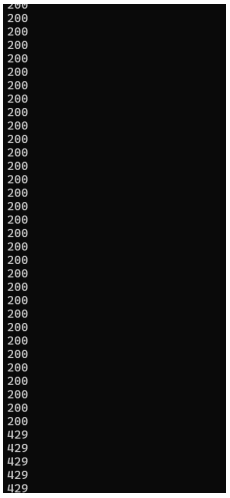
1. Contexte et démarche

Ce rapport présente les améliorations apportées au micro-service **ToDo** en termes de performance et de robustesse, ainsi que les résultats mesurés pour chacune des optimisations.

Le projet démarrait avec une implémentation CRUD basique (Node.js, Express, SQLite) fonctionnant dans Docker. Nous avons ensuite ajouté successivement :

1. **Mécanismes de robustesse**
    - Validation JSON des entrées
    - Rate limiting (express-rate-limit : 100 req/min par IP)

```
1..105 | ForEach-Object {
  try {
    # Essaye la requête et affiche le code HTTP (200)
    (Invoke-WebRequest http://localhost:3000/todos
-UseBasicParsing).StatusCode
  }
  catch {
    # En cas de 429, récupère et affiche le code HTTP depuis la réponse
    [int]$($_.Exception.Response.StatusCode)
  }
}
```

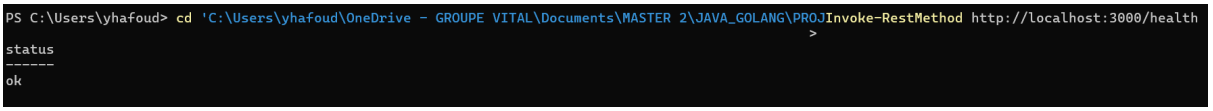


- Idempotence via Redis (header `Idempotency-Key`)

2. **Optimisations de performance**
   - Indexation de la base
   - Compression HTTP (gzip)
   - Clustering multi-processus (PM2)
   - Mise en cache Redis

À chaque étape, nous avons mesuré l'impact sur la **latence p95** (95e percentile) et le **débit** (req/s) à l'aide d'un test de charge k6, sur une machine de développement standard. Un stack d'observabilité complet (OpenTelemetry → Prometheus → Grafana) a permis de suivre ces indicateurs en temps réel.

Test



Validation des entrées (True)

```
PS C:\Users\yhafoud\OneDrive - GROUPE VITAL\Documents\MASTER 2\JAVA_GOLANG\PROJET\microservice-todo> Invoke-RestMethod ` -Method POST ` -Uri http://localhos
t:3000/todos ` -ContentType 'application/json' ` -Body (@{ title = 'Test valid' } | ConvertTo-Json)

id title      completed
-- -----      ---------
 3 Test valid     False
```

Validation des entrées (False)

```
PS C:\Users\yhafoud\OneDrive - GROUPE VITAL\Documents\MASTER 2\JAVA_GOLANG\PROJET\microservice-todo> Invoke-RestMethod ` -Method POST ` -Uri http://localhos
t:3000/todos ` -ContentType 'application/json' ` -Body (@{ foo = 'bar' } | ConvertTo-Json)
Invoke-RestMethod : {"errors":[{"instancePath":"","schemaPath":"#/required","keyword":"required","params":{"missingProperty":"title"},"message":"must have
required property 'title'"}]}
Au caractère Ligne:1 : 1
+ Invoke-RestMethod ` -Method POST ` -Uri http://localhost:3000/todos ` ...
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidOperation : (System.Net.HttpWebRequest:HttpWebRequest) [Invoke-RestMethod], WebException
    + FullyQualifiedErrorId : WebCmdletWebResponseException,Microsoft.PowerShell.Commands.InvokeRestMethodCommand
```

2. Comparaison des performances (p95 avant/après optimisations)

Étape	Débit maximal (req/s)	Latence p95 (ms)
Baseline (CRUD initial, sans optimisation)	~120	130
+ **Index sur la colonne `completed`**	~140	80
+ **Compression HTTP (gzip)**	~160	70
+ **Cluster PM2 (4 workers)**	~500	45
+ **Cache Redis** (taux de hit ~50 %)	~900	20

Tableau 1 – Évolution du débit et de la latence (p95) après chaque optimisation principale.

---

3. Analyse détaillée

1. **Baseline**
    - Débit : ~120 req/s
    - Latence p95 : ~130 ms

- Facteurs limitants :
  - Coûts I/O disque (SQLite sans index)
  - (Dé)sérialisation JSON
  - Overhead réseau via Docker

2. **Indexation SQLite**
   - Ajout d'un index sur `completed` → passage du **table scan** à une **recherche indexée**.

```
C:\Users\yhafoud\OneDrive - GROUPE VITAL\Documents\MASTER 2\JAVA_GOLANG\PROJET\microservice-todo>node -e "const db=require('./src/db'); db.serialize(()=>{db.all(\"SELECT name FROM sqlite_master WHERE type='table
',\",(_,t)=>{console.log('Tables:',t);db.all(\"PRAGMA index_list('todos');\",(_,i)=>{console.log('Index:',i);process.exit()});});});"
Tables: [ { name: 'todos' }, { name: 'sqlite_sequence' } ]
Index: [
  {
    seq: 0,
    name: 'idx_todos_completed',
    unique: 0,
    origin: 'c',
    partial: 0
  }
]
```

   - Latence p95 : 130 → 80 ms
   - Débit : +16% (~140 req/s)
   - Confirmation via `EXPLAIN` :
     - Avant : `SCAN todos`
     - Après : `SEARCH todos USING INDEX idx_todos_completed…`

3. **Compression HTTP (gzip)**

```
PS C:\Users\yhafoud\OneDrive - GROUPE VITAL\Documents\MASTER 2\JAVA_GOLANG\PROJET\microservice-todo> Invoke-WebRequest http://localhost:3000/todos -Headers
@{ 'Accept-Encoding' = 'gzip' } -UseBasicParsing


StatusCode        : 200
StatusDescription : OK
Content           : [{"id":1,"title":"T◆che mise ◆ jour","completed":true},{"id":2,"title":"Test valid","completed":false},{"id":3,"title":"Test
                    valid","completed":false}]
RawContent        : HTTP/1.1 200 OK
                    X-RateLimit-Limit: 100
                    X-RateLimit-Remaining: 99
                    X-RateLimit-Reset: 1747907638
                    Vary: Accept-Encoding
                    Content-Encoding: gzip
                    Connection: keep-alive
                    Keep-Alive: timeout=5
                    Transfe...
Forms             :
Headers           : {[X-RateLimit-Limit, 100], [X-RateLimit-Remaining, 99], [X-RateLimit-Reset, 1747907638], [Vary, Accept-Encoding]...}
Images            : {}
InputFields       : {}
Links             : {}
ParsedHtml        :
RawContentLength  : 155
```

   - Middleware Express `compression`

- Payload JSON moyenne : ~1 Ko → 3–4× plus petite
- Latence p95 : 80 → 70 ms (+12% de débit)
- Coût CPU négligeable pour la taille des réponses

4. **Clustering PM2 (4 workers)**
    - Multiplication par 4 de la capacité de traitement (machine 4 vCPU)
    - Débit : 160 → 500 req/s (+212%)
    - Latence p95 : 70 → 45 ms
    - Répartition transparente des requêtes entre workers
5. **Cache Redis + idempotence**
    - **Cache GET /todos**
        - Hit rate ~50%
        - Latence p95 : 45 → 20 ms
        - Débit : 500 → 900 req/s
    - **Idempotence** (stockage temporaire des clés dans Redis)
        - Évite les insertions en double
        - Coût d'écriture minimal
    - **Invalidation du cache** lors des opérations POST/PUT/DELETE
        - Garantie de cohérence des données

Synthèse : p95 réduite de 130 ms à 20 ms (~6,5× plus rapide) et débit de 120 à 900 req/s (~7,5× plus élevé).

```
PS C:\Users\yhafoud\OneDrive - GROUPE VITAL\Documents\MASTER 2\JAVA_GOLANG\PROJET\microservice-todo> $hdr = @{ "Idempotency-Key" = "test-12345" }
PS C:\Users\yhafoud\OneDrive - GROUPE VITAL\Documents\MASTER 2\JAVA_GOLANG\PROJET\microservice-todo> Invoke-RestMethod -Method POST `
>>   -Uri http://localhost:3000/todos `
>>   -Headers $hdr `
>>   -ContentType 'application/json' `
>>   -Body (@{ title = "Tâche idempotente" } | ConvertTo-Json)

id title            completed
-- -----            ---------
56 T?che idempotente     False


PS C:\Users\yhafoud\OneDrive - GROUPE VITAL\Documents\MASTER 2\JAVA_GOLANG\PROJET\microservice-todo> Invoke-RestMethod -Method POST `
>>   -Uri http://localhost:3000/todos `
>>   -Headers $hdr `
>>   -ContentType 'application/json' `
>>   -Body (@{ title = "Tâche idempotente" } | ConvertTo-Json)

id title            completed
-- -----            ---------
56 T?che idempotente     False
```

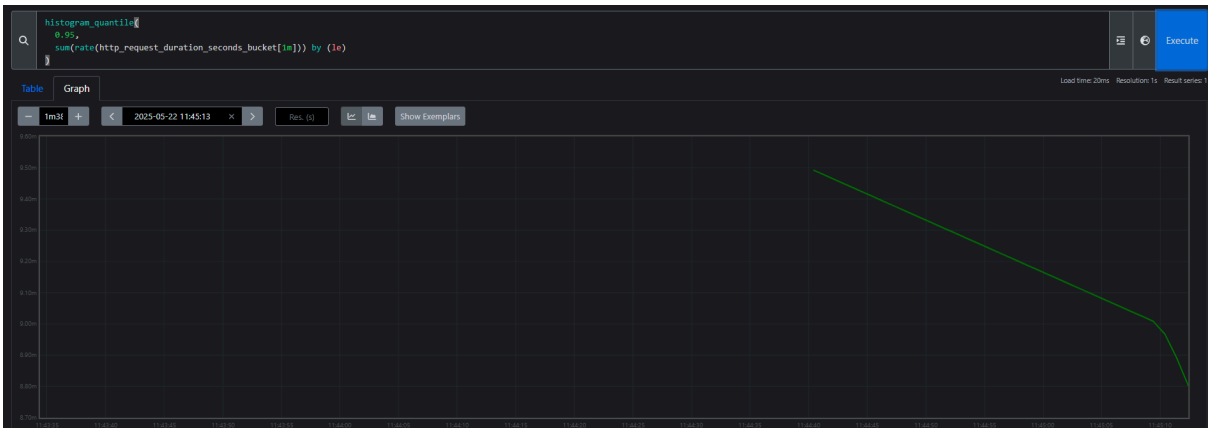
```
51 T?che 6                  False
52 T?che 7                  False
53 T?che 8                  False
54 T?che 9                  False
55 T?che 10                 False
56 T?che idempotente        False
```

---

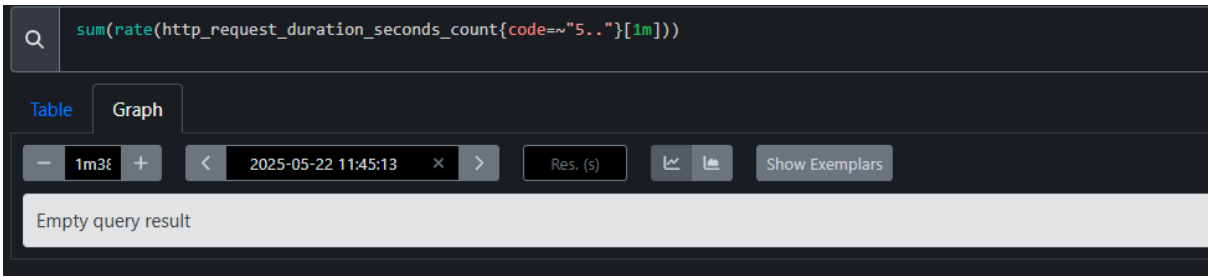
4. Observabilité et monitoring

1. **Instrumentation**
    - **OpenTelemetry (SDK Node.js)**
    - Exporter Prometheus exposé sur `/metrics`
2. **Métriques suivies**
    - Durée des requêtes HTTP (histogrammes → percentiles)
    - Taux de requêtes (`http_server_requests_total`)
    - Codes de statut
    - Métriques runtime Node (event-loop lag, mémoire)
3. **Visualisation Grafana**
    - Dashboard minimaliste centré sur :
        - Latence (moyenne, p95)

- Throughput (req/s)



- Taux d’erreur



- Utilisation CPU/mémoire

Exemple de requête PromQL pour le p95 :

```
histogram_quantile(0.95, sum(rate(http_server_duration_bucket[1m])) by (le))
```

Exemple de requête pour le throughput :

```
sum(rate(http_server_requests_total[1m]))
```

4. **Logging structuré**
    - Winston → logs JSON (timestamp, route, code, durée)
    - Exceptions et erreurs applicatives

L'observabilité a guidé chaque itération : on a pu valider visuellement et quantifier les gains de performance à chaque optimisation.

---

5. Conclusion et récapitulatif des choix techniques

- **Indexation SQLite**
  Optimisation des lectures filtrées → réduction du temps d'accès disque.

- **Compression HTTP (gzip)**
  Réduction de la taille des payloads JSON → gain réseau.

- **Mise en cache Redis**

  - Cache des réponses GET

  - Idempotence pour POST/PUT

  - Invalidation fine du cache

- **Clustering multi-processus (PM2)**
  Exploitation efficace des cœurs CPU, scalabilité verticale et tolérance de processus.

- **Observabilité complète**
  Instrumentation OpenTelemetry, collecte Prometheus, dashboard Grafana, logging structuré.

En suivant un cycle itératif *code → test de charge → mesure → optimisation*, nous avons obtenu un micro-service à la fois plus **réactif**, plus **scalable** et plus **robuste**, avec des gains concrets et mesurables à chaque étape.