LOG430 – Architecture Logicielle (Été 2025)
Laboratoire 4 - Auto-évaluation détaillée

Nom :Maksym Pravdin Code permanent : PRAM862902010

Informations générales

- URL du depot GitHub/GitLab : https://github.com/PraMaks/LOG430_Labo_0
- Outil de test de charge utilisé (JMeter, k6, Artillery...) : k6
- Outil(s) de monitoring utilisé(s) (Prometheus, Grafana, Spring Actuator...) : Prometheus et Grafana
- Outil de load balancing (NGINX, HAProxy, etc.) : NGINX
- Mécanisme de cache implémenté (local, Redis, etc.) : Redis

1. Evaluation technique par composant

1.1 Instrumentation et observabilité initiale

Critère	Oui	Non	Commentaire / Justification
Des logs structurés ont été intégrés à l'application	✓		Une librarie des logs est ajoutée à l'appli Express.js avec 3 niveaux : info, warn et error
Un endpoint de métriques est exposé (Prometheus, Actuator, etc.)	✓		L'extension npm Prometheus bundle est ajoutée à Express.js
Les métriques de base sont correctement collectées (requêtes, erreurs, temps de réponse)	✓		Un serveur Prometheus est lancé et collecte les métriques de l'appli Express.js
Les logs permettent de tracer les requêtes de bout en bout	✓		Tous les console.log qui étaient utilisées pour logging ont été remplacées par les logs et ont été ajoutées à tous les endroits pertinents dans l'application Express.js

1.2 Monitoring et visualisation (Grafana)

Critère	Oui	Non	Commentaire / Justification
Un serveur de visualisation des métriques a-t-il été déployé et connecté à une source de collecte de données pour faciliter le suivi du système?	✓		Un serveur Grafana est utilisée pour visualiser les métriques
Un ou plusieurs dashboards ont été configurés à l'aide de templates existants ou personnalisés	✓		Un dashboard personnalisée est configurée pour Grafana (code source dans repo github pour le dashboard)
Les 4 Golden Signals[1] sont représentés visuellement	✓		Il y a plusieurs graphiques dans le dashboard pour chaque métrique recommandée dans l'énoncé
Les résultats des tests sont documentés (captures, exports de données, analyses)	✓		Les captures pertinentes de dashboard sont pris ainsi que l'état de l'application (nombre d'instances, quel type de test, etc)

1.3 Test de charge initial (baseline)

Critère	Oui	Non	Commentaire / Justification
Des scénarios de test pertinents ont été définis (lecture/écriture, simultanéité, etc.)	✓		3 templates de tests k6 sont implémentées pour tester différents cas d'utilisation de l'application
Des données réalistes ou représentatives ont été injectées	✓		Les tests peuvent être configurées lors de leur appel ce qui permet de les tester avec un nombre différent d'utilisateurs simultanées pour simuler le trafic.
Les métriques de performances ont été collectées en condition de charge	✓		Prometheus collecte les métriques d'Express et Grafana les affiche
Une analyse de la saturation ou des goulots d'étranglement a été menée	✓		Les données sont collectées et seront analysées dans le rapport.

[1] . https://sre.google/sre-book/monitoring-distributed-systems/

1.4 Load Balancing

Critère	Oui	Non	Commentaire / Justification
Un répartiteur de charge a été mis en place (NGINX, HAProxy, etc.)	✓		NGINX est utilisé comme répartiseur de charge dans le système
Plusieurs instances de service ont été déployées derrière le répartiteur	✓		L'application Express.js peut être déployée en plusieurs instances grâce au docker-compose et à NGINX
Les tests de charge ont été répétés avec load balancing activé	✓		Les mêmes tests k6 sont utilisées pour lorsqu'il y a plusieurs instances de service
Les métriques avant/après ont été comparées rigoureusement	✓		Les données sont collectées et comparées dans le rapport.
Un comportement de tolérance aux pannes a été testé (arrêt d'une instance)	✓		Le système continue de fonctionner tant qu'il y a au moins 1 instance de service

1.5 Caching applicatif

Critère	Oui	Non	Commentaire / Justification
Les endpoints critiques ont été identifiés (latence élevée ou forte fréquence)	✓		L'accès au token est identifée comme le endpoint le plus critique de l'application, car il est verifié pour presque chaque opération dans toutes les routes de l'application Express.js.
Un mécanisme de cache a été ajouté (ex : @Cacheable, Redis...)	✓		Redis est utilisé pour la mise en cache
Des règles d'expiration, d'invalidation ou de cohérence ont été définies	✓		Le cache expire après un certain nombre de temps déterminé
L'impact du cache a été mesuré en termes de latence et de charge serveur	✓		Moins de charge sur la base de données, car elle est en partie prise par le serveur Redis

2. Réflexion personnelle approfondie

1. **Instrumentation :** Quelle stratégie avez-vous adoptée pour instrumenter votre système? Quels types de métriques vous ont semblé les plus révélateurs?

   Utilisation de serveur Prometheus qui collecte les métriques à partir d'une extension sur Express.js et du serveur Grafana pour instrumenter et visualiser les métriques du système. Les types de métriques les plus révélateurs étaient le temps de réponse pour la latence ainsi que le taux d'erreur pour des réponses http 4xx ou 5xx.

2. **Monitoring :** Dans quelle mesure l'utilisation de visualisations (par exemple via Grafana) peut-elle aider à identifier des faiblesses ou des comportements inattendus?

   L'utilisation de plusieurs graphiques visuels sur un dashboard permet de plus facilement interpréter les métriques récoltées par Prometheus. De plus, l'évolution des métriques permet de voir lors des tests k6 qu'elles mesures ont des « spikes » pour comprendre comment le système réagit aux tests.

3. **Load Balancing :** Quelles sont les limites d'une simple répartition de charge dans le contexte de votre architecture? Quelles optimisations complémentaires envisageriez-vous?

   Le load balancing enlève le "bottleneck" entre le frontend et le backend, mais il ne s'applique pas pour entre le backend et la base de données. Ainsi, ceci devient le nouveau bottleneck pour la gestion de trafic dans l'architecture du système. Il serait mieux de diviser le monolithe en plusieurs services avec leurs propres bases de données pour régler ce bottleneck.

4. **Caching :** Quels ont été les bénéfices mesurés du cache? Quels risques potentiels avez-vous identifiés (cohérence, obsolescence, etc.)?

   Serveur Redis permet de faire moins de requêtes et aide beaucoup pour la gestion de tokens entre différentes instances de service. Cependant, ce token peut expirer après certain nombre de temps ce qui pourrait troubler l'expérience des utilisateurs.

5. **Approche séquentielle :** En comparant chaque étape (baseline, balancing, cache), laquelle a eu l'effet le plus significatif sur les performances? Pourquoi?

   Balancing a eu le plus d'impact plus que j'ai augmenté le nombre d'instances parce que le trafic était distribué entre celles-ci, mais l'utilisation de la mise en cache avec Redis a eu des impacts significatifs aussi, car l'accès au token était dans le cache du serveur Redis ce qui a boosté les performances aussi.

6. **Professionnalisation :** Quelles compétences ou outils appris dans ce laboratoire pensez-vous réutiliser dans un projet réel en entreprise?

   Prometheus + Grafana : C'était la première fois que j'ai eu à les utiliser et leur configuration n'était pas aussi complexe que je le pensais. La visualisation des métriques de mon système via les graphiques dans le dashboard a énormément aidé à comprendre les points faibles.

   Redis Server : La mise en cache est extrêmement utile et a beaucoup aidé avec la performance de mon système, mais il était difficile de comprendre comment faire le set-up du serveur et comment pouvoir exploiter la mise en cache par la suite.