Rapport Arc42 – Laboratoire 4 (LOG430 – Été 2025)

1. Introduction

Ce rapport documente les travaux réalisés dans le cadre du Laboratoire 4 du cours LOG430 – Architecture Logicielle. L'objectif principal était d'ajouter des fonctionnalités avancées à l'application de caisse multi-magasins développée dans les laboratoires précédents. Ce laboratoire visait à valider la résilience du système via des tests de charge, de tolérance aux pannes, ainsi que par l'intégration d'un mécanisme de mise en cache et d'outils de supervision (Prometheus, Grafana).

2. Objectifs

Les objectifs de ce laboratoire étaient les suivants :

- Ajouter un mécanisme de mise en cache sur des endpoints critiques.
- Mettre en place des tests de charge avec k6.
- Intégrer un outil de supervision (Prometheus + Grafana).
- Tester la tolérance aux pannes.
- Comparer plusieurs stratégies de répartition de charge (round-robin, least connections, IP hash, etc.).
- Déployer tous les composants avec Docker.
- Structurer et documenter proprement tous les tests, résultats et observations.

3. Contexte et Contraintes

Le système de caisse repose sur une architecture trois tiers : API REST en FastAPI, PostgreSQL comme base de données, et une interface graphique React. Le backend est conteneurisé et répliqué sur trois instances FastAPI, derrière un load balancer Nginx.

Contraintes spécifiques du laboratoire :

- Une seule machine virtuelle à disposition (ou locale).
- Superviser au minimum deux composants (API et load balancer).
- Conserver une documentation reproductible et claire.
- Utiliser un token statique pour l'authentification.

4. Architecture

4.1 Vue logique

- **Backend** : FastAPI avec des routeurs modulaires.
- **Services** : Couche métier séparée de l'API.
- **Base de données** : PostgreSQL (persistée dans un volume Docker).
- **Frontend** : ReactJS (dashboard).
- **Supervision** : Prometheus (scraping) + Grafana (dashboard).
- **Load balancer** : Nginx configuré avec différentes stratégies.

4.2 Vue de développement

Répertoires principaux :

```
app/
├── routers/
├── services/
├── db.py, main.py, schemas.py, securite.py
tests/
docker-compose.yml
prometheus/
grafana/
nginx/
```

4.3 Vue de déploiement

- `fastapi1`, `fastapi2`, `fastapi3` : conteneurs FastAPI
- `nginx-lb` : load balancer (Nginx)
- `nginx-exporter` : exporter Prometheus
- `prometheus`, `grafana` : supervision
- Tous les services partagent les réseaux Docker `lbnet` et `observability`

5. Mécanismes techniques

5.1 Cache

Un cache a été ajouté aux endpoints les plus critiques avec `@lru_cache` ou `functools.cache`, notamment :

- `/api/performance/global`
- `/api/rapports/ventes`
- `/api/stock`

Cela réduit considérablement la charge sur la base de données.

5.2 Authentification

Un token statique est requis dans l'en-tête `x-token`. Tous les endpoints critiques (création de vente, annulation, rapport) en dépendent.

5.3 Formatage des erreurs

Les erreurs HTTP sont renvoyées dans un format JSON clair :

```
{
  "detail": "Produit non trouvé"
}
```

5.4 Monitoring

Prometheus scrape deux cibles :

- L'API (via `/metrics`)
- Nginx (via l'exporter sur `/nginx_status`)

Un dashboard Grafana a été configuré pour afficher :

- Latence
- Taux d'erreur (4xx, 5xx)
- Trafic total
- Saturation CPU
- Nombre de connexions

6. Tests réalisés

6.1 Tests de charge

Un script `test.js` en k6 a été exécuté avec des charges progressives :

```
stages: [
  { duration: '15s', target: 10 },
  { duration: '15s', target: 30 },
  { duration: '15s', target: 60 },
  { duration: '15s', target: 100 },
  { duration: '15s', target: 0 },
]
```

Trois exécutions avec 1, 2 puis 3 instances FastAPI :

- N = 3 : `fastapi1`, `fastapi2`, `fastapi3`
- N = 2 : `fastapi1`, `fastapi2`
- N = 1 : `fastapi1`

6.2 Tolérance aux pannes

Nous avons arrêté une instance FastAPI pendant les tests :

- Continuité du service assurée via redirection.
- Le système est resté fonctionnel sans erreur critique.
- La latence a légèrement augmenté, mais aucun crash détecté.

6.3 Comparaison de stratégies de load balancing

Nginx a été configuré tour à tour avec :

- Round Robin
- Least Connections
- IP Hash
- Weighted Round Robin

Chacune a été testée avec un script de charge. Résultats :

- Round Robin : équilibré, bon par défaut.
- Least Connections : plus efficace sous forte charge.
- IP Hash : maintient l'affinité client, utile pour les sessions.
- Weighted : utile si les backends ont des capacités inégales.

7. Résultats

7.1 Résultats avant mise en cache

- Latence moyenne : élevée (~2ms à 4ms)
- Pics de saturation CPU plus fréquents
- Plus de connexions actives simultanées

7.2 Résultats après cache

- Latence divisée par 2 sur certains endpoints
- Diminution notable des requêtes SQL
- Réduction du taux d'erreurs 5xx

8. Conclusion

Le laboratoire a permis de valider :

- La mise en place de stratégies de résilience.
- L'amélioration des performances par le cache.
- L'intérêt d'outils comme Grafana/Prometheus.
- La pertinence d'un load balancer bien configuré.

Des tests plus poussés sur plusieurs VM pourraient renforcer ces résultats. Ce laboratoire constitue une base solide pour tout système distribué professionnel.