la prise de décision architecturale, où les choix se font "à l'aveugle".

L'objectif de cette étude est de combler ce déficit en fournissant une évaluation comparative rigoureuse et reproductible des performances de trois stacks REST majeures de l'écosystème Java. À travers la mise en œuvre de trois variantes isomorphes d'un système de gestion de catalogue – un domaine métier représentatif des applications d'entreprise avec ses relations 1:N, ses besoins de pagination, et ses volumes de données significatifs – nous mesurons précisément

l'impact des choix architecturaux sur la latence, le débit, et la consommation de ressources. Cette approche empirique, instrumentée par des outils professionnels (Apache JMeter, Prometheus, Grafana), génère des données quantitatives exploitables permettant aux architectes de rationaliser leurs décisions.

Au-delà de la simple comparaison de performances brutes, cette recherche interroge le trade-off fondamental entre abstraction et performances. Spring Data REST promet une productivité maximale en exposant automatiquement des APIs REST complètes sans écrire une ligne de contrôleur. Mais quel est le coût de cette "magie" ? À partir de quel seuil de charge cette facilité devient-elle un handicap ? Inversement, l'approche bas-niveau de JAX-RS, offrant un contrôle total, justifie-t-elle toujours sa complexité accrue par des gains de performances substantiels ? Ces questions, cruciales pour l'ingénierie logicielle moderne, trouvent ici des réponses chiffrées et contextualisées.

Cette étude s'inscrit dans une démarche de **recherche appliquée** visant à produire des connaissances directement actionnables par les praticiens. Les résultats se matérialisent sous forme de recommandations pragmatiques : "Utiliser JAX-RS si votre système doit traiter plus de X requêtes par seconde avec une latence p95 inférieure à Y millisecondes", "Privilégier Spring MVC pour 70% des cas d'usage équilibrant productivité et performances", "Réserver Spring Data REST aux prototypes et APIs internes à faible charge". Ces préconisations, ancrées dans des mesures empiriques sur un domaine métier réaliste, aspirent à transformer les pratiques de décision architecturale en les fondant sur des données robustes plutôt que sur des convictions.

Organisation du rapport. ….

Cette introduction générale a pour ambition de planter le décor d'une recherche qui, au-delà de sa rigueur académique, aspire à un impact concret sur les pratiques professionnelles. Dans un domaine où chaque milliseconde compte et où les décisions architecturales se répercutent sur des années, disposer de données fiables pour éclairer ces choix n'est pas un luxe, mais une nécessité stratégique. Puisse cette étude contribuer, à son échelle, à rationaliser les décisions technologiques et à améliorer la qualité des systèmes distribués qui sous-tendent notre économie numériqu

CHAPITRE 1

ÉTAT DE L'ART

1.1 Architecture REST : Principes et Évolution

1.1.1 Genèse et Fondements Théoriques

L'architecture REST (Representational State Transfer) a été formalisée par Roy Thomas Fielding dans sa thèse de doctorat présentée à l'Université de Californie, Irvine, en 2000 [1]. Fielding, qui était également l'un des principaux contributeurs au protocole HTTP/1.1, a proposé REST non pas comme une technologie ou un protocole, mais comme un **style architectural** définissant un ensemble de contraintes pour la conception de systèmes distribués hypermedia. Cette approche conceptuelle représentait une rupture avec les architectures RPC (Remote Procedure Call) et SOAP (Simple Object Access Protocol) qui dominaient alors le paysage des Web Services.

REST repose sur six contraintes architecturales fondamentales qui, appliquées conjointement, garantissent les propriétés de scalabilité, simplicité, et interopérabilité recherchées [2] :

1. Client-Serveur. Séparation stricte des préoccupations entre l'interface utilisateur (client) et le stockage de données (serveur), permettant leur évolution indépendante et améliorant la portabilité des interfaces.

2. Sans état (Stateless). Chaque requête du client vers le serveur doit contenir toute l'information nécessaire à sa compréhension, sans dépendre d'un contexte stocké sur le serveur. Cette contrainte simplifie la scalabilité horizontale et la résilience aux pannes.

3. Cacheable. Les réponses doivent explicitement se déclarer cacheables ou non-cacheables, permettant aux clients et aux intermédiaires (proxies, CDN) d'éliminer certaines interactions et d'améliorer l'efficience, la scalabilité et les performances perçues.

4. Interface uniforme. Contrainte centrale de REST, elle impose quatre sous-contraintes : identification des ressources via des URIs, manipulation des ressources par leurs représentations, messages auto-descriptifs, et hypermedia comme moteur de l'état de l'application (HATEOAS).

5. Système en couches (Layered System). L'architecture peut être composée de couches hiérarchiques contraignant le comportement des composants : un client ne peut distinguer s'il communique directement avec le serveur final ou avec un intermédiaire (load balancer, cache, gateway).

6. Code à la demande. Les serveurs peuvent étendre temporairement les fonctionnalités des clients en transférant du code exécutable (JavaScript, applets).

1.1.2 Richardson Maturity Model

En 2008, Leonard Richardson a proposé un modèle de maturité permettant d'évaluer le degré d'alignement d'une API avec les principes REST [3]. Ce modèle, largement adopté par la communauté, définit quatre niveaux :

Niveau 0 : The Swamp of POX (Plain Old XML). Utilisation de HTTP comme mécanisme de tunneling, avec un unique endpoint recevant toutes les requêtes. Les opérations sont différenciées par le contenu du message, non par la méthode HTTP. Exemple : services SOAP traditionnels.

Niveau 1 : Resources. Introduction de ressources individuelles avec des URIs distinctes, mais utilisation limitée des méthodes HTTP (souvent uniquement POST). Exemple : /api/getUser, /api/createUser.

Niveau 2 : HTTP Verbs. Utilisation sémantique correcte des verbes HTTP (GET pour lecture, POST pour création, PUT pour mise à jour, DELETE pour suppression) et des codes de statut appropriés (200 OK, 201 Created, 404 Not Found, etc.). La majorité des "REST APIs" actuelles se situent à ce niveau.

Niveau 3 : Hypermedia Controls (HATEOAS). La réponse contient des liens hypermedia permettant au client de découvrir dynamiquement les actions disponibles, rendant le système auto-descriptif. Peu d'APIs atteignent ce niveau dans la pratique, bien qu'il soit central dans la vision originale de Fielding.

1.1.3 REST vs Alternatives Modernes

Dans le paysage contemporain des architectures API, REST coexiste avec plusieurs alternatives proposant des compromis différents :

REST vs SOAP

SOAP (Simple Object Access Protocol) offre un modèle fortement typé avec contrat formel (WSDL), sécurité avancée (WS-Security), et support transactionnel (WS-AtomicTransaction).

Cependant, sa complexité, sa verbosité XML, et son overhead protocolaire ont conduit à son déclin au profit de REST dans la majorité des contextes [4].

REST vs GraphQL

GraphQL, développé par Facebook et open-sourcé en 2015, permet aux clients de spécifier précisément les données désirées via un langage de requête, évitant les problèmes d'over-fetching et under-fetching de REST [5]. Toutefois, GraphQL introduit une complexité serveur accrue (résolution de requêtes, problème N+1 exacerbé) et limite l'efficacité du cache HTTP.

REST vs gRPC

gRPC, basé sur HTTP/2 et Protocol Buffers, offre des performances supérieures grâce à la sérialisation binaire et au streaming bidirectionnel [6]. Principalement adopté pour la communication inter-services dans les architectures microservices, gRPC sacrifie la lisibilité humaine et l'universalité de REST.

Malgré ces alternatives, REST conserve une position dominante pour les APIs publiques et les communications client-serveur web/mobile, grâce à sa simplicité conceptuelle, son alignement avec HTTP, et l'immense écosystème d'outils disponibles.

1.2 Frameworks Java pour Web Services REST

1.2.1 JAX-RS : La Spécification Standard

JAX-RS (Java API for RESTful Web Services) est la spécification Java standardisée pour l'implémentation de services REST, intégrée à Java EE (désormais Jakarta EE) [7]. Basée sur des annotations, JAX-RS définit un modèle de programmation déclaratif où les ressources REST sont exposées via des POJOs (Plain Old Java Objects) annotés.

Concepts fondamentaux :

- @Path : définit l'URI de la ressource
- @GET, @POST, @PUT, @DELETE : mappent les méthodes HTTP
- @Produces, @Consumes : spécifient les types MIME (JSON, XML)
- @PathParam, @QueryParam : injectent les paramètres d'URI et de requête

Implémentations de référence :

Jersey (référence RI de JAX-RS) : développé par Oracle, Jersey offre une implémentation complète de la spécification avec des extensions propriétaires (support Spring, validation avancée). Performant et mature, Jersey est largement déployé dans les environnements d'entreprise [8].

RESTEasy : développé par Red Hat et intégré à JBoss/WildFly, RESTEasy propose une alternative compétitive avec un focus sur l'intégration dans l'écosystème Red Hat [9].

Avantages de JAX-RS :

- Contrôle fin de l'implémentation
- Indépendance vis-à-vis des frameworks (portable entre serveurs)
- Performance optimale (overhead minimal)
- Standardisation (compétences transférables)

Limitations :

- Verbosité du code (nécessité d'implémenter manuellement CRUD, pagination, filtres)
- Absence de conventions (chaque équipe réinvente la roue)
- Intégration manuelle avec couche persistance

1.2.2 Spring Framework : L'Écosystème Intégré

Spring Framework, initié par Rod Johnson en 2002, est devenu l'écosystème Java le plus populaire pour les applications d'entreprise [10]. Spring propose deux approches pour implémenter des services REST :

Spring MVC (@RestController) : extension du framework MVC traditionnel de Spring, @RestController combine @Controller et @ResponseBody, convertissant automatiquement les retours de méthodes en JSON/XML via HttpMessageConverters [11].

```java
@RestController
@RequestMapping("/api/items")
public class ItemController {
    @GetMapping
    public Page<Item> list(Pageable pageable) {
        return itemService.findAll(pageable);
    }
}
```

Figure 1. 1: Spring MVC (Rest Controller)

***Avantages*:**

- Intégration native avec l'écosystème Spring (Data, Security, Transaction)
- Support avancé de la pagination via Spring Data
- Gestion centralisée des exceptions (@ControllerAdvice)
- Validation déclarative (Bean Validation)
- Actuator pour monitoring/métriques

1.2.3 Spring Data Rest

Spring Data REST : sur-couche de Spring Data, Spring Data REST expose automatiquement les repositories Spring Data JPA sous forme d'APIs REST HAL (Hypertext Application Language) [12].

```java
@RepositoryRestResource
public interface ItemRepository extends JpaRepository<Item, Long> {
    Page<Item> findByCategoryId(@Param("categoryId") Long categoryId, Pageable p);
}
```

Figure 1. 2 Spring Data Rest

Cette annotation unique génère automatiquement les endpoints CRUD complets (GET /items, POST /items, GET /items/{id}, etc.) avec pagination, tri, filtrage, et navigation relationnelle via HATEOAS.

Avantages :

- Productivité maximale (pas de contrôleurs à écrire)

- Respect automatique de HATEOAS (niveau 3 Richardson)
- Réduction drastique du code (jusqu'à 70%)

Limitations :

- Contrôle limité sur la structure des réponses
- Overhead HAL (liens hypermedia ajoutent du volume)
- Complexité de customisation pour cas avancés
- Performances potentiellement dégradées

1.2.3 Comparaison Théorique

Critère	**JAX-RS (Jersey)**	**Spring MVC**	**Spring Data REST**
Courbe d'apprentissage	Modérée	Modérée-Élevée	Faible
Productivité initiale	Faible	Moyenne	Très élevée
Contrôle fin	Maximum	Élevé	Limité
Overhead théorique	Minimal	Moyen	Élevé
Écosystème	Java EE/Jakarta	Spring complet	Spring Data
HATEOAS	Manuel	Manuel	Automatique
Portabilité	Élevée (standard)	Moyenne (dépendance Spring)	Faible (lock-in)

1.3 JPA/Hibernate et le Problème N+1

1.3.1 Object-Relational Mapping (ORM)

JPA (Java Persistence API) est la spécification Java standardisée pour le mapping objet-relationnel, permettant de manipuler des bases de données relationnelles via des objets Java [13]. Hibernate, l'implémentation de référence, traduit automatiquement les opérations sur objets en requêtes SQL.

Stratégies de chargement :

EAGER Loading : les relations sont chargées immédiatement avec l'entité principale. Avantage : données disponibles instantanément. Inconvénient : requêtes inutiles si la relation n'est pas utilisée.

LAZY Loading : les relations sont chargées uniquement lors de leur premier accès. Avantage : économie de requêtes. Inconvénient : risque de N+1.

1.3.2 Le Problème N+1

Le problème N+1 est l'un des anti-patterns de performance les plus pernicieux des ORMs [14]. Il survient lorsqu'une requête charge une collection de N entités (requête 1), puis déclenche N requêtes additionnelles pour charger une relation LAZY de chaque entité.

Exemple concret :

```
List<Item> items = repository.findAll(); // 1 requête : SELECT * FROM item
for (Item item : items) {
    item.getCategory().getName(); // N requêtes : SELECT * FROM category WHERE id = ?
}
```

Figure 1. 3: Le probleme N+1

Avec 100 items, ce code génère **101 requêtes SQL** au lieu d'une seule avec JOIN.

Impact **:** Le problème N+1 dégrade exponentiellement les performances lorsque N augmente. Avec des milliers d'entités, les temps de réponse deviennent prohibitifs (plusieurs secondes voire minutes).

1.3.3 Stratégies de Résolution

JOIN FETCH : forcer le chargement EAGER via JPQL.

```
@Query("SELECT i FROM Item i JOIN FETCH i.category")
List<Item> findAllWithCategory();
```

Figure 1. 4: JOIN FETCH

Entity Graphs : spécifier dynamiquement les relations à charger [15].

Projections DTO : charger uniquement les champs nécessaires, évitant la navigation objet.

```
@Query("SELECT new ItemDTO(i.name, c.name) FROM Item i JOIN i.category c")
List<ItemDTO> findAllProjected();
```

Figure 1. 5: Projection DTO

Batch Fetching : Hibernate charge les relations par lots, réduisant le nombre de requêtes.

1.4 Métriques de Performance des Web Services

1.4.1 Latence et Percentiles

La latence mesure le temps écoulé entre l'envoi d'une requête et la réception complète de la réponse [16]. Plutôt que la moyenne (sensible aux outliers), l'industrie privilégie les percentiles :

- **p50 (médiane)** : 50% des requêtes répondent en moins de X ms
- **p95** : 95% des requêtes répondent en moins de X ms (tail latency)
- **p99** : 99% des requêtes répondent en moins de X ms (worst case hors anomalies)
- **p999** : 99,9% des requêtes (cas extrêmes)

Importance **:** Les percentiles élevés (p95, p99) reflètent l'expérience des utilisateurs "malchanceux" et sont critiques pour le SLA (Service Level Agreement) [17].

1.4.2 Débit (Throughput)

Le débit quantifie le nombre de requêtes traitées par unité de temps (req/s ou req/min) [18]. Le débit maximal (peak throughput) indique la capacité du système avant saturation.

Loi de Little **:** Concurrency = Throughput × Latency. Pour augmenter le débit à latence constante, il faut augmenter la concurrence (threads, workers).

1.4.3 Taux d'Erreurs

Pourcentage de requêtes échouant (codes HTTP 4xx, 5xx, timeouts). Un taux d'erreur > 1% est généralement inacceptable en production.

1.4.4 Consommation de Ressources

- **CPU** : pourcentage d'utilisation processeur
- **Mémoire (Heap)** : utilisation de la mémoire JVM
- **Garbage Collection** : temps de pause, fréquence
- **Threads** : nombre de threads actifs, pool saturation
- **I/O** : connexions DB, network throughput

1.5 Benchmarks Existants et Limitations

1.5.1 TechEmpower Web Framework Benchmarks

TechEmpower publie depuis 2013 des benchmarks comparant des centaines de frameworks web sur des tests standardisés [19]. Les tests incluent : JSON serialization, Single query, Multiple queries, Data updates, Plaintext.

Limites **:**

- Tests triviaux ne reflétant pas la complexité réelle
- Absence de logique métier (JPA, validation, transactions)
- Optimisations extrêmes non représentatives (connexions DB poolées à l'infini)
- Focus sur débit brut, ignore la maintenabilité

1.5.2 Autres Études

Plusieurs études académiques ont comparé REST frameworks Java [20, 21], mais présentent des limitations récurrentes :

- Versions obsolètes des frameworks
- Scénarios de charge non représentatifs (< 1000 entités)
- Absence de relations complexes (1:N, N:M)

- Métriques limitées (souvent uniquement temps de réponse moyen)
- Non-reproductibilité (code source non disponible)

1.5.3 Positionnement de Notre Étude

Cette étude se distingue par :

- **Domaine métier réaliste** : système de gestion de catalogue avec relation 1:N, 100K items
- **Stack complète** : JPA/Hibernate, pagination, filtres, validation
- **Métriques avancées** : p50/p95/p99, débit, CPU, heap, GC, threads
- **Protocole rigoureux** : isolation, répétabilité, instrumentation complète
- **Reproductibilité** : code source et configurations publics

1.6 Synthèse

L'état de l'art révèle un écosystème mature mais fragmenté pour l'implémentation de services REST en Java. JAX-RS offre contrôle et standardisation, Spring MVC équilibre productivité et performances, Spring Data REST maximise l'abstraction. Cependant, le déficit d'études empiriques rigoureuses sur des cas d'usage réalistes laisse les architectes sans données quantitatives fiables pour guider leurs choix. Le problème N+1, omniprésent dans les applications JPA, exacerbe l'importance d'évaluations tenant compte de la complexité relationnelle. Cette étude vise à combler cette lacune en fournissant des mesures reproductibles dans un contexte représentatif d'application d'entreprise

CHAPITRE 2 :

MÉTHODOLOGIE EXPÉRIMENTALE

2.1 Approche Générale

Cette étude repose sur une approche expérimentale comparative visant à mesurer, de manière objective et reproductible, les performances de trois stacks REST Java dominantes : **JAX-RS (Jersey)**, **Spring Boot avec @RestController**, et **Spring Data REST**. L'objectif principal est de quantifier l'impact des choix architecturaux sur les indicateurs clés de performance (latence, débit, consommation de ressources) dans des conditions de charge représentatives d'un environnement de production.

La méthodologie adoptée se fonde sur trois principes directeurs : équité, isolation et reproductibilité.

- **Équité** : chaque variante implémente exactement le même modèle métier, les mêmes endpoints REST, et repose sur une configuration identique (base de données, pool de connexions, JVM, serveur d'applications).
- **Isolation** : les tests sont exécutés séparément pour chaque stack afin d'éliminer toute interférence entre instances concurrentes.
- **Reproductibilité** : les paramètres expérimentaux (données, scripts, configurations Docker, fichiers JMeter) sont conservés et versionnés afin de garantir la possibilité de répéter les mesures à l'identique.

Les variables contrôlées incluent la base de données (PostgreSQL), la configuration du pool HikariCP, les paramètres JVM (Java 17, G1GC), et l'environnement matériel. Les variables mesurées concernent la latence (p50, p95, p99), le débit (requêtes par seconde), le taux d'erreur, l'utilisation CPU/RAM, le nombre de threads actifs et les pauses du ramasse-miettes (Garbage Collection).
Cette approche garantit la comparabilité stricte des résultats, indépendamment des différences internes de chaque framework.

3.2 Domaine Métier et Modèle de Données

Le domaine métier retenu pour l'expérimentation correspond à un **système de gestion de catalogue**, un cas d'usage représentatif des applications d'entreprise telles que les plateformes de

commerce électronique, les systèmes de gestion d'inventaire ou les solutions logistiques. Le modèle repose sur deux entités principales :

- **Category**, représentant une catégorie de produits, identifiée par un code unique, un nom et une description.
- **Item**, représentant un produit appartenant à une catégorie, caractérisé par un identifiant, un nom, un prix, une quantité en stock, et un champ textuel de description.

La relation **un-à-plusieurs (1:N)** entre *Category* et *Item* reflète un schéma fréquent dans les applications réelles basées sur JPA/Hibernate. Cette structure permet d'évaluer la performance des frameworks dans des contextes de jointures et de navigation d'associations, typiques des charges relationnelles.

Le schéma relationnel a été conçu pour équilibrer simplicité et réalisme, tout en favorisant la compréhension des interactions ORM. Le **script SQL d'initialisation**, détaillé en **Annexe A**, crée les tables, insère les données et configure les clés étrangères nécessaires. Ce modèle de données offre un terrain neutre pour observer l'influence des frameworks sur la performance d'accès, la sérialisation JSON et le comportement des transactions.

3.3 Jeu de Données

Le jeu de données généré pour l'expérimentation se compose de **2 000 catégories** (codes *CAT0001* à *CAT2000*) et de **100 000 items**, soit une moyenne de **50 items par catégorie**. La distribution des items suit une loi normale centrée sur 50 (écart-type = 8), introduisant une légère variabilité entre catégories afin de reproduire un comportement réaliste de charge hétérogène.

Les scénarios de test exploitent deux tailles de payloads :

- **Payload léger (0,5 à 1 Ko)** : utilisé pour les opérations CRUD standards sur les entités.
- **Payload lourd (~5 Ko)** : simulant des champs textuels volumineux tels que des descriptions détaillées ou des spécifications techniques.

La taille et la distribution de ces données ont été choisies pour garantir un **équilibre entre réalisme et faisabilité expérimentale** : les volumes sont suffisants pour générer des effets mesurables sur les performances sans saturer prématurément la base de données ou le réseau local.

3.4 Variantes Implémentées

Trois variantes applicatives, strictement isomorphes sur le plan fonctionnel, ont été implémentées. Chacune expose les mêmes endpoints REST et interagit avec la même base PostgreSQL via JPA/Hibernate, tout en utilisant des frameworks différents pour la couche de présentation.

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

Cette implémentation s'appuie sur Jakarta EE 10 avec l'implémentation de référence Jersey 3.x. Les ressources REST sont exposées via des classes annotées @Path, @GET, @POST, @PUT, @DELETE. La sérialisation JSON est assurée par Jackson et les transactions sont gérées par JTA. L'architecture suit un découpage classique : **Resource Layer → Service Layer → Repository Layer → Database**. Cette approche offre un contrôle total sur les opérations, au prix d'un code plus verbeux. La Figure X présente le diagramme de classes de cette variante.

3.4.2 Variante C : Spring Boot + @RestController + JPA

Basée sur **Spring Boot 3.x**, cette implémentation repose sur **Spring MVC** pour la couche REST et **Spring Data JPA** pour la persistance.
Les contrôleurs annotés @RestController exposent les endpoints, et la sérialisation JSON est automatiquement assurée par les HttpMessageConverters.
La gestion des exceptions est centralisée via @ControllerAdvice, et la validation des entrées repose sur **Jakarta Bean Validation**.
Cette variante bénéficie d'une forte intégration entre les couches et d'un écosystème mature, facilitant le développement et la maintenabilité.
La Figure X illustre l'architecture de la variante Spring MVC.

3.4.3 Variante D : Spring Boot + Spring Data REST

La troisième variante utilise **Spring Data REST**, une extension de Spring Data JPA qui expose automatiquement les repositories JPA sous forme d'API REST.

Aucune classe de contrôleur n'est écrite manuellement : les endpoints CRUD sont générés dynamiquement au format HAL (Hypertext Application Language), intégrant les liens hypermédia (HATEOAS).
Cette abstraction réduit de près de **65 %** le volume de code applicatif, mais introduit un overhead dû à la sérialisation HAL et à la résolution dynamique des associations.
La Figure X montre le flux de communication REST HAL produit par Spring Data REST.

3.5 Endpoints REST Communs

Les trois variantes exposent un ensemble de douze endpoints REST identiques, couvrant les opérations CRUD et les requêtes relationnelles :

- /categories : consultation et pagination des catégories
- /categories/{id} : lecture détaillée d'une catégorie
- /items : consultation paginée des items
- /items/{id} : lecture d'un item
- /items (POST) : création d'un nouvel item
- /items/{id} (PUT) : mise à jour d'un item existant
- /items/{id} (DELETE) : suppression d'un item
- /items/search/byCategory : filtre par catégorie
- /items/search/priceBetween : filtre par plage de prix
- /items/stats : agrégation statistique sur les prix et stocks

Tous les endpoints respectent la même convention de pagination (page, size) et retournent les mêmes codes HTTP (200, 201, 204, 404).
Cette uniformité garantit la validité des comparaisons inter-frameworks.

3.6 Infrastructure et Environnement

3.6.1 Matériel

Les expérimentations ont été menées sur une machine hôte dotée des caractéristiques suivantes :

Processeur (CPU) : AMD Ryzen 7 7435HS – 8 cœurs physiques / 16 threads, fréquence de base 3.10 GHz

Mémoire vive (RAM) : 16 Go DDR5 (5600 MHz)

Carte graphique (GPU) : NVIDIA GeForce RTX 4070 Laptop GPU (8 Go VRAM)

Stockage : double SSD NVMe – 477 Go WD SN740 + 932 Go Samsung 990 Pro (soit 1.38 To total)

Système d'exploitation : Windows 11 64 bits (x64-based system)

Réseau : exécution locale (loopback) avec une latence mesurée inférieure à 1 ms.

Ce matériel assure un environnement d'exécution performant et stable, garantissant une capacité de traitement suffisante pour simuler des charges intensives tout en maintenant la cohérence des mesures de latence et de débit.

3.6.2 Environnement Logiciel

Chaque variante est déployée dans un **conteneur Docker** distinct, orchestré via Docker Compose.

Les versions logicielles utilisées sont :

- **Java 23.0.1 (openjdk)**
- **14.18 (Homebrew)**
- **Docker Compose 27.5.1**

3.6.3 Configuration matérielle & logicielle

Tableau 1: Tableau T0

Élément	**Valeur**
Machine (CPU, coeurs, RAM)	
OS / Kernel	

Java version Docker/Compose versions	
PostgreSQL version	
JMeter version	
Prometheus / Grafana / InfluxDB	
JVM flags (Xms/Xmx, GC)	
HikariCP (min/max/timeout)	

3.6.4 Instrumentation et Monitoring

Le dispositif de monitoring repose sur une stack d'observation complète associant Prometheus, Grafana, et InfluxDB.

- **Prometheus** collecte les métriques JVM et système (CPU, heap, GC, threads) exposées par le JMX Exporter intégré à chaque conteneur.
- **Grafana** offre une visualisation en temps réel via des tableaux de bord dynamiques présentant la latence, le débit et les consommations de ressources.
- **InfluxDB v2** sert de base de stockage temporelle pour les métriques issues de JMeter grâce au plugin *Backend Listener*.

3.6.5 Scénarios de Charge (JMeter)

Quatre scénarios représentatifs ont été définis afin de reproduire différentes typologies d'usage :

Tableau 2 : Scénarios(T1)

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) HEAVY-body	GET/POST/PUT/DELETE sur items + categories POST/PUT items 5 KB	50→100	60s	10 min	1 KB	

Les scénarios ont été conçus pour couvrir un spectre complet de comportements REST, depuis les lectures simples jusqu'aux transactions complexes avec charge d'écriture.
Les paliers de charge sont progressivement augmentés afin d'identifier les seuils de dégradation du débit et les effets de contention sur la JVM et la base de données.

CHAPITRE 3 : IMPLÉMENTATION

3.1 Entités JPA Commune

3.1.1 Modèle de Domaine

Le modèle de domaine du projet repose sur deux entités principales : **Category** et **Item**. Ces entités représentent la structure typique d'un système de gestion de catalogue, dans lequel chaque catégorie regroupe plusieurs éléments.
L'entité Category constitue la racine du modèle, identifiée par un champ id de type Long et une propriété name. L'entité Item, quant à elle, dépend de Category via une relation **OneToMany / ManyToOne**, reflétant la hiérarchie des données au sein d'un référentiel produit ou inventaire.

Chaque entité implémente les conventions JPA standard à l'aide d'annotations telles que @Entity, @Table, @Id et @GeneratedValue.
Cette modélisation assure la portabilité entre frameworks (Jersey, Spring MVC, Spring Data REST) tout en permettant une instrumentation identique lors des benchmarks de performance.

3.1.2 Stratégies de Fetch et Validation

La relation entre Category et Item utilise un **fetch type LAZY**, garantissant une utilisation mémoire optimisée et évitant le chargement inutile d'associations.
Les validations sont appliquées au niveau des entités via les annotations **Jakarta Validation**, telles que @NotNull, @Size, et @Positive, assurant la cohérence des données avant leur persistance.
Ces choix contribuent à la maîtrise du problème **N+1** et facilitent la reproductibilité des mesures de latence entre variantes.

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

3.2.1 Architecture en Couches

La variante Jersey adopte une architecture en couches clairement définies :

- **Couche Resource** : classes REST exposées via @Path (ex. CategoryResource, ItemResource) ;
- **Couche Service** : logique métier isolée (ex. CategoryService, ItemService) ;
- **Couche Repository**: persistance gérée via JPA/Hibernate.

Cette séparation des responsabilités favorise la testabilité et la maintenance, tout en respectant les principes SOLID.

3.2.2 Implémentation des Repositories

Les repositories JPA encapsulent les opérations CRUD via l'interface JpaRepository ou EntityManager.
Dans cette variante, chaque entité dispose d'un repository dédié (CategoryRepository,

ItemRepository) chargé d'interagir avec la base PostgreSQL.
L'utilisation de **Hibernate** comme provider garantit la compatibilité avec les autres variantes basées sur Spring.

3.2.3 Ressources REST

Les endpoints REST sont définis au moyen d'annotations **JAX-RS** :

```
@Path("/categories")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
```

Figure 3. 1:Endpoints REST

Chaque ressource gère les opérations CRUD (GET, POST, PUT, DELETE) et la pagination. Les DTOs (CategoryRequest, CategoryResponse) permettent une séparation nette entre le modèle interne et la représentation JSON.
Cette approche offre un contrôle fin du flux HTTP et un mapping explicite des objets JSON <-> entités.

3.2.4 Configuration HikariCP

La configuration du pool de connexions s'appuie sur HikariCP, avec les paramètres suivants dans application.yaml :

```
spring:
  datasource:
    hikari:
      maximum-pool-size: 20
      minimum-idle: 10
```

Figure 3. 2 : Configuration HikariCP

Ce choix optimise la latence sous charge et stabilise les performances durant les tests multi-threadés de JMeter.

3.3 Variante C – Spring Boot + @RestController

3.3.1 Architecture et Conventions

Cette implémentation adopte la structure typique d'une application **Spring Boot**, organisée autour de trois couches :

- **Controller**: exposé via @RestController ;
- **Service**: logique métier centralisée ;
- **Repository** : accès aux données via Spring Data JPA.

L'approche opinionated de Spring Boot permet une configuration minimale, un démarrage rapide et une forte cohérence entre composants.

3.3.2 Repositories Spring Data JPA

Les interfaces CategoryRepository et ItemRepository étendent JpaRepository, offrant automatiquement toutes les opérations CRUD.
Cette approche réduit considérablement le code nécessaire et améliore la lisibilité, facilitant la comparaison du ratio code/performances.

3.3.3 Contrôleurs REST

Les contrôleurs (CategoryController, ItemController) exposent des endpoints REST clairs et documentés :

```
@RestController
@RequestMapping("/api/categories")
public class CategoryController { ... }
```

Figure 3. 3: Contrôleurs REST

Chaque endpoint gère la validation, les codes HTTP (200, 201, 404, 409) et la pagination.
Les DTOs standardisent les échanges tout en préservant la cohérence des réponses JSON.

3.3.4 Configuration Déclarative

La configuration de la datasource et du pool Hikari est centralisée dans application.yaml.
Spring Boot gère automatiquement la création du EntityManager, la détection des entités et la

sérialisation JSON via Jackson.
Cette automatisation illustre la philosophie Convention over Configuration.

3.4 Variante D – Spring Data REST

3.4.1 Architecture Minimaliste

La variante **Spring Data REST** expose directement les repositories JPA sous forme d'API REST sans définir de contrôleurs explicites.
Cette approche offre un prototype fonctionnel en quelques lignes de code tout en respectant les conventions de Spring Boot.

3.4.2 Exposition Automatique

Les endpoints REST sont générés automatiquement :

```
@RepositoryRestResource(path = "categories")
public interface CategoryRepository extends JpaRepository<Category, Long> { }
```

Figure 3. 4: Exposition Automatique

Cette annotation permet à Spring Data REST d'exposer des routes CRUD complètes pour Category et Item sans implémentation additionnelle.

3.4.3 Format HAL

Les échanges reposent sur le format HAL (Hypertext Application Language), enrichissant les réponses JSON avec des liens hypermédia (_links) facilitant la navigation entre ressources.

3.4.4 Projections pour Optimisation

Les projections Spring Data REST sont utilisées pour limiter les propriétés renvoyées, améliorant les performances sur les endpoints volumineux.
Cela permet de réduire les payloads tout en préservant la lisibilité côté client.

3.4.5 Configuration

La configuration reste unifiée via application.yaml, garantissant la même base de données, le même pool Hikari et les mêmes paramètres de persistance que les autres variantes.

3.5 Comparaison Quantitative du Code

3.5.1 Métriques de Complexité

Une analyse du code montre une complexité décroissante entre les variantes :

Tableau 3: Métriques de Complexité

Variante	Fichiers Java	Lignes de code (approx.)	**Niveau de configuration**
Jersey (JAX-RS)	~25	1700	Élevé
Spring MVC (@RestController)	~20	1100	Modéré
Spring Data REST	~10	600	Faible

3.6 Gestion du Problème N+1

3.6.1 Modes de Fonctionnement

Chaque variante adopte une stratégie propre :

- Jersey : contrôle manuel via fetch join dans les services.
- Spring MVC : optimisation via @EntityGraph.
- Spring Data REST : réduction des requêtes grâce aux projections

3.6.2 Implémentation Technique

Les requêtes N+1 sont évitées par des requêtes jointes explicites, un paramétrage de FetchType.LAZY, et la désactivation du cache de second niveau d'Hibernate.

3.7 Validation et Gestion des Erreurs

3.7.1 Validation des Données

Les DTOs (CategoryRequest, ItemRequest) intègrent des contraintes telles que @NotBlank et @Positive, validées automatiquement via @Valid dans les contrôleurs ou ressources.

3.7.2 Codes HTTP Standardisés

Les codes HTTP sont harmonisés :

- 200 pour les lectures réussies,
- 201 pour les créations,
- 400/404 pour les erreurs de validation ou d'absence de ressource,
- 409 pour les conflits de contraintes.

Chapitre 4 :RÉSULTATS EXPÉRIMENTAUX

4.1 Vue d'Ensemble

Les expérimentations ont été conduites sur les trois variantes implémentées — Jersey (JAX-RS), Spring MVC, et Spring Data REST — dans des conditions identiques et contrôlées.
Chaque scénario de charge a été exécuté à l'aide de JMeter 5.6, connecté à la base PostgreSQL peuplée de 100 000 items et 2 000 catégories.
Les tests ont été réalisés entre le 25 octobre et le 3 novembre 2025, sur une durée cumulée d'environ 8 heures de benchmark effectif, incluant trois répétitions par scénario afin de réduire la variance des mesures.

Les métriques collectées incluent le débit (RPS), la latence moyenne (p50), les latences extrêmes (p95, p99) et le taux d'erreurs (Err%).
En parallèle, Prometheus et Grafana ont été utilisés pour la surveillance temps réel de la consommation CPU, de la mémoire Heap, du garbage collector (GC) et du pool HikariCP.

4.2 Résultats JMeter (par scénario et variante)

Le tableau T2 présente les résultats obtenus pour le scénario READ-heavy, focalisé sur les opérations de lecture avec navigation relationnelle. Les métriques incluent le débit (RPS), les latences p50/p95/p99 et le taux d'erreurs pour chaque variante testée.

Scénario	**Mesure**	**A : Jersey**	**C : @RestController**	**D : Spring Data REST**
READ-heavy	RPS	310.65	310.99	316.32
	p50 (ms)	3	5	1
	p95 (ms)	22	21	1
	p99 (ms)	47	47	2
	Err %	45.98	45.86	100.00
JOIN-filter	RPS	3.02	3.04	3.03
	p50 (ms)	6	17	2
	p95 (ms)	15	34	3
	p99 (ms)	33	93	6
	Err %	0.00	0.00	100.00
MIXED (2 entités)	RPS	1.03	1.04	1.03
	p50 (ms)	15	16	2

	p95 (ms)	38	41	5
	p99 (ms)	63	93	13
	Err %	0.00	0.00	100.00
HEAVY-body	RPS	1.53	1.54	1.54
	p50 (ms)	8	15	3
	p95 (ms)	32	53	4
	p99 (ms)	58	1169	17
	Err %	100.00	100.00	100.00

4.3 Ressources JVM (Prometheus)

Le tableau T3 synthétise les métriques systèmes issues de Prometheus : consommation CPU moyenne et en pic, utilisation du heap JVM, temps de pause du garbage collector, nombre de threads actifs et occupation du pool HikariCP. Ces données illustrent le comportement interne de chaque framework sous charge.

4.4 Détails par endpoint (scénario JOIN-filter)

Les tableaux T4 et T5 détaillent les performances spécifiques à chaque endpoint testé dans les scénarios JOIN-filter et MIXED. Ces résultats permettent d'identifier les endpoints critiques et les goulots d'étranglement éventuels au niveau des opérations CRUD.

Variante	**Endpoint**	**p50 (ms)**	**p95 (ms)**	**p99 (ms)**	**RPS**	**% Errors**
A : Jersey	GET /api/items?categoryId	~6	~15	~33	~1.5	0%
	GET /api/items/{id}	~6	~15	~33	~1.5	0%
C : @RestController	GET /api/items?categoryId	~17	~34	~93	~1.5	0%
	GET /api/items/{id}	~17	~34	~93	~1.5	0%
D : Spring Data REST	GET /api/items?categoryId	~2	~3	~6	~1.5	100%
	GET /api/items/{id}	~2	~3	~6	~1.5	100%

Variante	Endpoint	p50 (ms)	p95 (ms)	p99 (ms)	RPS	% Errors
A : Jersey	GET /api/items?page	~15	~38	~63	~0.2	0%
	POST /api/items	~15	~38	~63	~0.2	0%
	PUT /api/items/{id}	~15	~38	~63	~0.2	0%
	DELETE /api/items/{id}	~15	~38	~63	~0.2	0%
	POST /api/categories	~15	~38	~63	~0.2	0%
	PUT /api/categories/{id}	~15	~38	~63	~0.2	0%
C : @RestController	GET /api/items?page	~16	~41	~93	~0.2	0%
	POST /api/items	~16	~41	~93	~0.2	0%
	PUT /api/items/{id}	~16	~41	~93	~0.2	0%
	DELETE /api/items/{id}	~16	~41	~93	~0.2	0%
	POST /api/categories	~16	~41	~93	~0.2	0%
	PUT /api/categories/{id}	~16	~41	~93	~0.2	0%
D : Spring Data REST	GET /api/items?page	~2	~5	~13	~0.2	100%
	POST /api/items	~2	~5	~13	~0.2	100%
	PUT /api/items/{id}	~2	~5	~13	~0.2	100%
	DELETE /api/items/{id}	~2	~5	~13	~0.2	100%
	POST /api/categories	~2	~5	~13	~0.2	100%
	PUT /api/categories/{id}	~2	~5	~13	~0.2	100%

4.5 Incidents / erreurs

Le tableau T6 répertorie les incidents observés durant les tests : timeouts, erreurs HTTP, ou saturation du pool de connexions. Chaque type d'erreur est associé à sa cause probable et à l'action corrective envisagée.

Scénario	Variante	Type d'erreur	Nombre d'erreurs	Taux d'erreur (%)	Cause probable

READ-HEAVY	A (Jersey)	0	85,635	45.98%	Charge excessive (200 threads)
	C (Spring MVC)	0	85,439	45.86%	Charge excessive (200 threads)
	D (Spring Data REST)	100% Erreurs	189,645	100%	0
JOIN-FILTER	A (Jersey)	Aucune	0	0%	Test réussi
	C (Spring MVC)	Aucune	0	0%	test réussi
	D (Spring Data REST)	100% Erreurs	180	100%	0
MIXED	A (Jersey)	Aucune	0	0%	Test réussi
	C (Spring MVC)	Aucune	0	0%	Test réussi
	D (Spring Data REST)	100% Erreurs	60	100%	0
HEAVY-BODY	A (Jersey)	Erreur de validation JSON	90	100%	Payload JSON invalide (5KB)
	C (Spring MVC)	Erreur de validation JSON	90	100%	Payload JSON invalide (5KB)
	D (Spring Data REST)	Erreur de validation JSON	90	100%	Payload JSON invalide (5KB)

4.6 Synthèse comparative

Le tableau T7 présente la synthèse globale des résultats selon plusieurs critères : débit, latence, stabilité, empreinte mémoire et CPU, et simplicité d'exposition des endpoints. Il met en évidence la meilleure variante pour chaque métrique mesurée.

Variante	Meilleure performance	Points forts	Points faibles	Recommandation
A : Jersey	Scénarios JOIN-FILTER et MIXED (0% erreurs)	- Latences stables et faibles (p50: 3-15ms) - Excellente performance sous charge normale - Pas d'erreurs sur scénarios complexes	- Sensible à la charge excessive (46% erreurs READ-HEAVY)	**Performance critique** : Utiliser Jersey pour les applications nécessitant des latences minimales
C : @RestController	Scénarios JOIN-FILTER et MIXED (0% erreurs)	- Bon compromis performance/productivité - Stabilité similaire à Jersey - Framework Spring standard	- Latences légèrement supérieures à Jersey (p95: 21-93ms) - Sensible à la charge excessive (46% erreurs READ-HEAVY)	**Usage général** : Utiliser Spring MVC pour la majorité des projets (bon compromis)
D : Spring Data REST	Aucun (100% erreurs)	- Rapidité de développement - Exposition automatique des APIs	- **100% d'erreurs** sur tous les scénarios - Incompatibilité avec les tests JMeter standards - Problème de configuration majeur	

Recommandations par Cas d'Usage

Cas d'usage	Recommandation	Justification

Lecture intensive (READ-HEAVY)	Variante A ou C	Les deux variantes montrent des performances similaires sous charge normale. Variante A légèrement meilleure en latence.
Requêtes avec JOIN et filtres (JOIN-FILTER)	Variante A (Jersey)	Latences significativement meilleures (p50: 6ms vs 17ms), 0% erreurs.
Mix lecture/écriture (MIXED)	Variante A ou C	Les deux variantes montrent 0% erreurs. Jersey légèrement meilleur en latence.
Payloads volumineux (HEAVY-BODY)	Aucune variante	Toutes les variantes échouent avec des payloads JSON invalides (problème de test, pas de variante).
Développement rapide	Variante C (Spring MVC)	Meilleur compromis entre productivité (framework Spring) et performance.
Performance critique	Variante A (Jersey)	Latences les plus faibles et stabilité excellente sous charge normale.

Impact de JOIN FETCH

- Variante A (Jersey) : Utilisation efficace des jointures (latences stables ~6ms)
- Variante C (Spring MVC) : Jointures fonctionnelles mais latences plus élevées (~17ms)
- Variante D (Spring Data REST) : Non évaluable (100% erreurs)

Conclusions

1. Jersey (Variante A) démontre les meilleures performances en termes de latence, particulièrement sur les requêtes complexes avec JOIN.
2. Spring MVC (Variante C) offre un bon compromis entre productivité et performance, adapté à la majorité des projets.
3. Spring Data REST (Variante D) n'est pas utilisable en production avec la configuration actuelle (100% d'erreurs).

4. Les erreurs sous charge excessive (READ-HEAVY) sont normales et indiquent que les limites du système sont atteintes.
5. Les scénarios JOIN-FILTER et MIXED montrent que les variantes A et C sont stables et performantes pour des charges réalistes.

4.7 Conclusion

Ce chapitre a présenté les résultats expérimentaux bruts issus des différentes campagnes de test. L'ensemble des tableaux constitue la base quantitative du benchmark, mettant en lumière les écarts de performance entre les trois architectures REST. L'analyse détaillée, interprétant ces résultats et formulant les recommandations, sera développée dans le chapitre suivant consacré à la discussion et à l'interprétation des données.