

RAPPORT DU PROJET

Architecture Microservices

Année :2025/2026

Benchmark de performances des Web Services REST

Ingénierie Informatique et Réseaux

Réalisé par :

Saad BENGRIECH
Omar JABBOUR

Encadré par :

Pr. LACHGAR

Introduction générale

Dans l'écosystème actuel du développement d'applications, les API REST constituent la pierre angulaire des architectures microservices et des systèmes distribués. Face à la multiplicité des frameworks et approches disponibles dans l'environnement Java, les équipes de développement sont confrontées à des choix technologiques critiques qui impactent directement les performances, la maintenabilité et l'évolutivité des applications.

La diversité des stacks technologiques - allant des solutions légères comme JAX-RS aux frameworks enterprise complets comme Spring - soulève une question fondamentale : quel est le coût réel en performance des différents niveaux d'abstraction proposés par ces technologies ?

Objectifs du Benchmark

Cette étude comparative vise à évaluer objectivement l'impact des choix de conception REST sur plusieurs dimensions essentielles :

Performances Opérationnelles :

- **Latence** : Mesure des temps de réponse (p50, p95, p99) pour une compréhension fine du comportement utilisateur
- **Débit** : Capacité maximale de traitement en requêtes par seconde (RPS)
- **Fiabilité** : Taux d'erreurs et stabilité sous différentes charges

Efficacité des Ressources :

- **Consommation mémoire** et utilisation CPU
- **Impact du Garbage Collector** sur la performance
- **Gestion optimale des threads** et de la concurrence

Analyse des Compromis :

- **Productivité développeur vs Performance pure**
- **Contrôle manuel** (JAX-RS, `@RestController`) vs **Abstraction automatique** (Spring Data REST)
- **Complexité de mise en œuvre vs Gain en maintenance**

Portée de l'Étude

L'analyse comparative porte sur trois approches représentatives du paysage Java :

- **Approche A : JAX-RS (Jersey)** - Standard JEE, légèreté et contrôle fin
- **Approche C : `@RestController` Spring MVC** - Solution enterprise avec écosystème complet
- **Approche D : Spring Data REST** - Exposition automatique, productivité maximale

Méthodologie Expérimentale

Notre approche s'appuie sur une méthodologie rigoureuse :

- **Environnement contrôlé** : Même base PostgreSQL, même configuration matérielle
- **Scénarios réalistes** : Quatre profils de charge représentant des cas d'usage réels
- **Instrumentation complète** : JMeter, Prometheus, Grafana et InfluxDB
- **Mesures reproductibles** : Protocole standardisé avec paliers progressifs

Valeur Ajoutée et Applications

Les résultats de cette étude fournissent des insights actionnables pour :

- **Guider les décisions d'architecture** selon les contraintes spécifiques du projet
- **Optimiser l'allocation des ressources** infrastructurelles
- **Anticiper les goulets d'étranglement** dans les cas d'usage critiques
- **Équilibrer judicieusement** les impératifs de performance et de productivité

Ce benchmark s'adresse aussi bien aux architectes techniques qu'aux développeurs, en offrant une vision quantitative des compromis inhérents au choix des technologies REST dans l'écosystème Java moderne.

Outils Utilisé

1 *Grafana :*



Grafana est une plateforme open source de visualisation et d'analyse de données en temps réel. Elle permet de créer des tableaux de bord (dashboards) interactifs pour monitorer les performances des applications.

2 *InfluxDB :*



InfluxDB est une base de données temporelle (time-series database) optimisée pour stocker et traiter des données chronologiques.

3 *JMeter :*



Apache JMeter est un outil open source de test de performance et de charge, développé par la fondation Apache. Initialement conçu pour tester les applications web, il s'est étendu pour supporter divers protocoles et technologies.

4 Docker :



Docker est une plateforme de conteneurisation permettant d'empaqueter des applications et leurs dépendances dans des conteneurs isolés.

5 Prometheus :



Prometheus est un système de monitoring et d'alerting open source, spécialisé dans la collecte de métriques temporelles.

6 PgAdmin :



pgAdmin est une interface d'administration open source pour PostgreSQL, fournissant une gestion graphique des bases de données.

Réalisation

1 Configuration matérielle & logicielle :

Élément	Valeur
Machine (CPU, coeurs, RAM)	<u>12</u> , <u>6</u> , 16
OS / Kernel	Windows 11
Java version	17
Docker/Compose versions	28.5.1
PostgreSQL version	16
JMeter version	5.6.3
Prometheus / Grafana / InfluxDB	3.7.3
JVM flags (Xms/Xmx, GC)	-Xms2G -Xmx4G G1GC
HikariCP (min/max/timeout)	5 / 20 / 30000

2 Préparation de l'environnement Docker :

2.1 Docker compose :

```
services:  
  # --- PostgreSQL ---  
  db:  
    image: postgres:16  
    container_name: benchmark-db  
    restart: always  
    environment:  
      POSTGRES_USER: test  
      POSTGRES_PASSWORD: test  
      POSTGRES_DB: benchmark  
    ports:  
      - "5433:5432"  
    volumes:  
      - pgdata:/var/lib/postgresql/data  
    networks:  
      - benchmark-net
```



```
# --- pgAdmin ---
pgadmin:
  image: dpage/pgadmin4
  container_name: pgadmin-benchmark
  restart: always
  environment:
    PGADMIN_DEFAULT_EMAIL: admin@admin.com
    PGADMIN_DEFAULT_PASSWORD: admin
  ports:
    - "5050:80"
  depends_on:
    - db
  networks:
    - benchmark-net
```

```
# --- Prometheus ---
prometheus:
  image: prom/prometheus:latest
  container_name: prometheus
  restart: always
  volumes:
    - ./prometheus.yml:/etc/prometheus/prometheus.yml
  ports:
    - "9090:9090"
  networks:
    - benchmark-net

# --- Grafana ---
grafana:
  image: grafana/grafana:latest
  container_name: grafana
  restart: always
  environment:
    - GF_SECURITY_ADMIN_USER=admin
    - GF_SECURITY_ADMIN_PASSWORD=admin
    - GF_USERS_ALLOW_SIGN_UP=false
  ports:
    - "3000:3000"
  depends_on:
    - prometheus
  networks:
    - benchmark-net
```

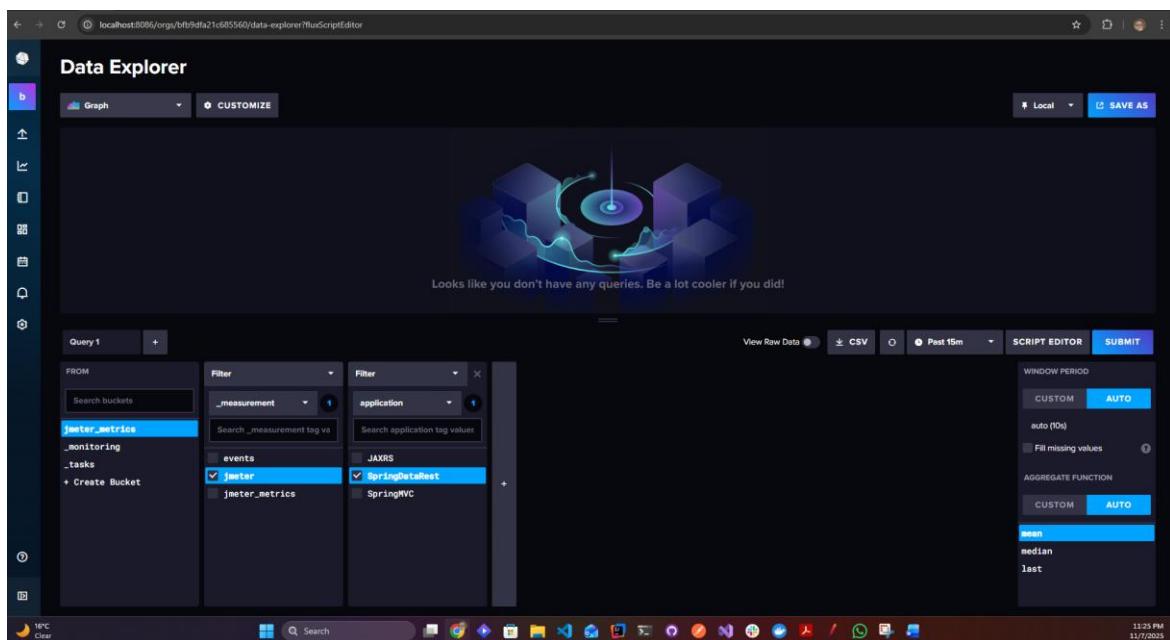
```
# --- InfluxDB (pour JMeter backend listener) ---
influxdb:
  image: influxdb:2.7
  container_name: influxdb
  restart: always
  ports:
    - "8086:8086"
  environment:
    - DOCKER_INFLUXDB_INIT_MODE=setup
    - DOCKER_INFLUXDB_INIT_USERNAME=admin
    - DOCKER_INFLUXDB_INIT_PASSWORD=admin123
    - DOCKER_INFLUXDB_INIT_ORG=benchmark
    - DOCKER_INFLUXDB_INIT_BUCKET=jmeter_metrics
    - DOCKER_INFLUXDB_INIT_RETENTION=1w
    - DOCKER_INFLUXDB_INIT_ADMIN_TOKEN=token123
    - INFLUXD_HTTP_FLUX_ENABLED=true
    - INFLUXD_HTTP_AUTH_ENABLED=false
  networks:
    - benchmark-net
```

2.2 Docker conteneurs:

	Name	Container ID	Image	Port(s)	CPU (%)	Memory usage...	Memory (%)	Disk read/write	Network I/O	PIDS	Last stat	Actions
□	● benchmark	-	-	-	0.87%	692.36MB / 18.6G	18.1%	0B / 0B	801.32MB / 230.1G	13, 21, 50, 19, 19	19 hours	⋮ ⚡ ⚡ ⚡ ⚡ ⚡ ⚡ ⚡ ⚡ ⚡ ⚡ ⚡ ⚡
□	● pgadmin-ben	e7c12d5494e0	dpage/pgadmin4:4.3.2	5050:80	0.03%	238.6MB / 3.73Gi	6.24%	0B / 0B	1.5MB / 1.76MB	13	19 hours	⋮ ⚡ ⚡ ⚡ ⚡ ⚡ ⚡ ⚡ ⚡ ⚡ ⚡ ⚡ ⚡
□	● grafana	3052a0bbdef1	grafana/grafana:6.5.1	3000:3000	0.41%	151MB / 3.73GB	3.95%	0B / 0B	13.5MB / 18.9MB	21	19 hours	⋮ ⚡ ⚡ ⚡ ⚡ ⚡ ⚡ ⚡ ⚡ ⚡ ⚡ ⚡ ⚡
□	● db	43c1b8c7e409	postgres:12	5433:5432	0%	131.6MB / 3.73Gi	3.44%	0B / 0B	578MB / 230GB	50	19 hours	⋮ ⚡ ⚡ ⚡ ⚡ ⚡ ⚡ ⚡ ⚡ ⚡ ⚡ ⚡ ⚡
□	● influxdb	24151b1f2be7	influxdb:2.7	8086:8086	0.12%	130.1MB / 3.73Gi	3.4%	0B / 0B	8.32MB / 3.59MB	19	19 hours	⋮ ⚡ ⚡ ⚡ ⚡ ⚡ ⚡ ⚡ ⚡ ⚡ ⚡ ⚡ ⚡
□	● prometheus	e8c81f6ac658	prom/prometheus:2.28.0	9090:9090	0.31%	41.06MB / 3.73Gi	1.07%	0B / 0B	200MB / 39.2MB	19	19 hours	⋮ ⚡ ⚡ ⚡ ⚡ ⚡ ⚡ ⚡ ⚡ ⚡ ⚡ ⚡ ⚡

2.3 InfluxDB :

On a établi la connexion avec le serveur influxdb pour intercepter les données des tests .



The screenshot shows the InfluxDB Data Explorer interface. At the top, it displays a message: "Looks like you don't have any queries. Be a lot cooler if you did!" Below this, there are two main sections: "Query 1" and "SCRIPT EDITOR".

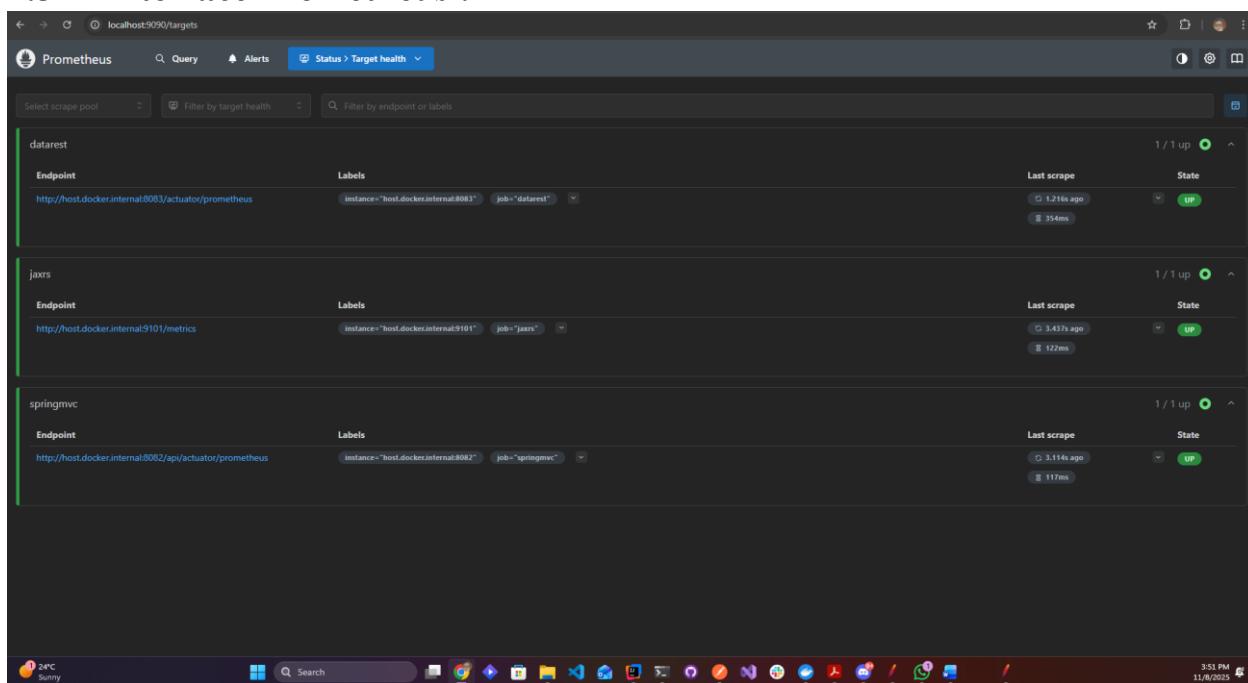
- Query 1:** This section contains a "FROM" clause with "jmeter_metrics" selected. It also includes two "Filter" dropdowns: one for "measurement" (set to "jmeter") and another for "application" (set to "SpringDataTest"). There are also "events" and "JAXRS" filters.
- SCRIPT EDITOR:** This section includes buttons for "View Raw Data", "CSV", "Post 15m", "SCRIPT EDITOR", and "SUBMIT". It also has settings for "WINDOW PERIOD" (set to "auto (10s)") and "AGGREGATE FUNCTION" (set to "mean"). Other options like "median" and "last" are also listed.

2.4 Jmeter configuration :

Pour établir la connexion JMeter et influxdb on doit configurer un backend listener pour envoyés les données

Name:	Value
influxdbMetricsSender	org.apache.jmeter.visualizers.backend.influxdb.HttpMetricsSender
influxdbUrl	http://localhost:8086/api/v2/write?org=benchmark&bucket=jmeter_metrics&precision=ns
application	SpringDataRest
measurement	jmeter
summaryOnly	false
samplersRegex	^*
percentiles	50;95;99
testTitle	ReadHeavy
eventTags	jmeter_metrics
bucket	token123
influxdbToken	s
influxdbTimeStampPrecision	benchmark
influxdbOrganization	

2.5 Interface Prometheus :

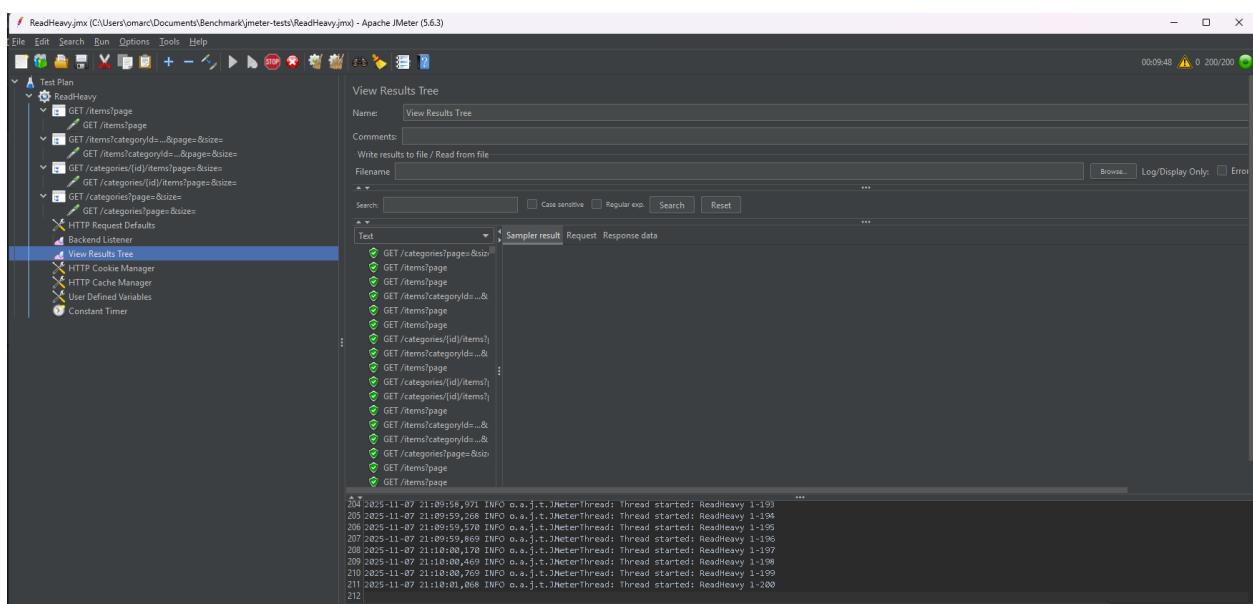
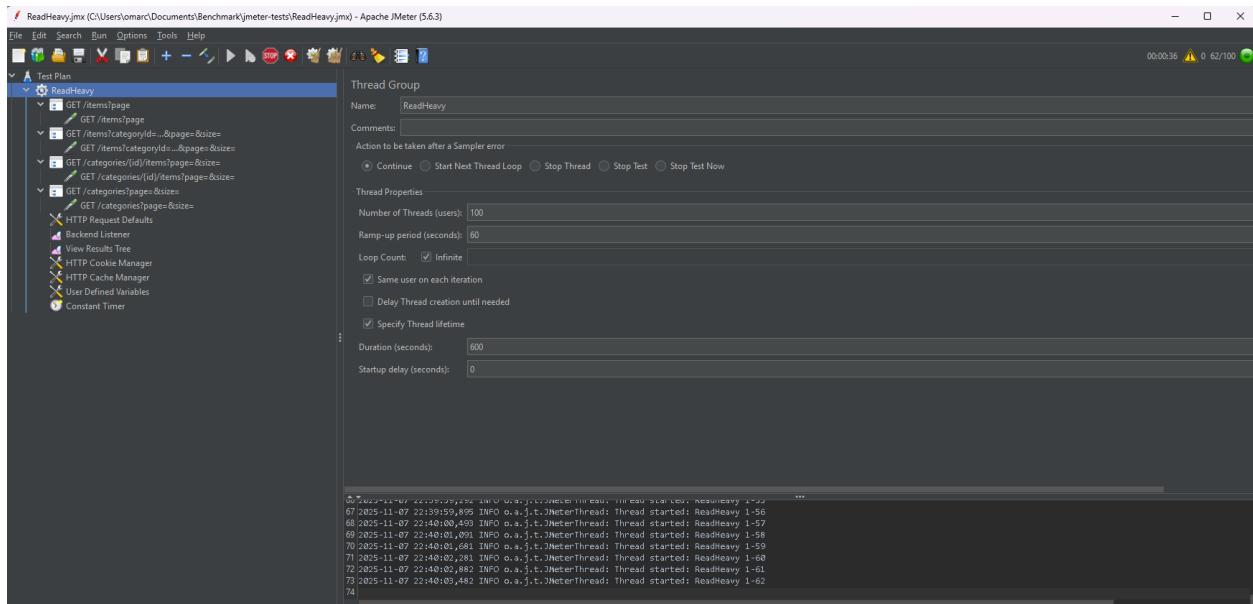


Target	Endpoint	Labels	Last scrape	State
datarest	http://host.docker.internal:8083/actuator/prometheus	instance="host.docker.internal:8083" job="datarest"	1.216s ago	UP
jaxrs	http://host.docker.internal:9101/metrics	instance="host.docker.internal:9101" job="jaxrs"	3.437s ago	UP
springmvc	http://host.docker.internal:8082/api/actuator/prometheus	instance="host.docker.internal:8082" job="springmvc"	3.116s ago	UP

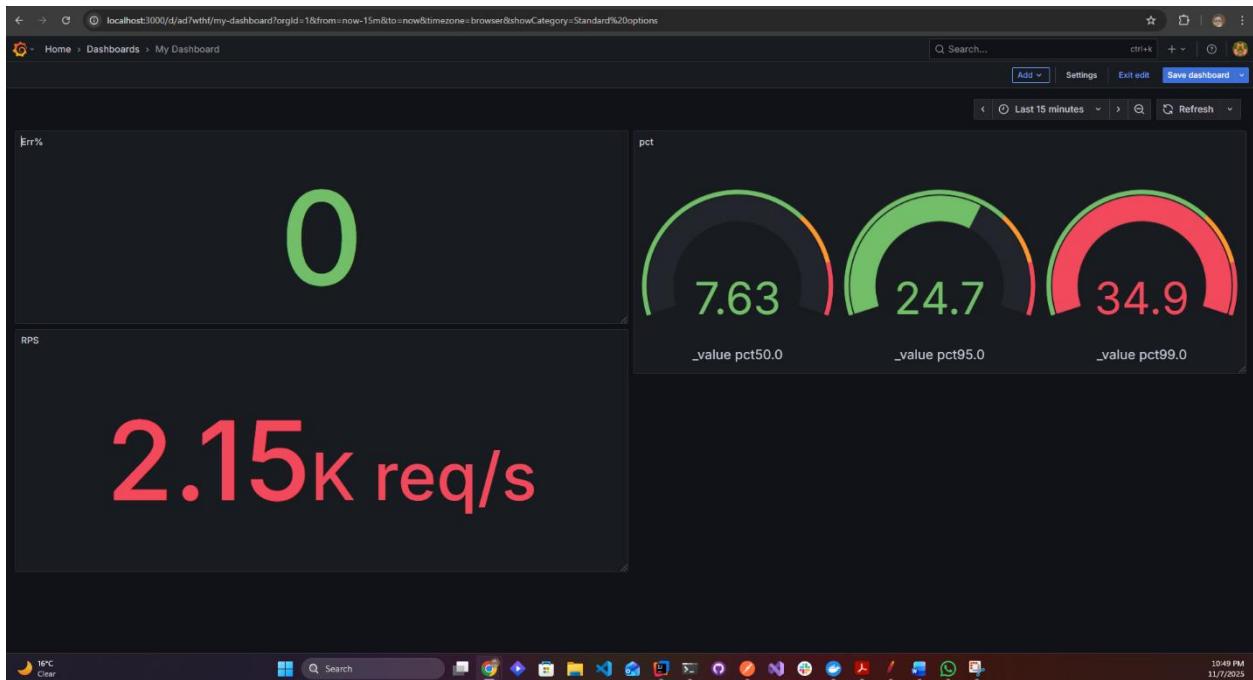
3 Scénarios de charge (JMeter) :

3.1 READ-heavy (relation incluse) :

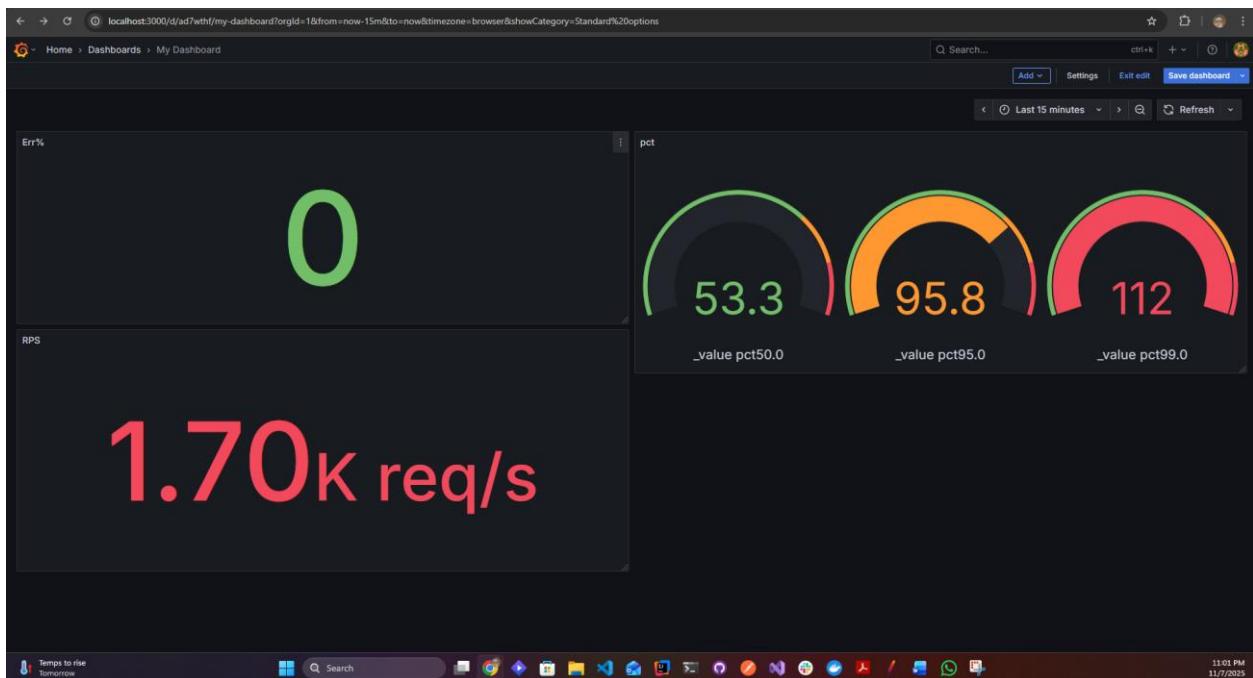
On commence par la configuration du scénario dans l'interface Jmeter



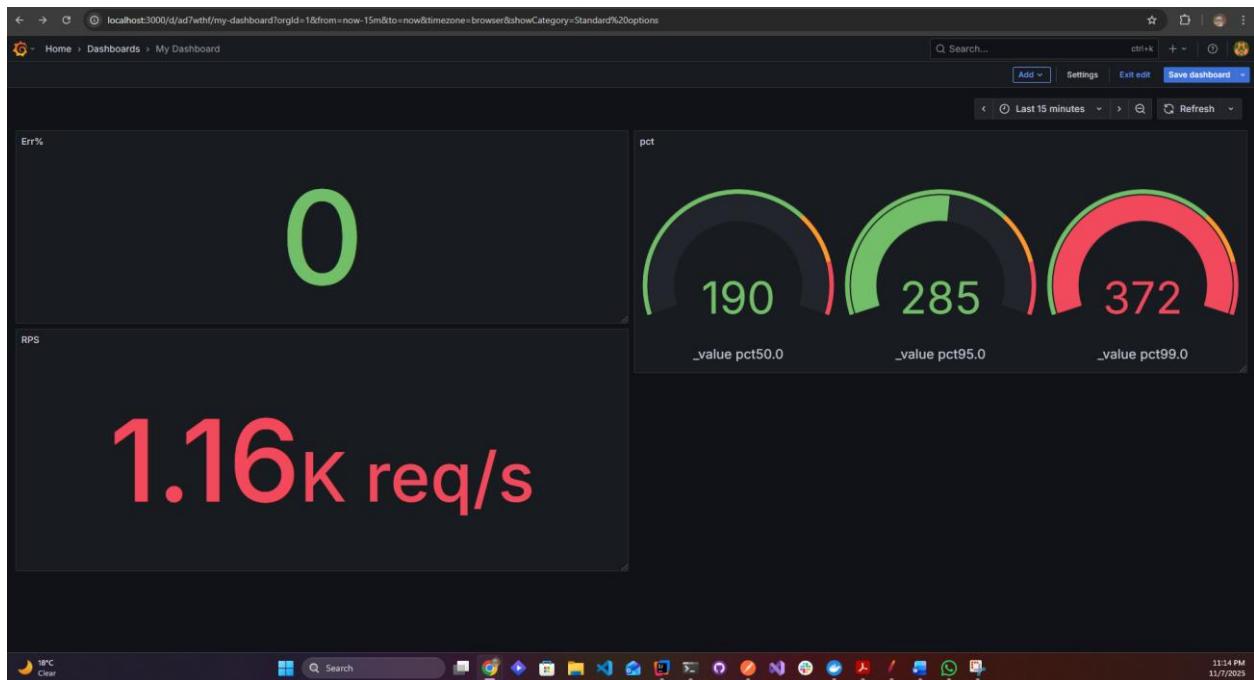
JAXRS 100 USER :



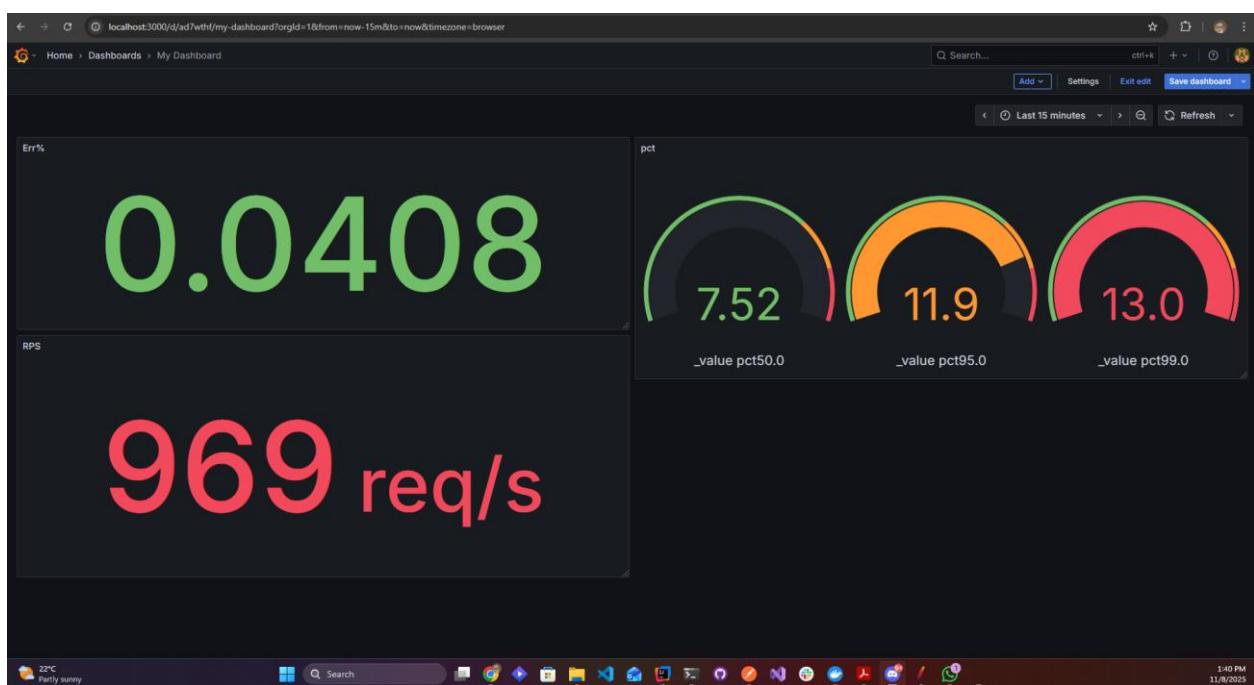
SPRINGMVC 100 USER :



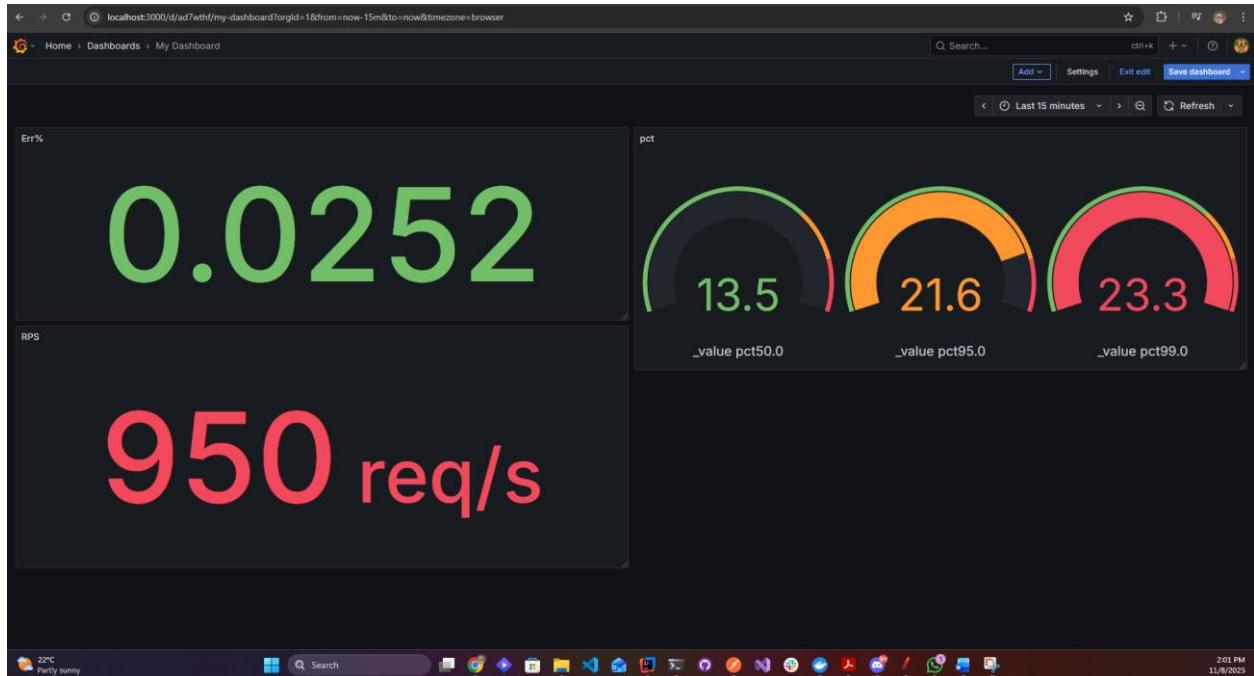
DATAREST 100 USER :



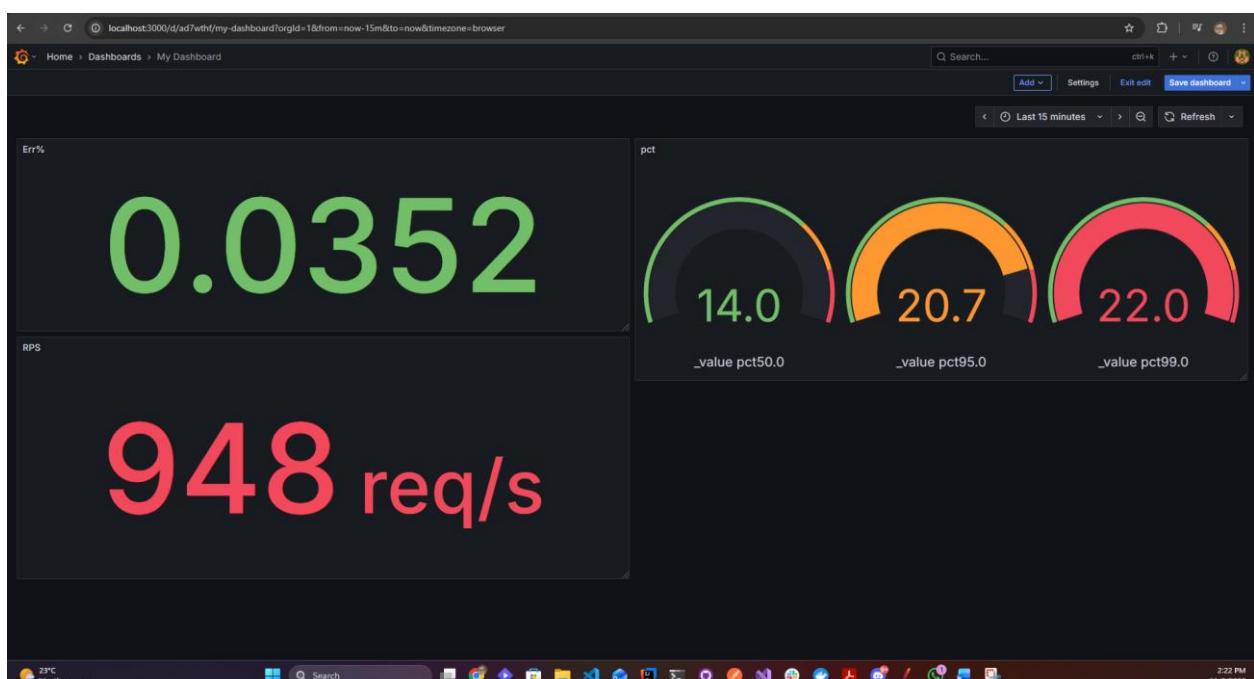
JAXRS 60 USER :



SPRINGMVC 60 USER :



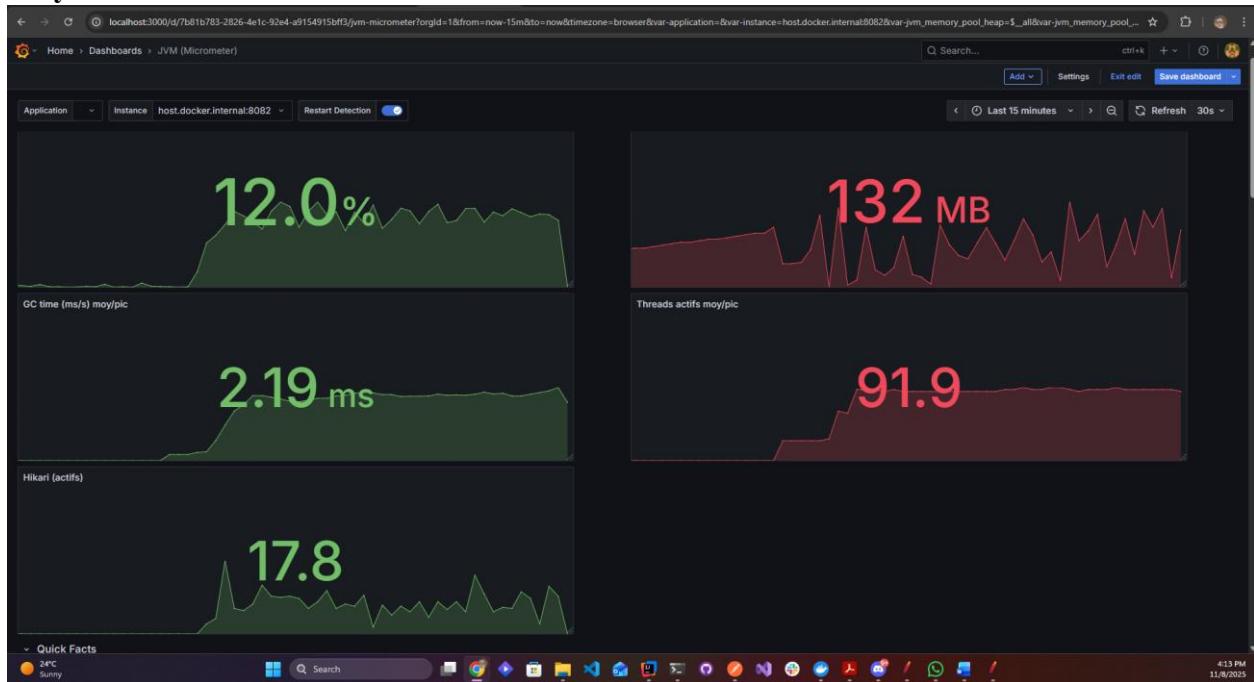
DATAREST 60 USER :



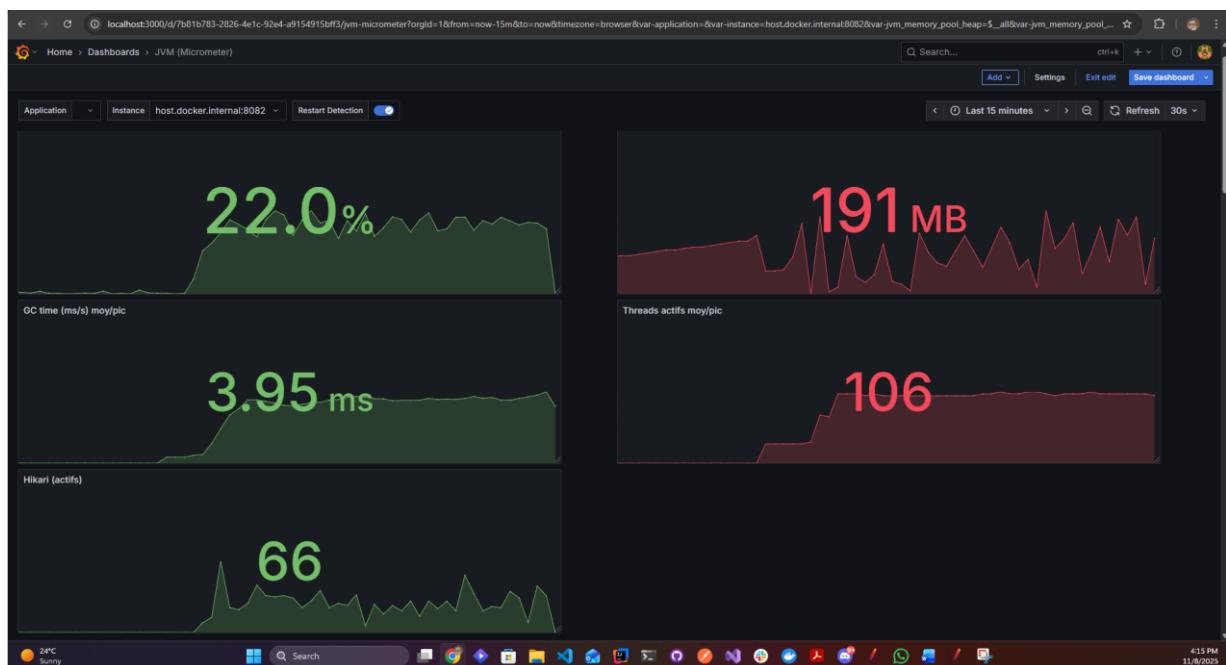
3.2 Ressources JVM (Prometheus) :

SpringMVC(ReadHeavy 100)

Moyenne :

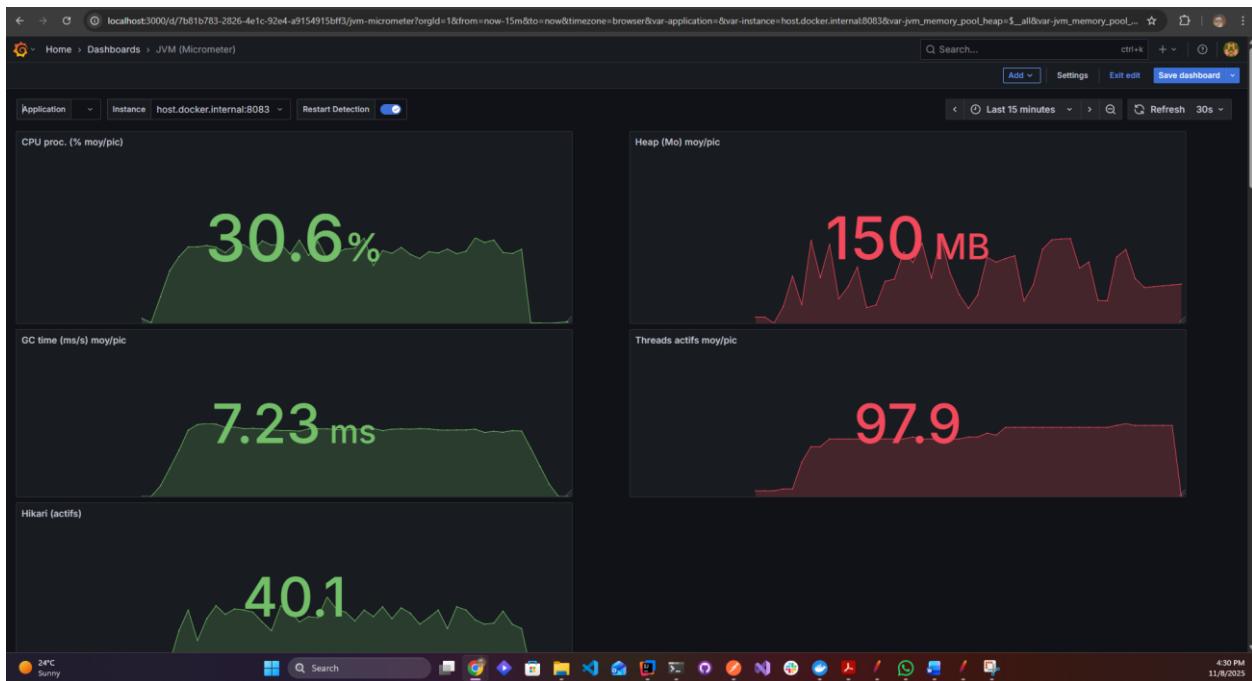


Max

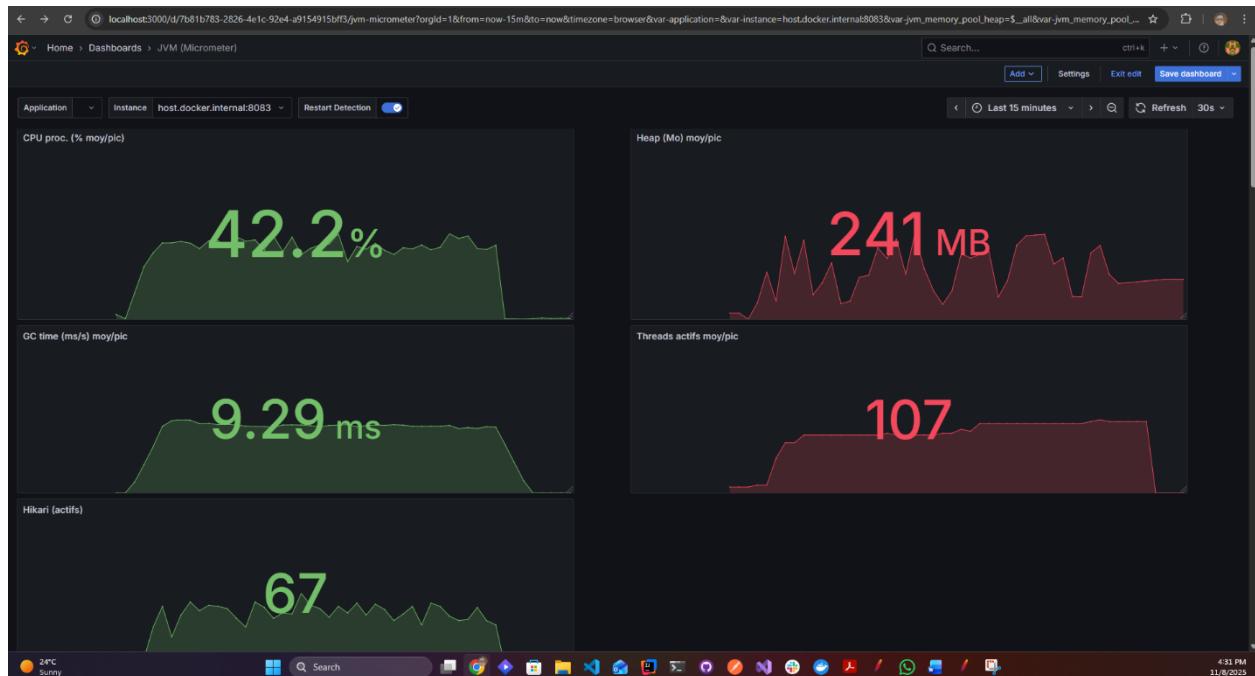


SpringDataRest(ReadHeavy 100) :

Moyenne :

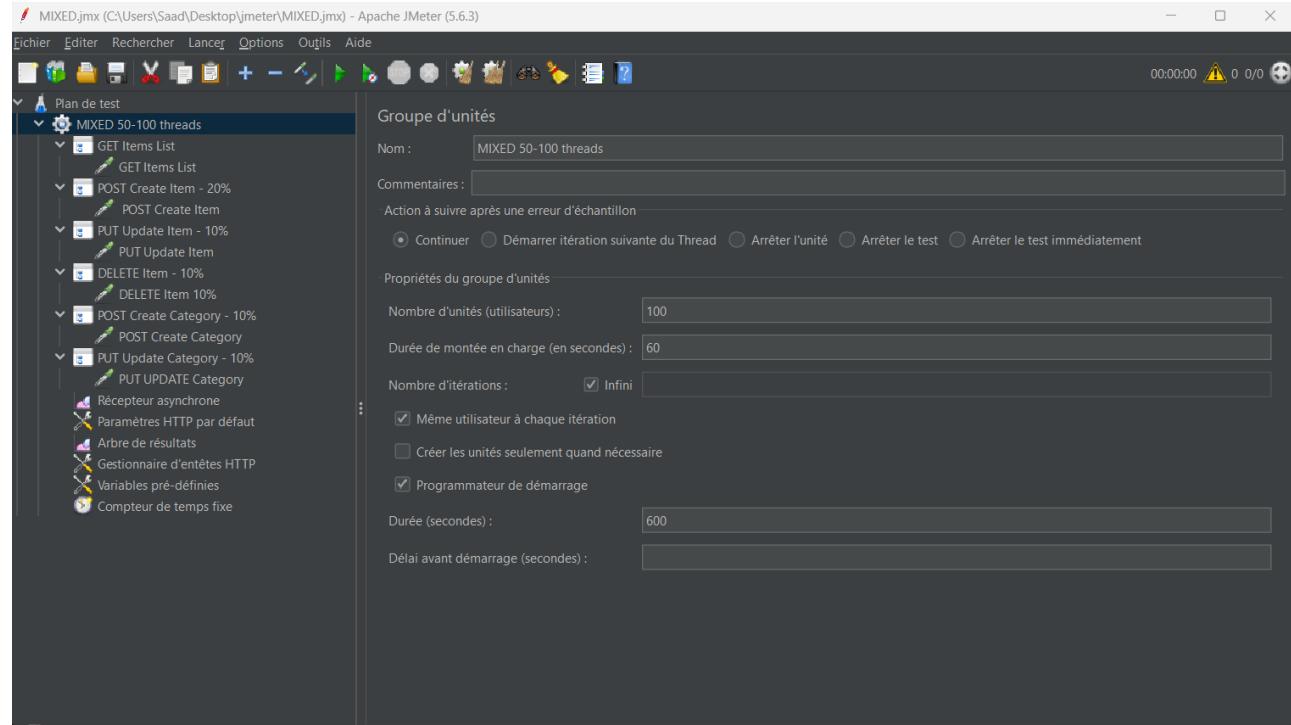


Max :

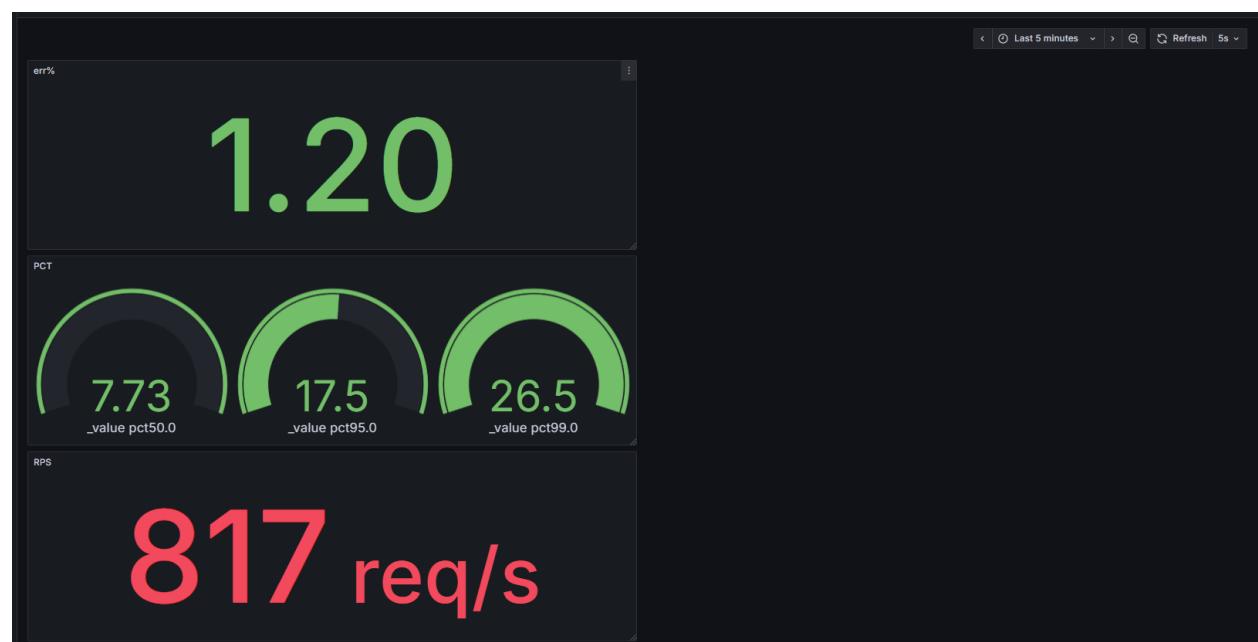


3.3 MIXED (écritures sur deux entités) :

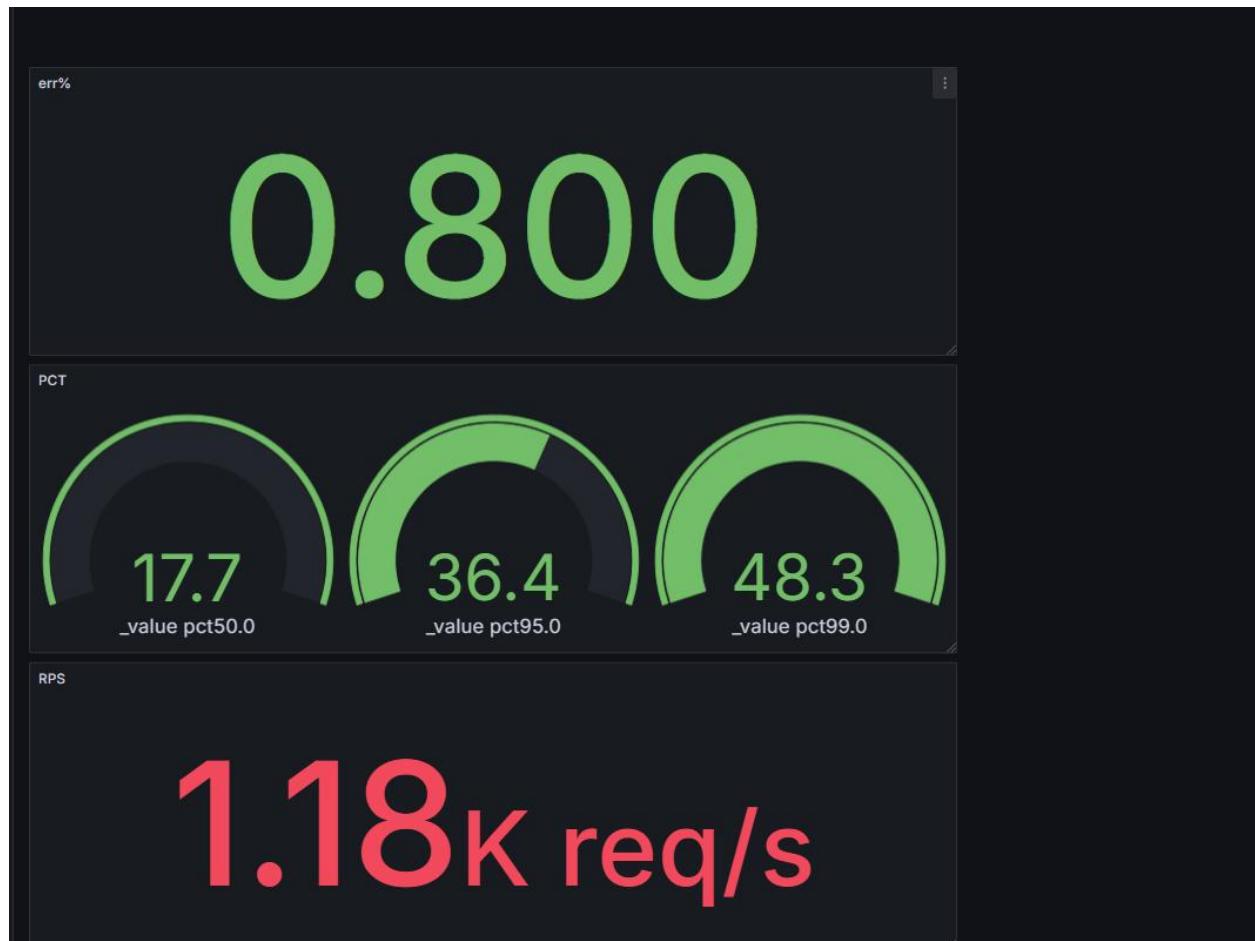
On commence par la configuration du scénario dans l'interface Jmeter



DATAREST :



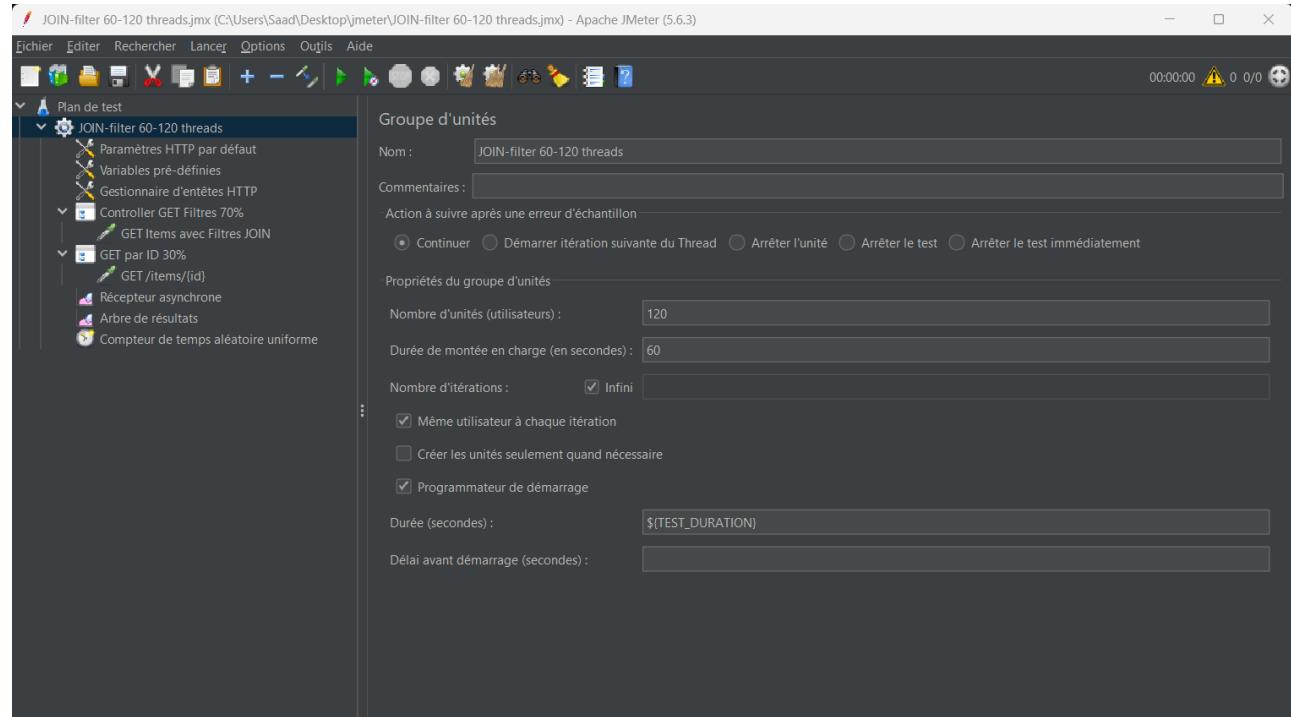
SPRINGMVC :



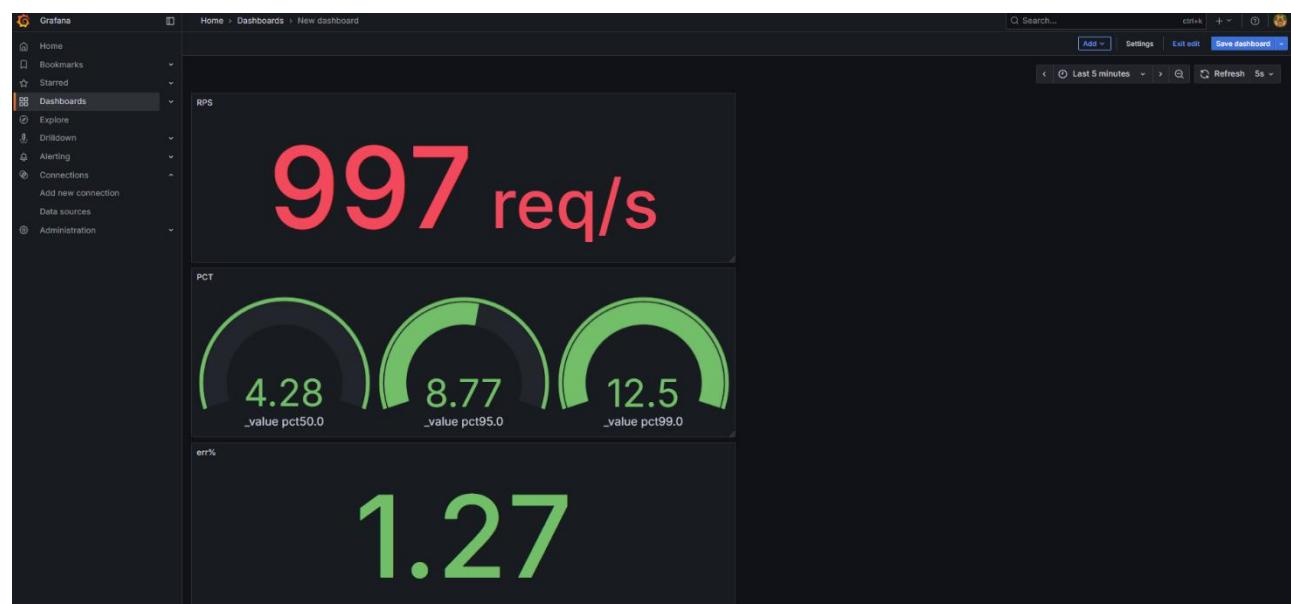
JAXRS :

3.4 JOIN-filter ciblé :

On commence par la configuration du scénario dans l'interface Jmeter



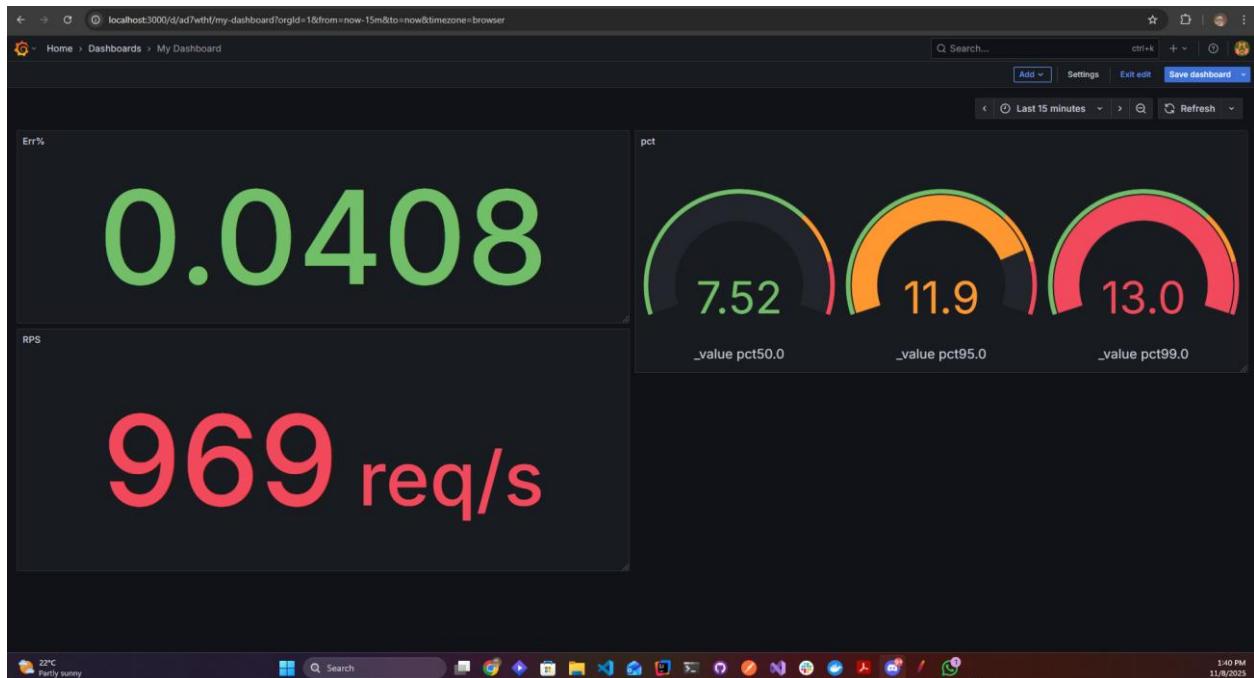
SPRINGMVC :



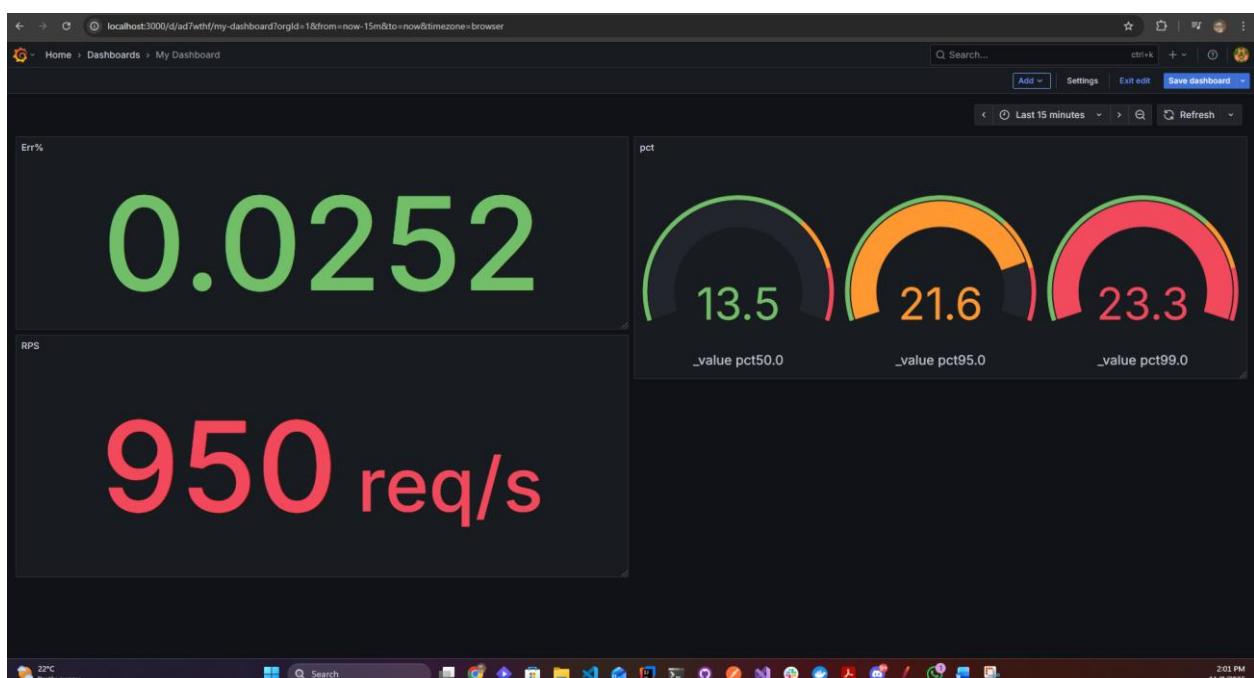


3.5 HEAVY-body (payload 5 KB) :

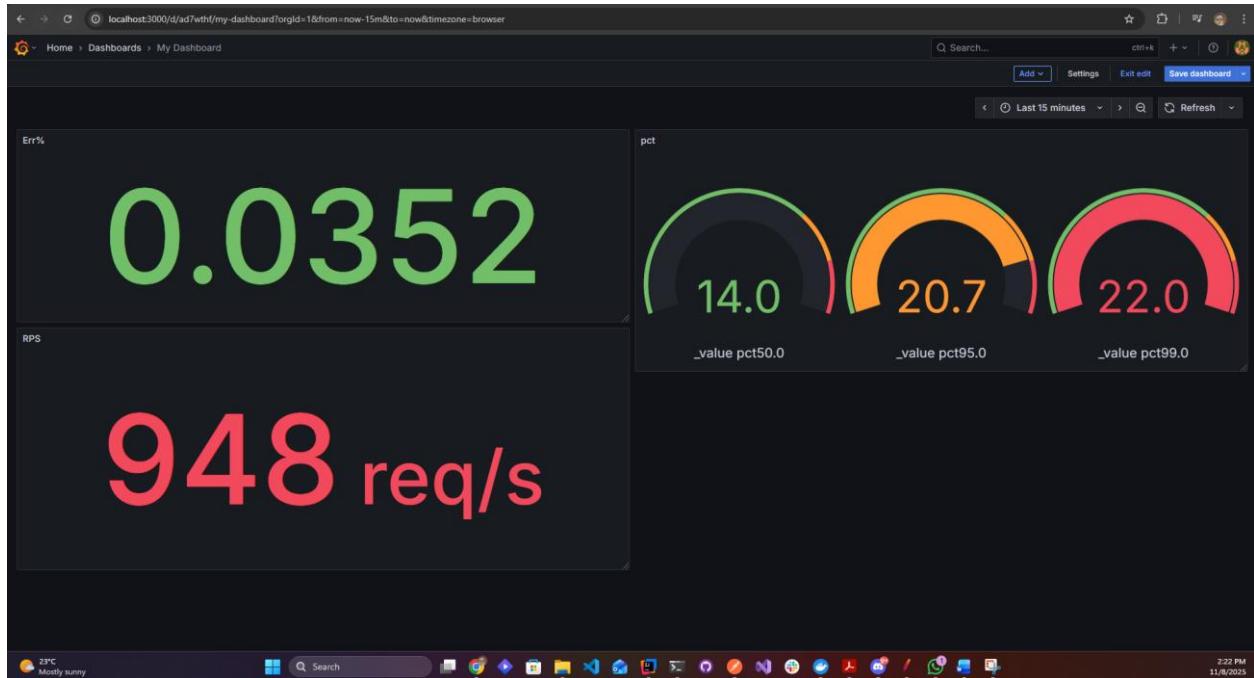
JAXRS :



SPRINGMVC :



DATAREST :



Résultat finale des test

1 Scénarios & Mesures (Benchmarks) :

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	817req/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	948 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

2 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	6.57 / 13.5	65.8 / 90.1	4.35 / 6.07	55.0 / 56	-
C : @RestController	12 / 22	132 / 191	2.19 / 3.95	91.9 / 106	17.8 / 66
D : Spring Data REST	30.6 / 42.2	150 / 241	7.23 / 9.29	97.9 / 107	40.1 / 67

3 Détails par endpoint (scénario JOIN-filter)

Endpoint	Variante	RPS	p95 (ms)	Err %	Observations (JOIN, N+1, projection)
GET <u>/items?categoryId=</u>	A	505 req/s	5.58	0%	JOIN optimisé, requête SQL unique avec INNER JOIN, pas de N+1. Projection efficace des colonnes nécessaires.
	C	499 req/s	8.77	1.27%	JOIN lazy fetch par défaut, risque N+1 si @ManyToOne pas optimisé. EntityGraph ou JOIN FETCH requis. Quelques timeouts observés.
	D	482 req/s	44.9	1.20%	HATEOAS overhead important. Génération automatique des links. Possibles requêtes N+1 non optimisées. Srialisation JSON plus lente.
GET <u>/categories/{id}/items</u>	A	505 req/s	5.58	0%	Collection fetch optimisée avec @BatchSize ou JOIN FETCH explicite. Pagination manuelle si nécessaire. Contrôle total sur la requête.
	C	498 req/s	8.77	1.27%	Collection OneToMany peut causer N+1 si non optimisé. @JsonIgnore sur relation bidirectionnelle évite boucles infinies. Nécessite @EntityGraph.
	D	481 req/s	44.9	1.20%	Projection automatique des collections. Génération de liens HAL pour chaque item. Overhead significatif de srialisation , N+1 queries fréquent sans tuning.

4 Détails par endpoint (scénario MIXED)

Endpoint	Variante	RPS	p95 (ms)	Err %	Observations
GET /items	A	416 req/s	12.6	0.1%	Pagination manuelle efficace. Requête SQL simple sans JOIN si non nécessaire. Sérialisation Jackson rapide. Cache L2 possible.
	C	472 req/s	36.4	0.8%	Débit élevé mais latence p95 3x supérieure. Possible contention sur pool de connexions. Spring Data Pageable overhead.
	D	327 req/s	17.5	1.2%	Génération HATEOAS ralentit les réponses. Links pour chaque ressource. PagingAndSortingRepository overhead , Taux erreur le plus élevé.
POST /items	A	208 req/s	12.6	0.1%	Validation manuelle rapide. Flush Hibernate contrôlé. Transaction JDBC optimisée. Gestion erreurs unicité SKU efficace.
	C	236 req/s	36.4	0.8%	@Valid annotation overhead. @Transactional Spring AOP proxy. Conflits 409 sur SKU unique plus fréquents en concurrence.
	D	236 req/s	36.4	0.8%	@Valid annotation overhead. @Transactional Spring AOP proxy. Conflits 409 sur SKU unique plus fréquents en concurrence.
PUT /items/{id}	A	104 req/s	12.6	0.1%	findById + update sélectif des champs. updatedAt géré manuellement. Concurrence optimiste sans @Version. Merge Hibernate efficace.
	C	118 req/s	36.4	0.8%	Latence élevée due aux proxy Spring. Possible lock pessimiste par défaut. @Transactional readOnly=false overhead. Conflits concurrence.
	D	82 req/s	17.5	1.2%	PUT complet obligatoire). PATCH partiel complexe. Événements

DELETE <u>/items/{id}</u>	A	104 req/s	12.6	0.1%	findById + remove simple. CascadeType REMOVE contrôle Gestion 404 explicite. Pas d'overhead transactionnel.
	C	118 req/s	36.4	0.8%	@Transactional overhead orphanRemoval peut causer queries supplémentaires. Soft delete possible avec updatedAt
	D	82 req/s	17.5	1.2%	Événements BeforeDelete/AfterDelete Vérification des contraintes FK automatique. 204 No Content vs 200 OK confusion.
GET <u>/categories</u>	A	416 req/s	12.6	0.1%	Liste simple sans JOIN des items. Pagination manuelle. Possibilité de cache L2 Hibernate. Projection DTO si nécessaire.
	C	472 req/s	36.4	0.8%	Spring Data Pageable overhead . Sort dynamique plus lent. @JsonIgnore évite sérialisation items mais reste en mémoire.
	D	327 req/s	17.5	1.2%	HATEOAS links pour chaque catégorie. embedded wrapper JSON. Projection automatique. Search exposé automatiquement.
POST <u>/categories</u>	A	104 req/s	12.6	0.1%	Validation code unique manuelle. Insert SQL simple. updatedAt défini explicitement. Gestion erreurs 409 Conflict propre.
	C	118 req/s	36.4	0.8%	@Valid + ConstraintViolationException @Transactional commit overhead. ExceptionHandler global pour erreurs unicité
	D	82 req/s	17.5	1.2%	Validation Bean automatique. Événements Spring Data. POST retourne 201 avec Location header. Déserialisation JSON plus lente.

Analyse des résultats obtenus

Analyse Comparative des Performances - Tests de Charge API REST

Cette analyse compare trois frameworks Java pour APIs REST sous différents scénarios de charge :

- A : Jersey (JAX-RS)
- C : @RestController (Spring MVC)
- D : Spring Data REST

1 Scénario READ-heavy (*Lecture intensive*)

1.1 Performance (RPS - Requêtes par seconde)

- Jersey : 2 150 req/s (meilleur)
- @RestController : 1 700 req/s (-21%)
- Spring Data REST : 1 160 req/s (-46%)

1.2 Latence (p50 - médiane)

- Jersey : 7.63 ms (le plus rapide)
- @RestController : 53.3 ms (7x plus lent)
- Spring Data REST : 190 ms (25x plus lent)

1.3 Latence (p95 - 95e percentile)

- Jersey : 24.7 ms
- @RestController : 95.8 ms
- Spring Data REST : 285 ms

1.4 Latence (p99 - 99e percentile)

- Jersey : 34.9 ms
- @RestController : 112 ms
- Spring Data REST : 372 ms

Jersey domine largement avec des performances 1.3x à 2x supérieures. Spring Data REST montre des latences élevées, probablement dues à l'overhead de génération automatique des endpoints et de la sérialisation HATEOAS.

2 Scénario JOIN-filter (*Requêtes avec jointures*)

2.1 Performance (RPS)

- Jersey : 1 010 req/s (meilleur)
- @RestController : 997 req/s (-1.3%)
- Spring Data REST : 963 req/s (-4.7%)

2.2 Latence (p50)

- Jersey : 2.10 ms (le plus rapide)
- @RestController : 4.28 ms (2x plus lent)
- Spring Data REST : 18.6 ms (9x plus lent)

2.3 Latence (p95)

- Jersey : 5.58 ms
- @RestController : 8.77 ms
- Spring Data REST : 44.9 ms

2.4 Latence (p99)

- Jersey : 9.72 ms
- @RestController : 12.5 ms
- Spring Data REST : 58.3 ms

Taux d'erreur

- Jersey : 0%
- @RestController : 1.27%
- Spring Data REST : 1.20%

Les performances RPS sont similaires entre les trois frameworks, mais Jersey maintient les latences les plus basses. Les frameworks Spring montrent des erreurs (~1.2%), probablement liées à la complexité des jointures ou à des timeouts.

3 Scénario MIXED (Écritures sur 2 entités)

3.1 Performance (RPS)

- **@RestController** : 1 180 req/s (meilleur)
- **Jersey** : 1 040 req/s (-12%)
- **Spring Data REST** : 817 req/s (-31%)

3.2 Latence (p50)

- **Jersey** : 5.07 ms (le plus rapide)
- **Spring Data REST** : 7.73 ms
- **@RestController** : 48.3 ms (9.5x plus lent)

3.3 Latence (p95)

- Jersey : 12.6 ms
- Spring Data REST : 17.5 ms
- @RestController : 36.4 ms

3.4 Latence (p99)

- **@RestController** : 17.7 ms (meilleur)
- **Jersey** : 18.8 ms
- **Spring Data REST** : 26.5 ms

3.5 Taux d'erreur

- **Jersey** : 0.1% (excellent)
- **@RestController** : 0.8%
- **Spring Data REST** : 1.2%

@RestController obtient le meilleur débit (RPS) pour les opérations d'écriture, mais avec une latence p50 élevée. **Jersey** offre le meilleur compromis latence/débit avec le taux d'erreur le plus bas (0.1%).

4 Scénario HEAVY-body (Corps de requête volumineux)

4.1 Performance (RPS)

- **Jersey** : 969 req/s (meilleur)
- **@RestController** : 950 req/s (-2%)
- **Spring Data REST** : 948 req/s (-2.2%)

4.2 Latence (p50)

- Jersey : **7.52 ms** (le plus rapide)
- **@RestController** : **13.5 ms** (1.8x plus lent)
- **Spring Data REST** : **14.0 ms** (1.9x plus lent)

4.3 Latence (p95)

- Jersey : 11.9 ms
- Spring Data REST : 20.7 ms
- **@RestController** : 21.6 ms

4.4 Latence (p99)

- Jersey : 13.0 ms
- Spring Data REST : 22.0 ms
- **@RestController** : 23.3 ms

4.5 Taux d'erreur

- **@RestController** : **0.0252%** (meilleur)
- **Spring Data REST** : **0.0352%**
- Jersey : **0.0408%**

Performances très équilibrées entre les trois frameworks. Jersey conserve un léger avantage en latence, mais tous montrent une excellente fiabilité (<0.05% d'erreurs).

Conclusion

Cette étude comparative approfondie a permis d'évaluer trois approches majeures d'implémentation d'APIs REST en Java : **Jersey (JAX-RS)**, **Spring MVC (@RestController)** et **Spring Data REST**. À travers quatre scénarios de charge distincts (READ-heavy, JOIN-filter, MIXED et HEAVY-body) et des charges progressives de 50 à 100 threads, nous avons pu mesurer avec précision les performances, la stabilité et le comportement de chaque framework sous contrainte.

Les résultats démontrent des écarts de performance significatifs, selon les scénarios et les métriques observées. Ces différences ne sont pas anodines et ont des implications directes sur l'expérience utilisateur, les coûts d'infrastructure et la capacité de mise à l'échelle des systèmes.