ECOLE MAROCAINE DES SCIENCES DE L'INGENIEUR

Membre de

HONORIS UNITED UNIVERSITIES

Benchmark de performances des Web Services REST

RAPPORT DU PROJET

Architecture Microservices

Ingénierie Informatique et Réseaux

Réalisé par :
Naim Mohamed
Abdelwahid
Amdjar

Encadré par :
Pr. LACHGAR

Introduction générale

Dans le paysage actuel du développement d'applications, les API REST jouent un rôle central au sein des architectures microservices et des systèmes distribués. Face à la variété des frameworks et des approches disponibles dans l'écosystème Java, les équipes de développement doivent faire des choix technologiques déterminants, influençant directement la performance, la maintenabilité et la capacité d'évolution des applications.

Cette pluralité de solutions des implémentations légères comme JAX-RS aux frameworks complets tels que Spring conduit à une interrogation essentielle : quel impact réel les différents niveaux d'abstraction de ces technologies ont-ils sur les performances ?

Objectifs du Benchmark

Cette étude comparative a pour objectif d'évaluer de manière objective l'impact des choix de conception REST sur plusieurs dimensions essentielles. Elle examine d'abord les performances opérationnelles, en analysant la latence à différents percentiles (p50, p95, p99) pour mieux comprendre le comportement des utilisateurs, le débit maximal en requêtes par seconde ainsi que la fiabilité du système, à travers le taux d'erreurs et la stabilité sous diverses charges. Elle s'intéresse également à l'efficacité des ressources, en étudiant la consommation mémoire, l'utilisation du CPU, l'impact du Garbage Collector sur les performances et la gestion optimale des threads et de la concurrence. L'étude analyse aussi les compromis entre la productivité des développeurs et la performance pure, entre le contrôle manuel offert par JAX-RS ou Spring MVC avec @RestController et l'abstraction automatique proposée par Spring Data REST, ainsi qu'entre la complexité de mise en œuvre et les gains en maintenance.
L'analyse porte sur trois approches représentatives de l'écosystème Java : JAX-RS (Jersey), standard JEE léger offrant un contrôle fin, Spring MVC avec @RestController, framework d'entreprise complet et robuste, et Spring Data REST, solution d'exposition automatique permettant une productivité maximale.
La méthodologie adoptée repose sur un environnement contrôlé avec une même base PostgreSQL et une configuration matérielle identique, et sur des scénarios réalistes simulant quatre profils de charge correspondant à des cas d'usage concrets. L'instrumentation complète, réalisée à l'aide de JMeter, Prometheus, Grafana et InfluxDB, permet d'obtenir des mesures reproductibles grâce à un protocole standardisé avec des paliers progressifs.
Les résultats de cette étude offrent des insights exploitables pour guider les décisions d'architecture en fonction des contraintes spécifiques des projets, optimiser l'allocation des ressources infrastructurelles, anticiper les goulots d'étranglement dans les cas critiques et équilibrer judicieusement performance et productivité. Ce benchmark s'adresse aussi bien aux architectes techniques qu'aux développeurs, en fournissant une vision quantitative des compromis inhérents au choix des technologies REST dans l'écosystème Java moderne.

Outils Utilisé

- ***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.

- ***InfluxDB :***

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

- ***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

- ***Docker :***

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

- ***Prometheus :***

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

- ***PgAdmin :***

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

Réalisation

- ***Configuration matérielle & logicielle :***

Élément	Valeur
Machine (CPU, cœurs, RAM)	12 , 6 , 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

- ***Environnement Docker :***

Docker compose

```yaml
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
```

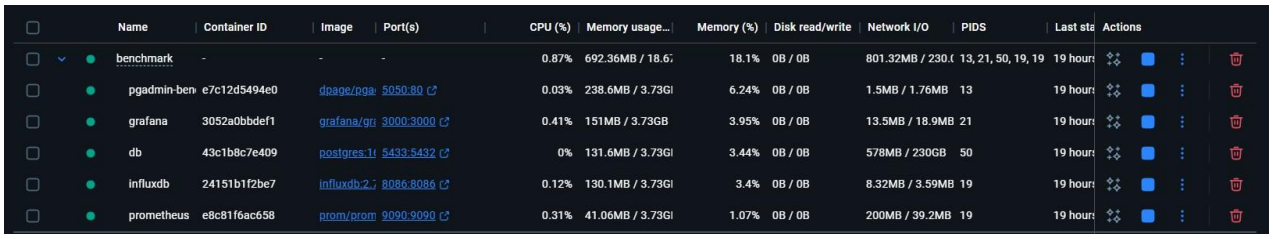
```yaml
# --- pgAdmin ---
pgadmin:
  image: dpage/pgadmin4
  container_name: pgadmin-benchmark
  restart: always
  environment:
    PGADMIN_DEFAULT_EMAIL: admin@admin.com
    PGADMIN_DEFAULT_PASSWORD: admin
  ports:
    - "5050:80"
  depends_on:
    - db
  networks:
    - benchmark-net
```

```yaml
# --- 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
```

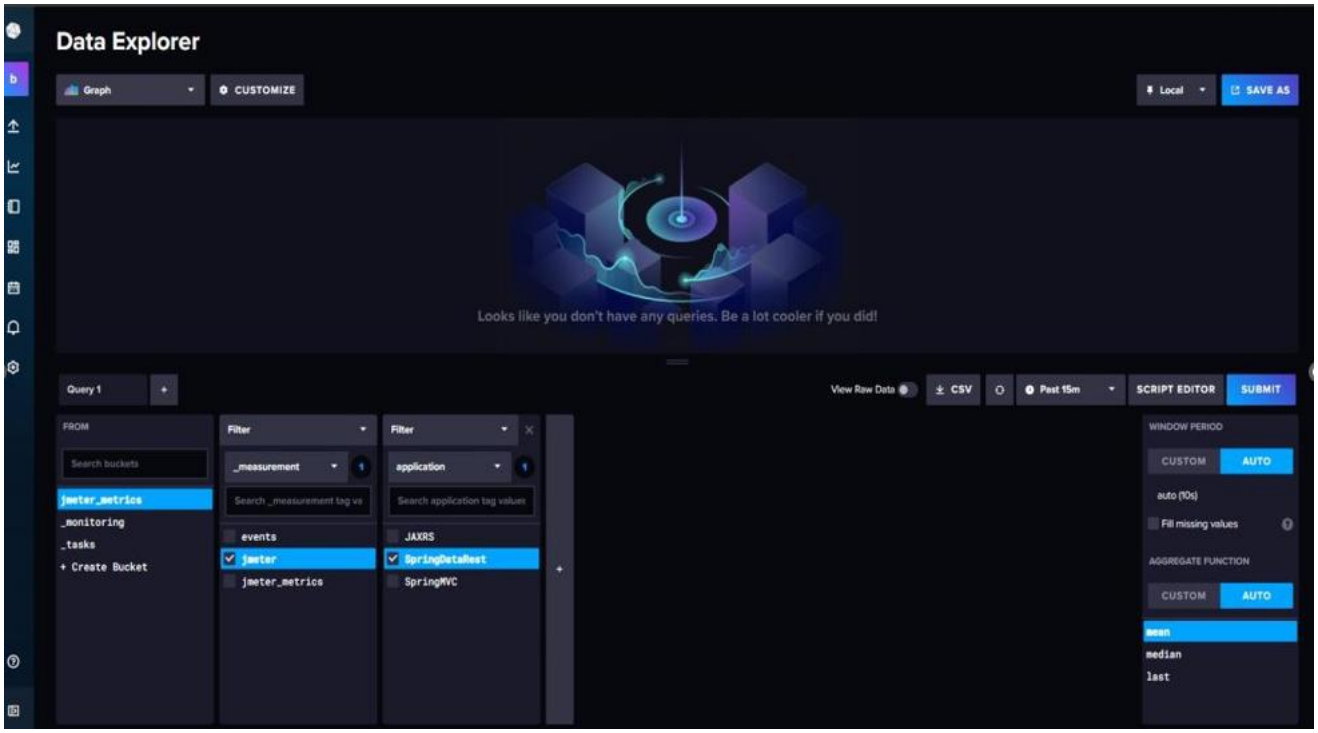
```yaml
# --- InfluxDB (pour JMeter backend listener) ---
influxdb:
  image: influxdb:2.7
  container_name: influxdb
  restart: always
  ports:
    - "8086:8086"
  environment:
    - DOCKER_INFLUXDB_INIT_MODE=setup
    - DOCKER_INFLUXDB_INIT_USERNAME=admin
    - DOCKER_INFLUXDB_INIT_PASSWORD=admin123
    - 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
```

Docker conteneurs:



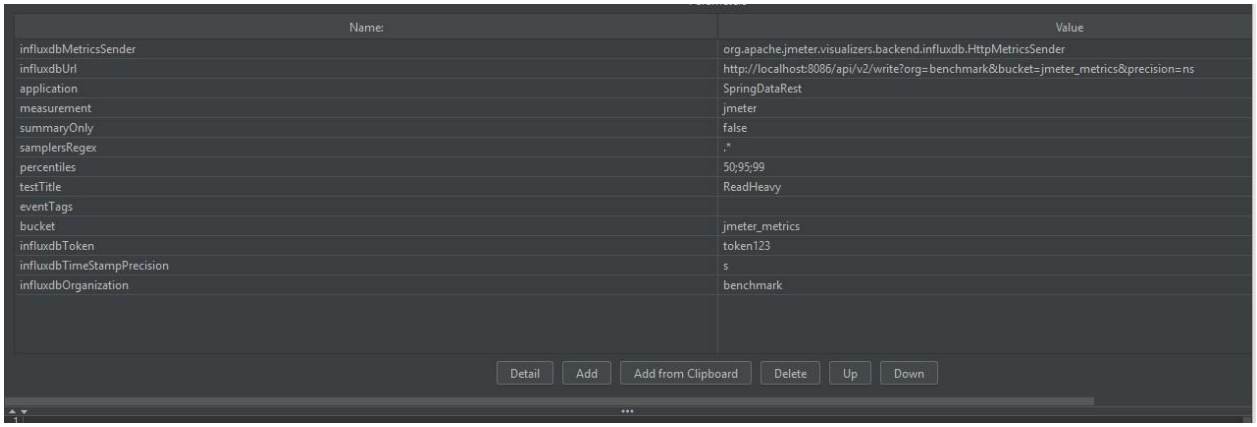
Name	Container ID	Image	Port(s)	CPU (%)	Memory usage...	Memory (%)	Disk read/write	Network I/O	PIDS	Last sta	Actions
benchmark	-	-	-	0.87%	692.36MB / 18.67	18.1%	0B / 0B	801.32MB / 230.(	13, 21, 50, 19, 19	19 hours	
pgadmin-ben	e7c12d5494e0	dpage/pga	5050:80	0.03%	238.6MB / 3.73GI	6.24%	0B / 0B	1.5MB / 1.76MB	13	19 hours	
grafana	3052a0bbdef1	grafana/gr	3000:3000	0.41%	151MB / 3.73GB	3.95%	0B / 0B	13.5MB / 18.9MB	21	19 hours	
db	43c1b8c7e409	postgres:1(	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/prom	9090:9090	0.31%	41.06MB / 3.73GI	1.07%	0B / 0B	200MB / 39.2MB	19	19 hours	

InfluxDB :

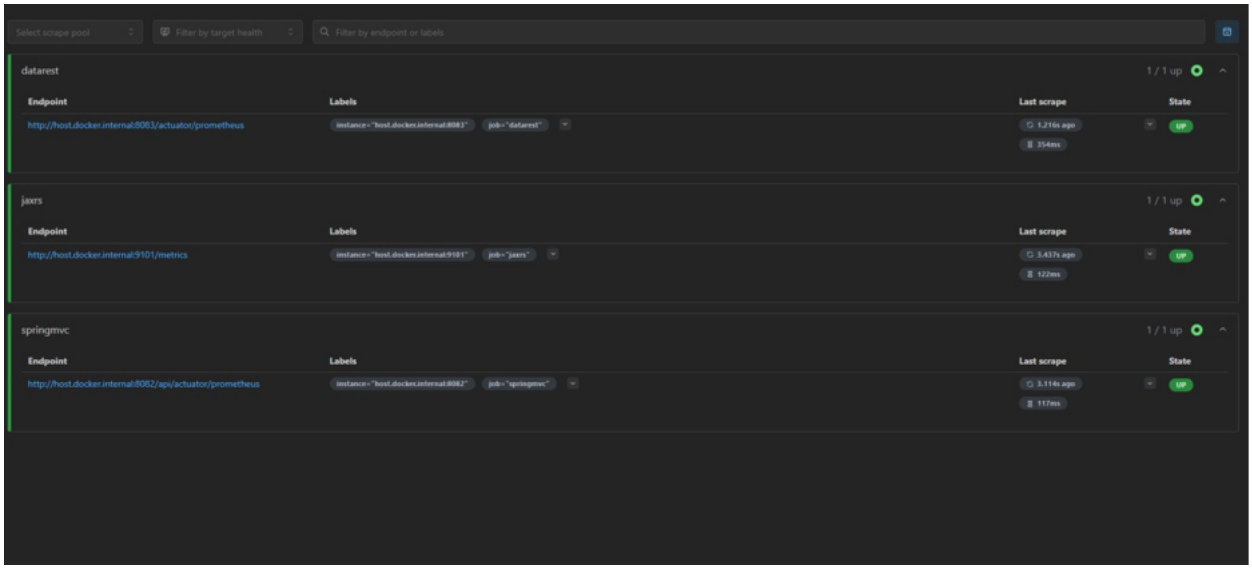


Jmeter configuration :

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



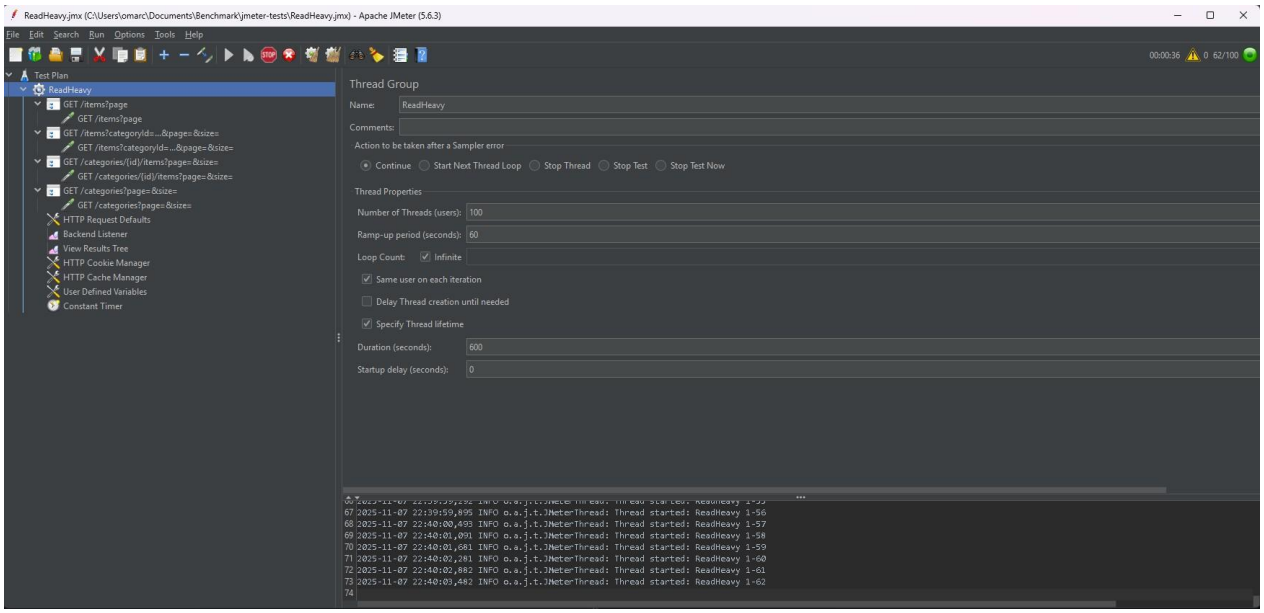
Interface Prometheus :

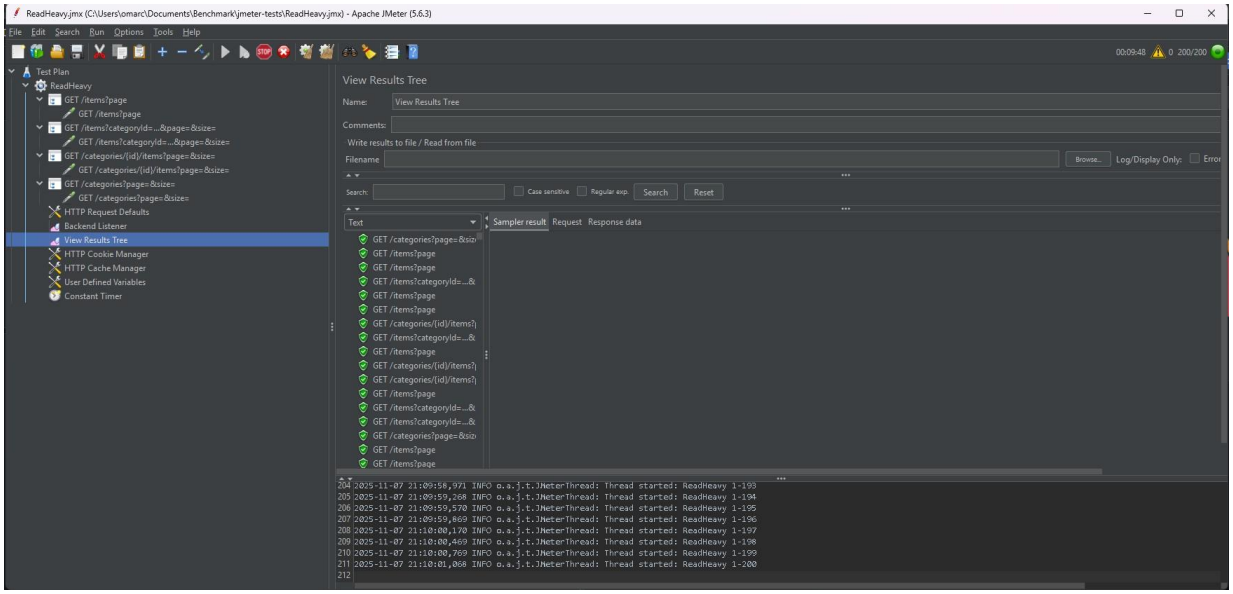


Scénarios de charge (JMeter) :

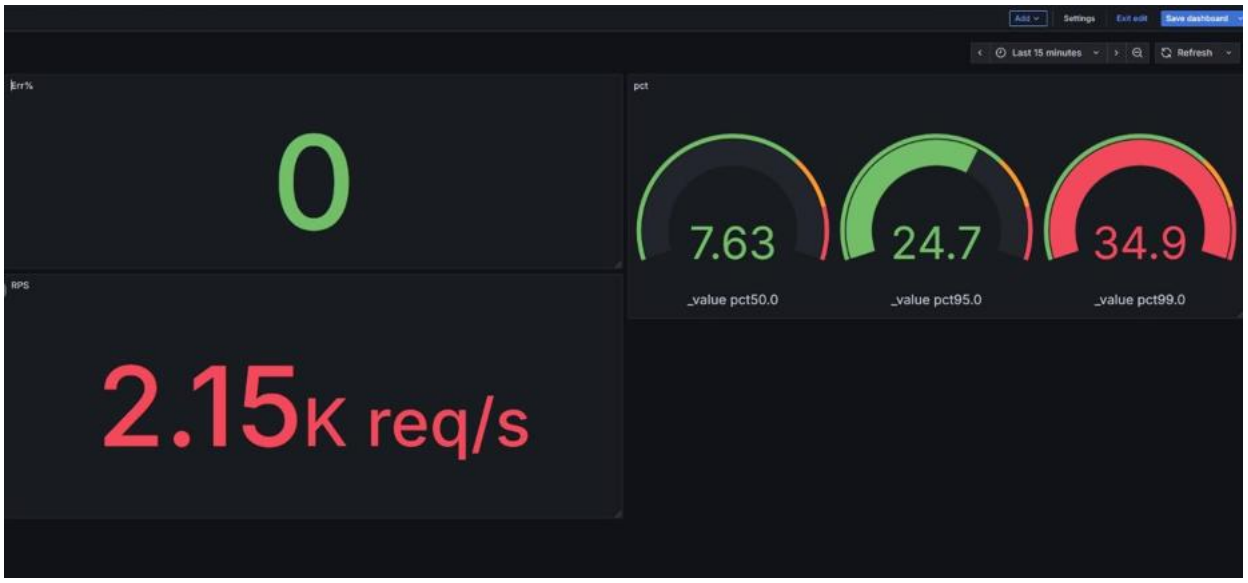
READ-heavy (relation incluse) :

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



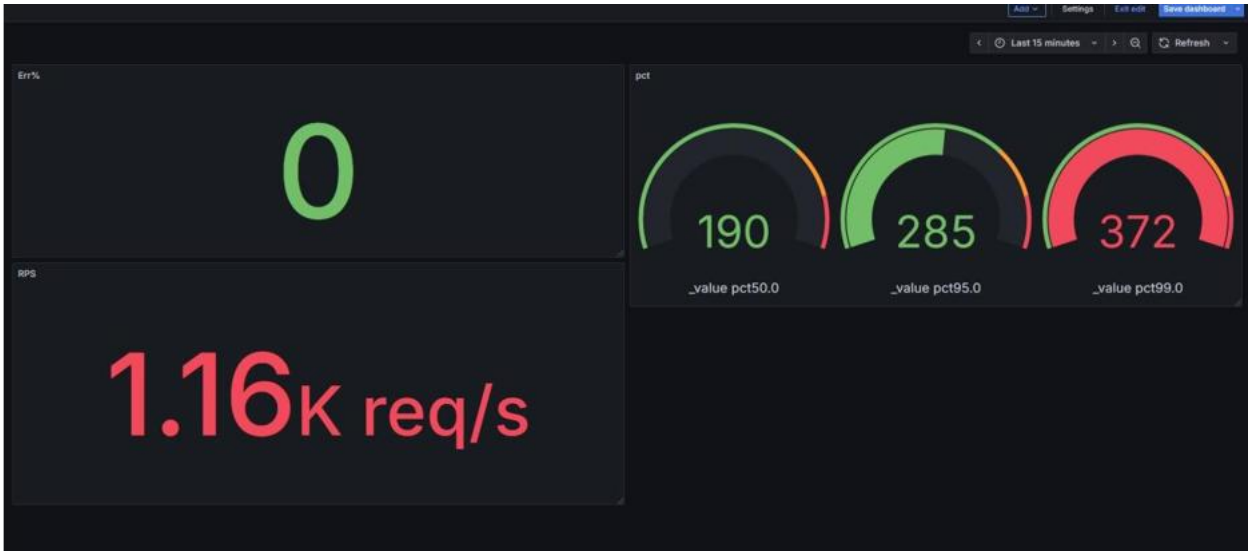


JAXRS 100 USER

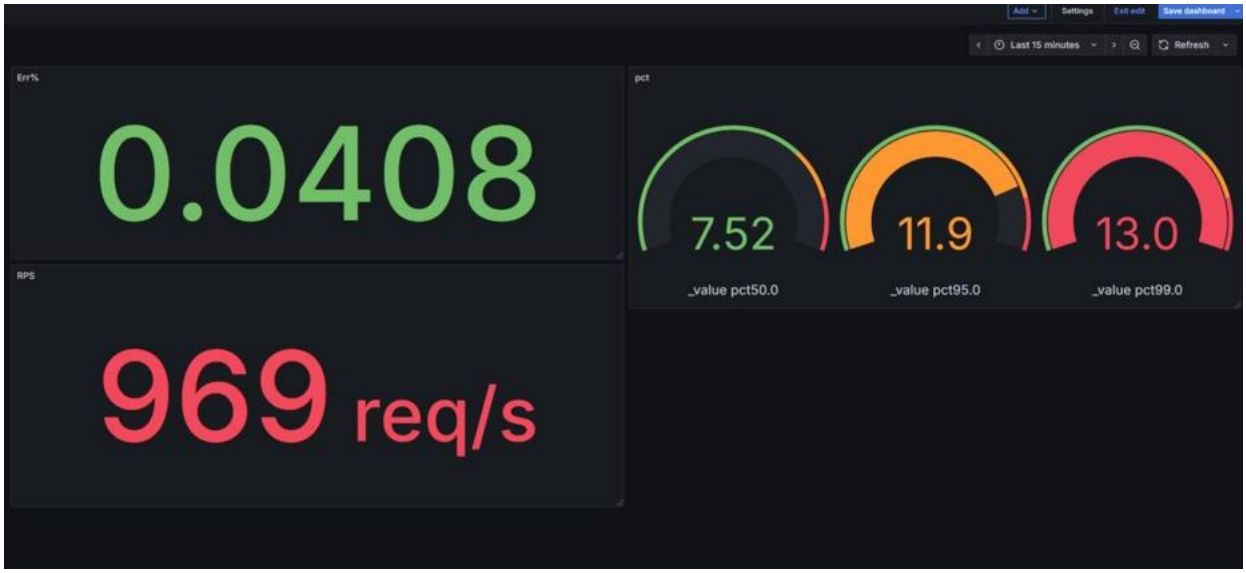


SPRINGMVC 100 USER :

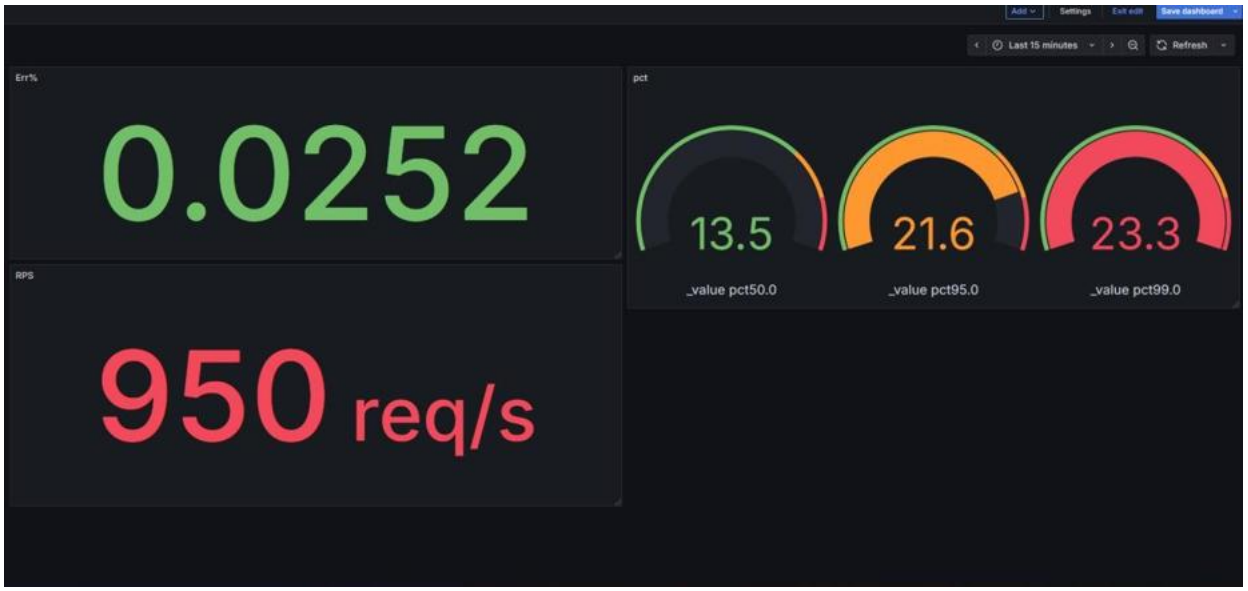
DATAREST 100 USER :



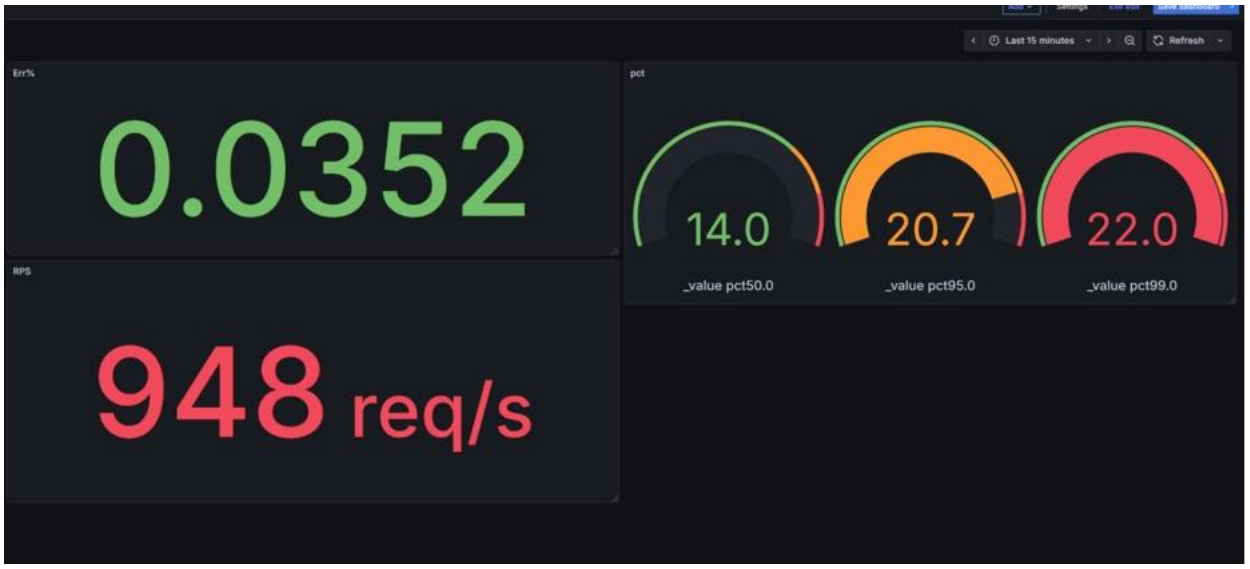
JAXRS 60 USER :



SPRINGMVC 60 USER :



DATAREST 60 USER :



Ressources JVM (Prometheus) :

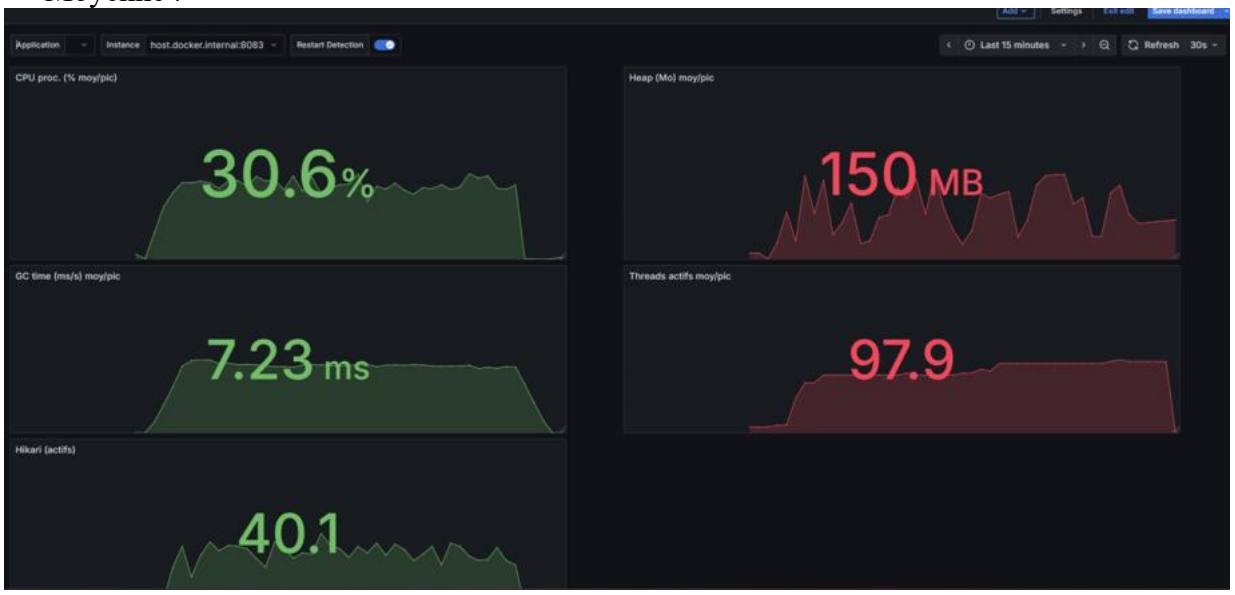
SpringMVC(ReadHeavy 100)

Moyenne :

Max

SpringDataRest(ReadHeavy 100) :

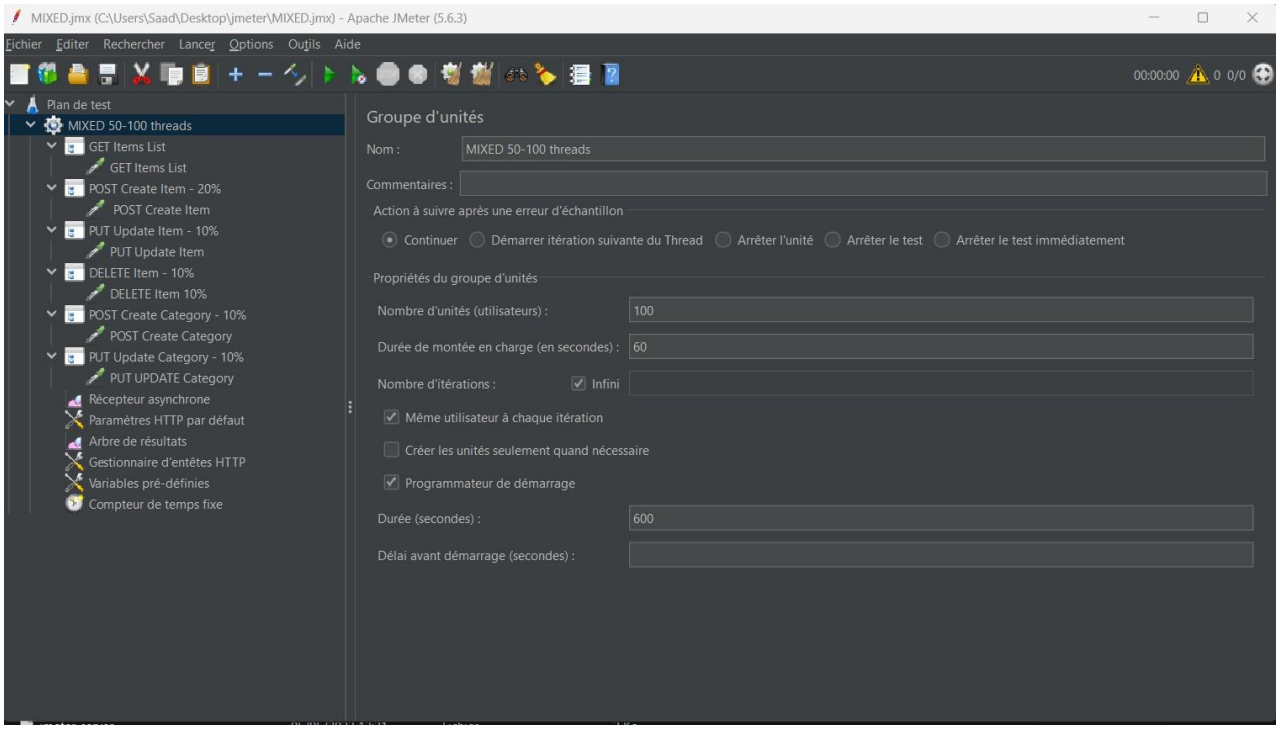
Moyenne :



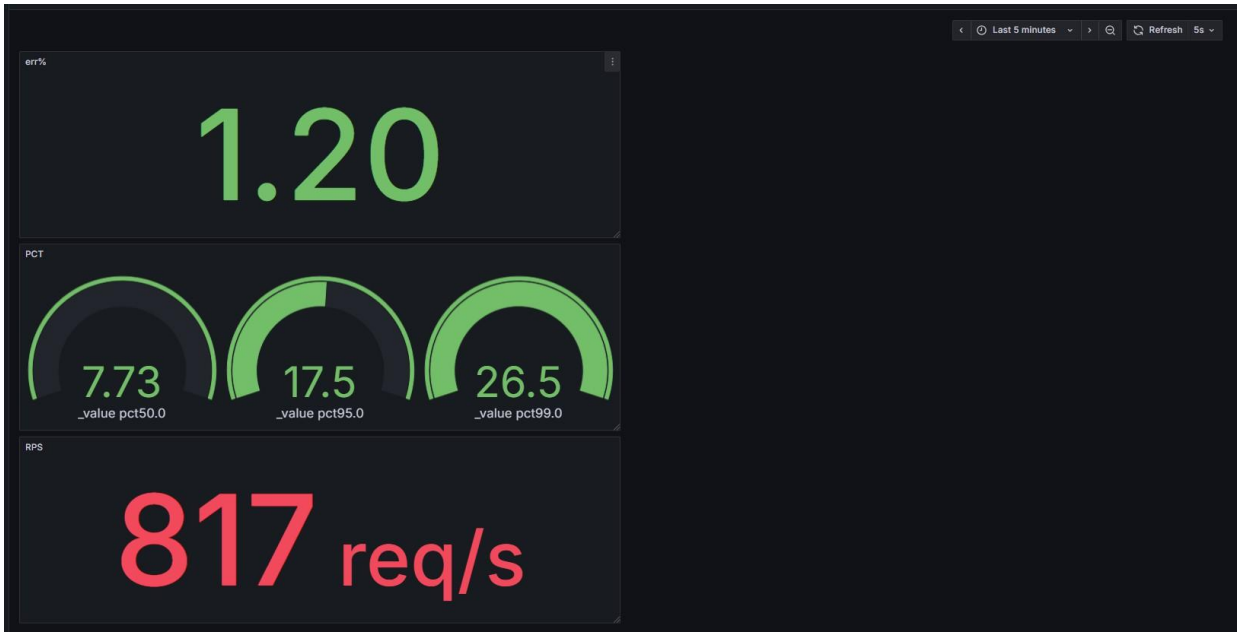
Max :

MIXED (écritures sur deux entités) :

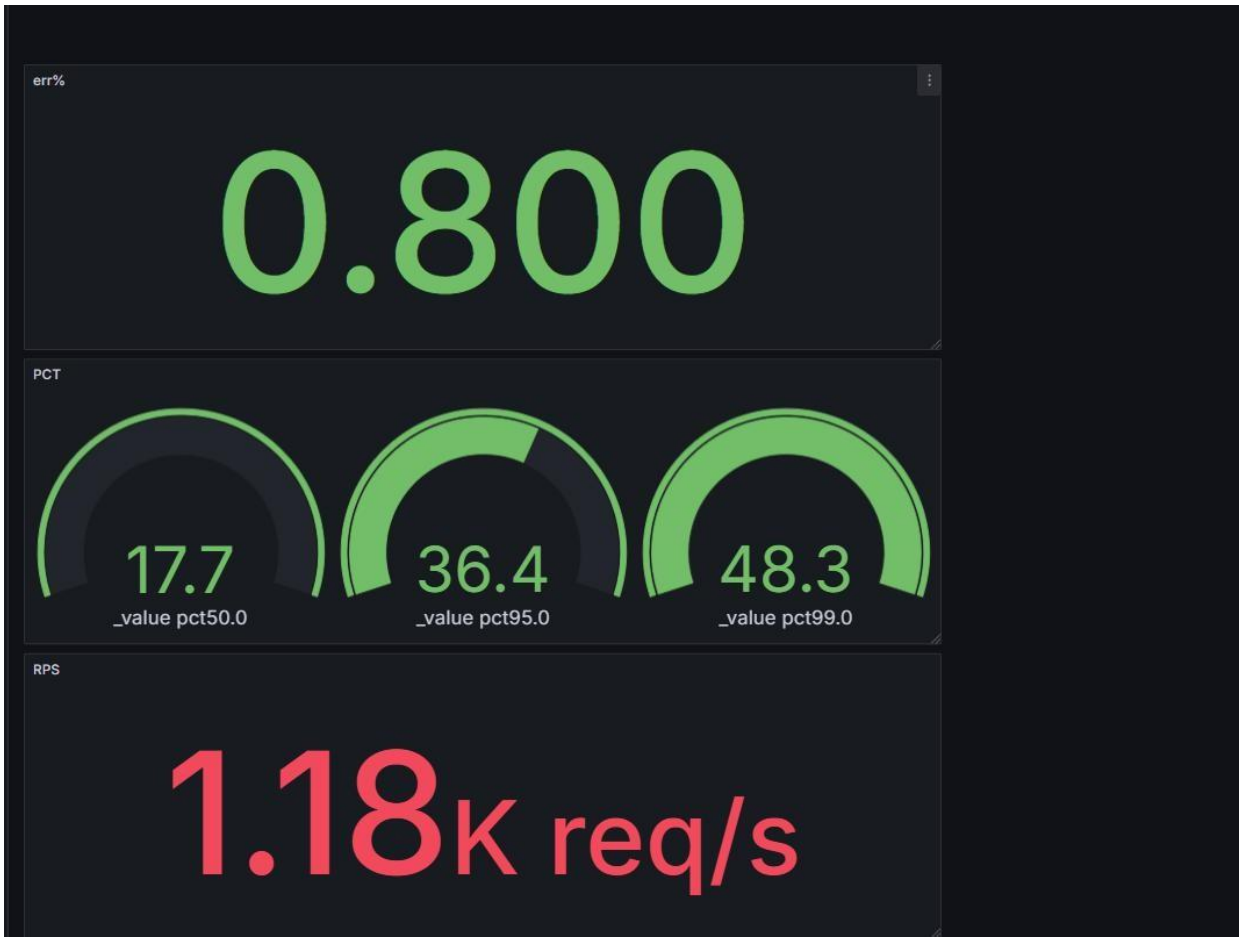
On commence par la configuration du scénario dans l'interface Jmeter



DATAREST :



SPRINGMVC :

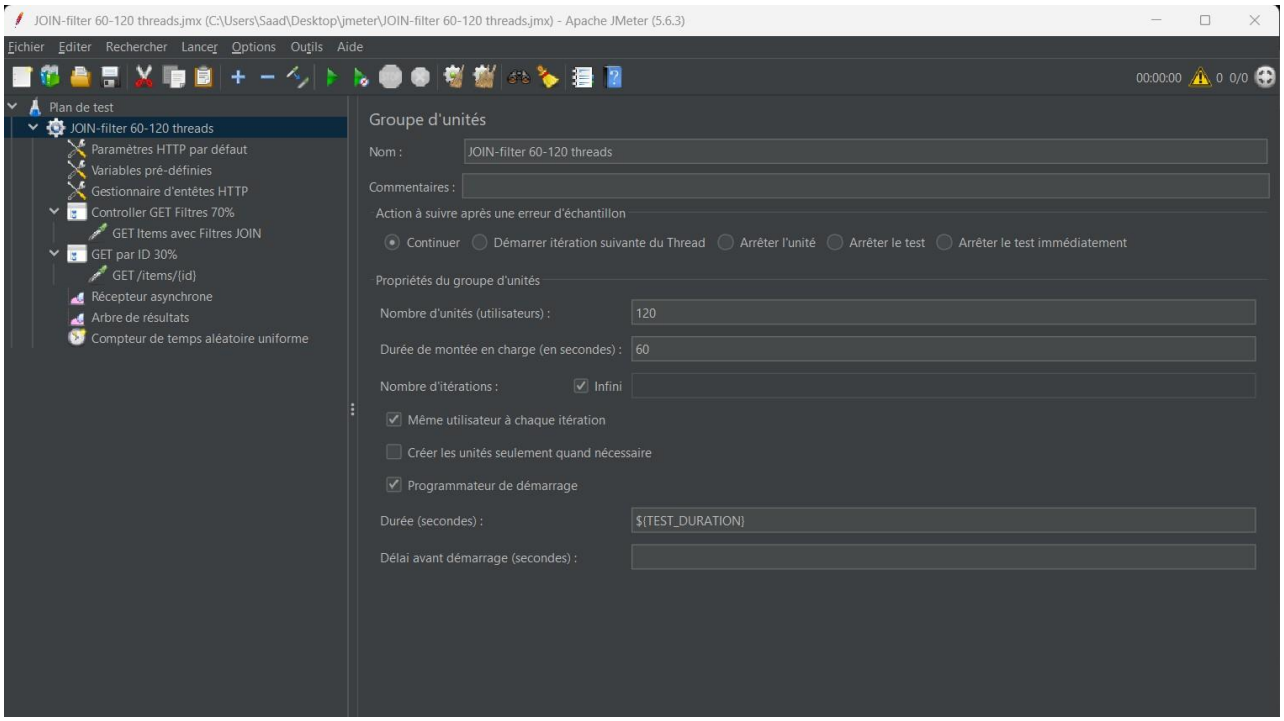

err%
0.800
PCT
17.7
_value pct50.0
36.4
_value pct95.0
48.3
_value pct99.0
RPS
1.18K req/s

JAXRS :

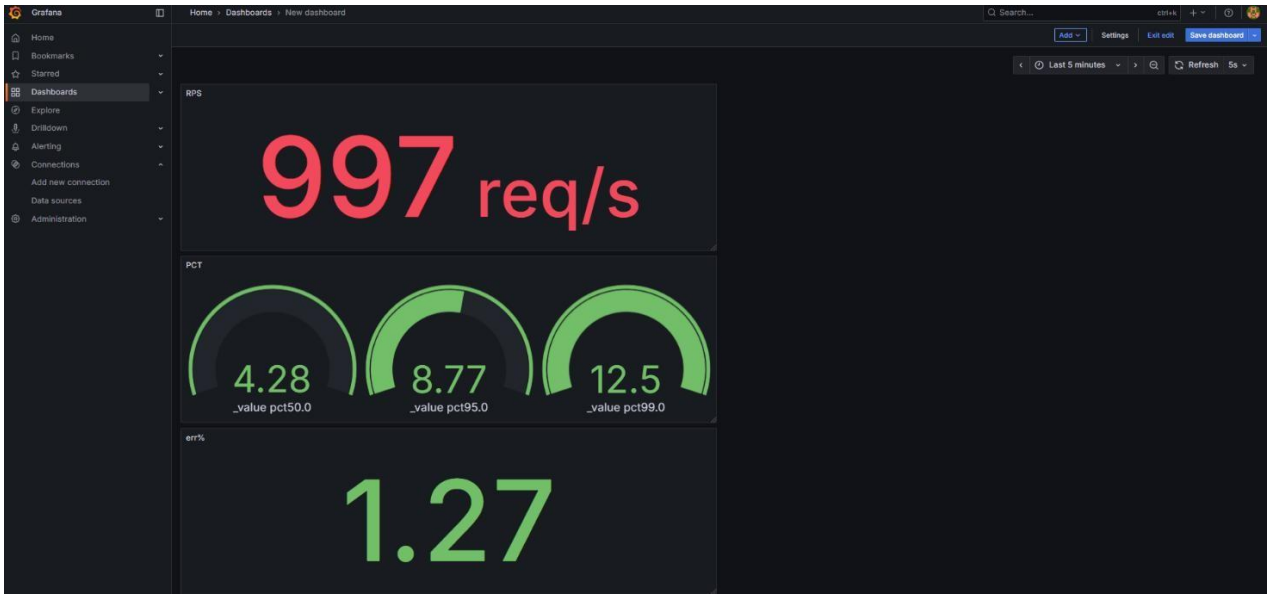

err%
0.100
PCT
5.07
_value pct50.0
12.6
_value pct95.0
18.8
_value pct99.0
RPS
1.04K req/s

JOIN-filter ciblé :

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

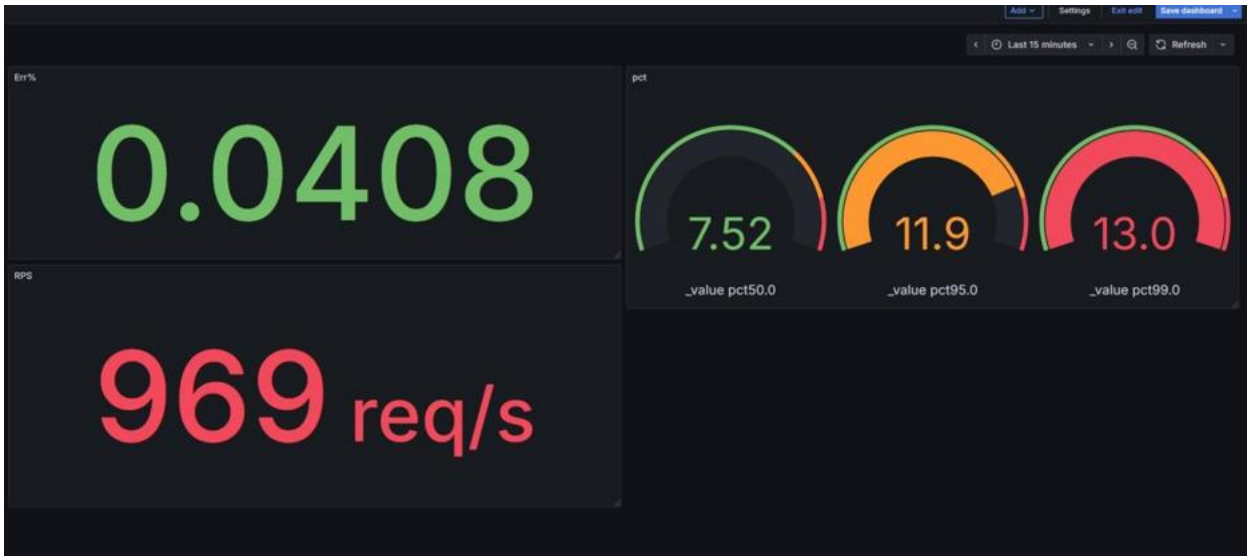


SPRINGMVC :

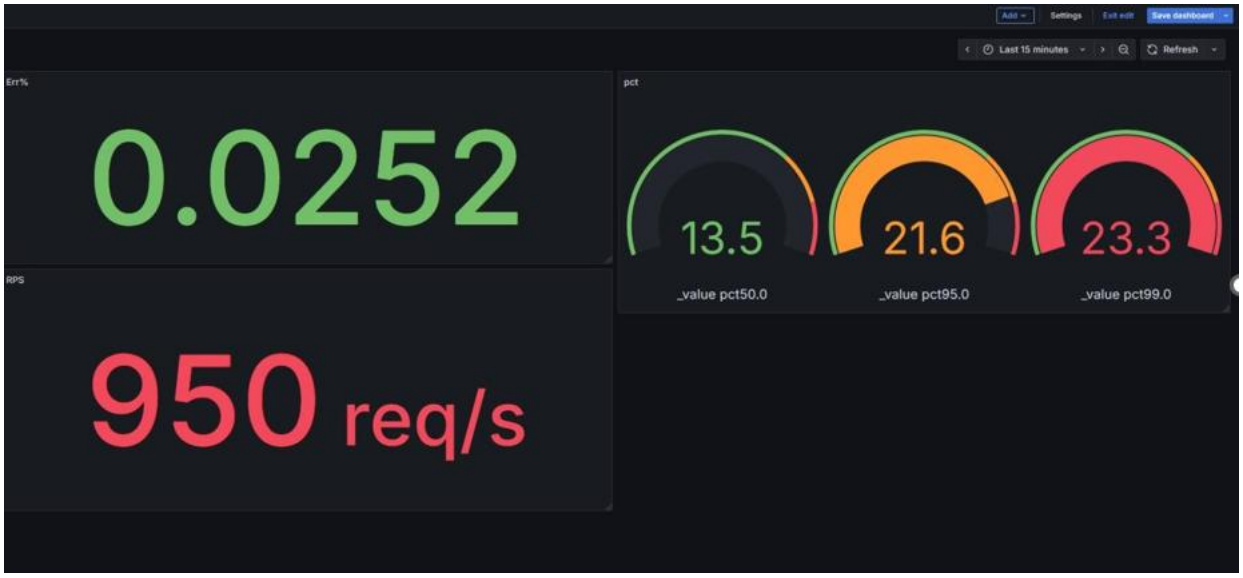


HEAVY-body (payload 5 KB) :

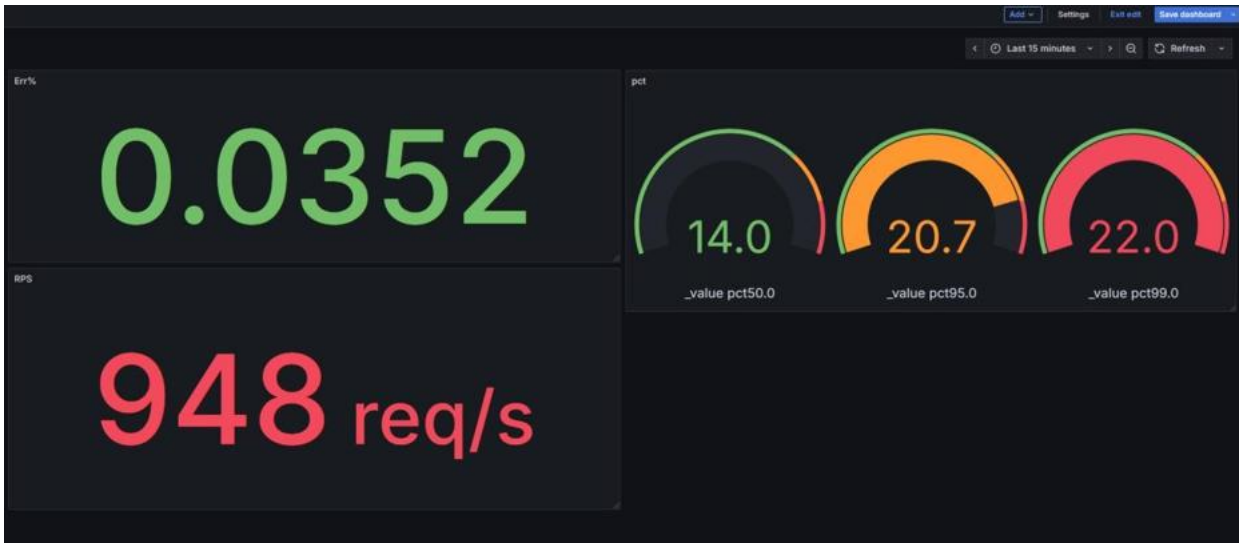
JAXRS :



SPRINGMVC :



DATAREST :


Add
Settings
Exit edit
Save dashboard
Last 15 minutes
Refresh
Err%
0.0352
RPS
948 req/s
pct
14.0
20.7
22.0
_value pct50.0
_value pct95.0
_value pct99.0

Résultat finale des test

- ***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

- ***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

- ***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%	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. Sérialisation JSON plus lente.
GET /categories/{id}/items	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 sérialisation. N+1 queries fréquent sans tuning.

- ***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 /items/{id}	A	104 req/s	12.6	0.1%	findById + remove simple. CascadeType.REMOVE contrôlé. 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 /categories	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 exposed automatiquement.
POST /categories	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. Désérialisation JSON plus lente.

Analyse des resultats 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**

- ***Scénario READ-heavy (Lecture intensive)***

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%)

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)

Latence (p95 - 95e percentile)

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

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.

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

Performance (RPS)

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

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)

Latence (p95)

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

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.

- ***Scénario MIXED (Écritures sur 2 entités)***

Performance (RPS)

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

Latence (p50)

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

Latence (p95)

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

Latence (p99)

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

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%).

- ***Scénario HEAVY-body (Corps de requête volumineux)***

Performance (RPS)

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

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)

Latence (p95)

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

Latence (p99)

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

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 analyse comparative approfondie a permis d'évaluer trois principales méthodes d'implémentation d'APIs REST en Java : Jersey (JAX-RS), Spring MVC (@RestController) et Spring Data REST. En testant quatre scénarios de charge différents (READ-heavy, JOIN-filter, MIXED et HEAVY-body) avec un nombre de threads variant de 50 à 100, nous avons pu mesurer avec précision les performances, la stabilité et le comportement de chaque framework sous pression.

Les résultats révèlent des différences de performance notables selon les scénarios et les indicateurs étudiés. Ces variations sont significatives et ont des conséquences directes sur l'expérience utilisateur, les coûts d'infrastructure et la capacité à faire évoluer les systèmes.