ECOLE SUPERIEURE
D'INGENIEURS DE RENNES

Université de Rennes

Rapport d’architecture logiciel

VINCENT Théo
MARTIN Axel

Introduction

L'objectif de ce TP est de conteneuriser les différents services de notre projet de WM à l'aide de Docker. Nous y ajoutons également quelques autres services.

Notre architecture

Afin de réaliser cela, nous optons pour une architecture comme suit :

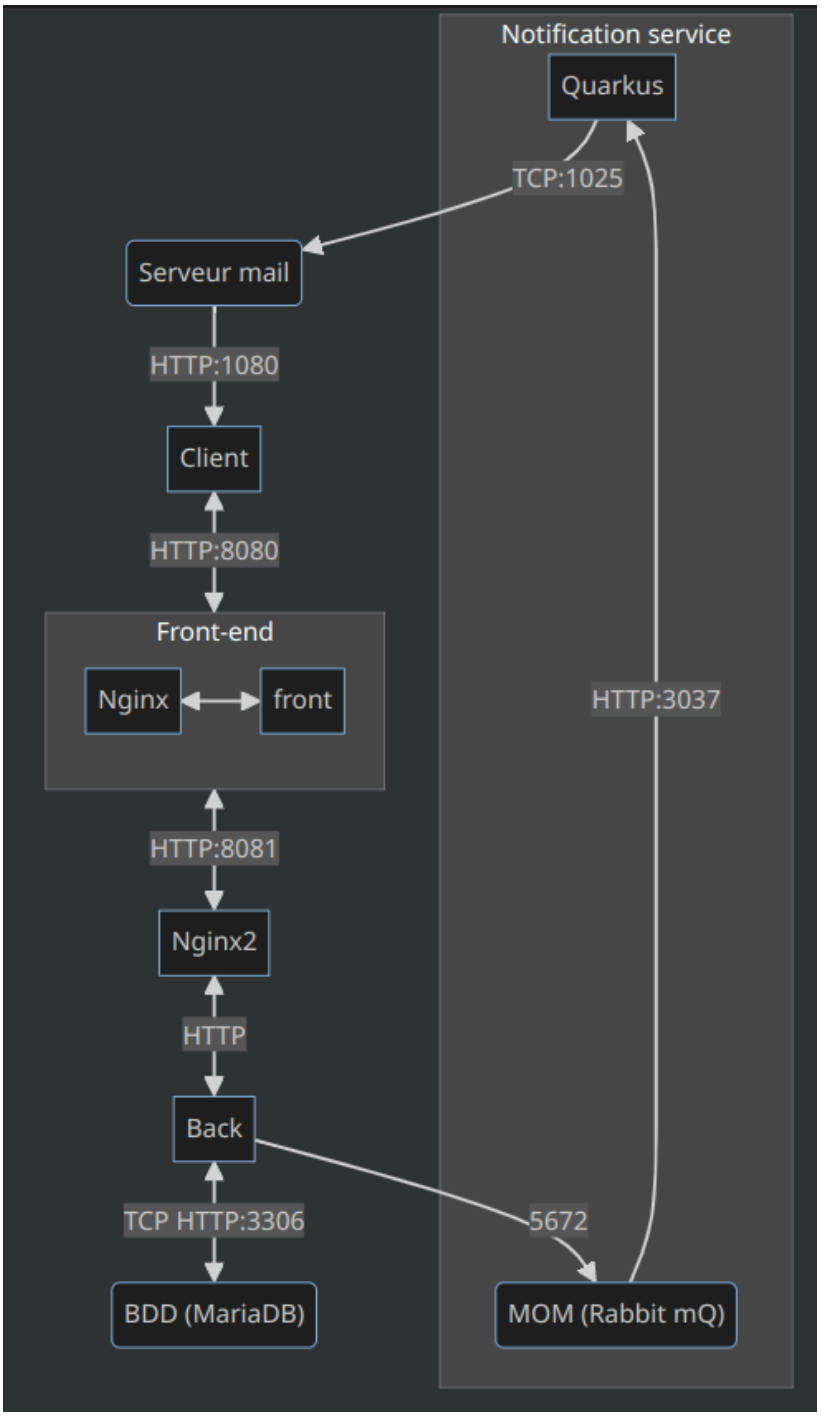


Schéma de notre architecture

Dans un premier conteneur, accessible par l'utilisateur (client) on a configuré NGINX pour servir une application web (notre frontend) en tant que proxy inverse en écoutant sur le port 8080 de notre machine locale, tout en étant accessible sur le port 80 à l'intérieur du conteneur. Il gère les fichiers HTML, JavaScript, CSS, images, etc., avec des règles spécifiques pour la mise en cache et la gestion des erreurs. Ce conteneur contient directement notre frontend en plus de NGINX. On gagne ainsi également en sécurité.

Ce conteneur communique via le port 8081 avec un autre conteneur NGINX (seul cette fois-ci), qui redirige les requêtes vers notre backend, lui-même dans un autre conteneur. Ce dernier communique lui-même via le port 3306 avec notre base de données MariaDB, également dans un conteneur. Le conteneur du backend communique aussi avec RabbitMQ pour recevoir et émettre des messages depuis le port 5672 . Cette approche basée sur les messages offre plusieurs avantages, notamment une meilleure résilience, une meilleure extensibilité et une séparation claire des responsabilités entre les différents services.

Nous mettons ensuite en place un système de notifications, composé de deux conteneurs.

En premier lieu, dans un cinquième conteneur, nous avons mis en place un message-oriented middleware (MOM) utilisant RabbitMQ. Ce MOM permet la communication asynchrone entre différents composants de notre architecture. On peut accéder à son interface depuis le port 15672.

Ensuite, dans un sixième conteneur, nous avons déployé un service développé avec Quarkus, un framework Java natif pour les applications cloud-native. Ce service est connecté au MOM (RabbitMQ) pour écouter et réagir aux événements émis par le backend. Quarkus offre des performances exceptionnelles et une faible empreinte mémoire, ce qui en fait un choix idéal pour les applications basées sur les microservices.

Un septième conteneur héberge un serveur de messagerie électronique, configuré pour gérer l'envoi de courriers électroniques en réponse à des événements spécifiques. Ce serveur de messagerie est connecté au service de notifications pour coordonner les messages qui nécessitent une communication par e-mail.

Enfin, le client pourra communiquer avec le conteneur du frontend, via le serveur web NGINX sur le port 8080, et recevra des notifications en temps réel via le service de notifications et peut recevoir des courriers électroniques pour certaines actions.

Cette architecture basée sur des conteneurs offre une grande modularité, une scalabilité facile et une gestion des ressources efficace. Chaque composant est encapsulé dans un conteneur indépendant, ce qui facilite le déploiement et la maintenance de l'ensemble du système. La communication asynchrone via RabbitMQ améliore la résilience du système, tandis que Quarkus offre des performances optimales. La séparation des services de notifications et de messagerie électronique garantit une gestion flexible des canaux de communication avec les utilisateurs. En somme, cette architecture propose une solution robuste et évolutive pour notre application web.

Des architectures alternatives ont été envisagées ; nous aurions pu séparer NGINX et le frontend, mais nous avons choisi par souci de simplicité, de sécurité et de performance de les rassembler.

Listes des services disponibles

L'architecture décrite plus haut était l'architecture que nous prévoyons de mettre en place initialement. Malheureusement, nous n'avons pas réussi à tout déployer à temps.

Les services conteneurisés sont donc les suivants :

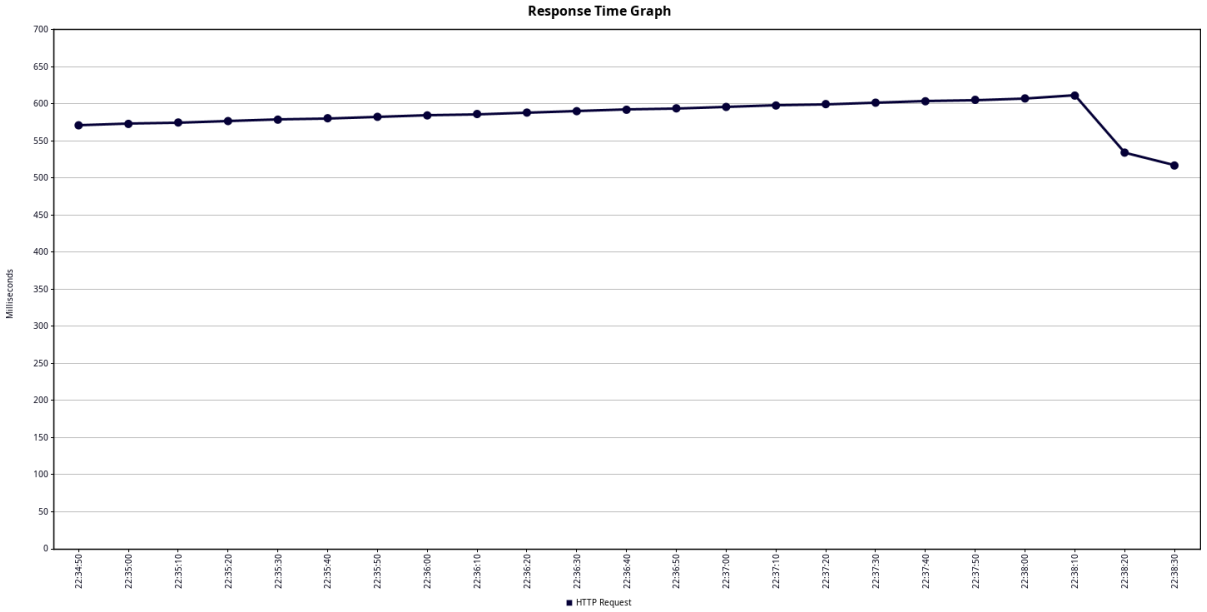
- la BDD
- le backend
- le frontend, avec NGINX.
- un conteneur avec NGINX seul
- MOM (utilisant RabbitMQ)
- un serveur mail (Quarkus n'est pas implémenté, ce qui fait que le serveur mail ne peut rien recevoir)

Chacun des services ci-dessus est implémenté mais n'a pas encore fait l'objet de tests de charge. Néanmoins, nous avons effectué des tests avec Apache JMeter (se situe hors du projet) sur notre backend en tentant de réaliser un GET à l'adresse http://localhost:8081/users, visant à récupérer l'ensemble des utilisateurs. Sur un total de 1000 appels (utilisant 1000 threads), nous observons un taux d'échec de 17,10%, un chiffre qui varie en fonction des itérations de nos tests.

Label ↓	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request	1000	539	2	1068	375.15	17.10%	513.1/sec	673.06	50.68	1343.3
TOTAL	1000	539	2	1068	375.15	17.10%	513.1/sec	673.06	50.68	1343.3

Summary report

Nous constatons également que la latence moyenne est de 539 ms, avec une valeur minimale de 2 ms et une valeur maximale de 1068 ms. Cette disparité dans la vitesse des requêtes, bien que similaires, est notable. De plus, dans l'arbre des résultats ("View Results Tree"), nous remarquons que les requêtes qui échouent apparaissent initialement, suggérant que le système peut s'adapter à la quantité de requêtes. Cependant, ceci n'est qu'une hypothèse.



Response time graph sur 10k requête

Utilisation

Afin d'utiliser notre application, il faut exécuter la commande `docker-compose up`, et aller à l'adresse `localhost:8080`.

À ce stade, aucun utilisateur n'est présent dans la base de données, car le volume vient d'être créé. Il est donc impossible de passer la page de login.
Pour pouvoir se connecter, il faut ajouter un utilisateur dans la database. On peut utiliser dans un terminal la commande `curl.exe -X POST -d "lastname=test&firstname=test&password=123&age=19" http://localhost:8081/users`, qui créera un utilisateur avec comme identifiant "1" et mot de passe "123". On peut se connecter en utilisant ces identifiants.
Lorsque nous sommes connectés, nous arrivons dans la partie "admin" de notre front.
L'intégralité des fonctionnalités implémentées sont indiquées dans le "README.md" du projet.
Maintenant si on se dirige dans la liste des users, il est possible d'en créer un nouveau à partir d'un nom, prénom, âge et d'un mot de passe. Il sera ensuite affiché dans la liste. Cette création va envoyer une notification visible à l'adresse http://localhost:15672, dans une queue. Pour se connecter, il faut utiliser "myuser" et "mypassword".
Néanmoins, Quarkus ne fonctionnant pas, ces derniers ne sont pas envoyés sous forme de mail à un serveur mail ce dernier est tout de même accessible à l'adresse http://localhost:1080.

Options de configuration de notre application

Dans notre *compose.yaml*, nous configurons les variables d'environnements suivantes :

- MYSQL_HOST: mariadb
- MYSQL_USER: root
- MYSQL_PASSWORD: secret
- MYSQL_DATABASE: mydatabase
- MYSQL_PORT: 3306
- RABBITMQ_USER : myuser
- RABBITMQ_PASS: mypassword
- RABBITMQ_HOST: rabbitmq
- RABBITMQ_QUEUE: myQueue

Chacune de ces variables d'environnement sont utilisées pour le backend et permettent d'être appelé dans l'implémentation de ce dernier. Cette méthode d'utilisation va nous permettre de modifier rapidement et cela partout dans le backend la valeur utilisée pour chacun des éléments cités ci-dessus (nom de la base de données, port…)

La configuration de NGINX se trouve dans le fichier *nginx.conf*.
Cette configuration NGINX définit un serveur web pour une application en écoutant sur le port 80. Elle utilise un seul processus principal (`worker_processes 1`) et gère les journaux d'erreurs ainsi que le suivi des processus à travers des fichiers spécifiés. L'événement `worker_connections` est limité à 1024, indiquant le nombre maximal de connexions simultanées qu'un seul processus worker peut gérer.

Le bloc `http` contient des directives globales telles que la configuration des chemins temporaires et le type de contenu par défaut. Le format des journaux est défini dans `log_format`, et les journaux d'accès et d'erreurs sont configurés pour être stockés dans des fichiers spécifiques.
La section `gzip` active la compression gzip pour certains types de fichiers, optimisant ainsi le transfert des données.
Le bloc `server` spécifie la configuration pour le serveur web, définissant le répertoire racine de l'application (*/app/browser*). Les règles de gestion des erreurs et les directives pour le cache et l'expiration des fichiers statiques sont également incluses, améliorant les performances en limitant la charge serveur.

Limiter le nombre de ports utilisés, sécurité

Nous utilisons 1 port de chaque côté (utilisateur et conteneur) par connexion nécessaire. Pour la communication entre NGINX et le frontend (au sein du même conteneur), il n'y a qu'une seule liaison, de même que pour la communication entre le frontend et le conteneur contenant NGINX seul, et entre ce dernier et le backend. Pour RabbitMQ et le serveur mail, une liaison sert à la communication entre services, l'autre sert à accéder à l'interface.

En termes de sécurité, tout d'abord nous avions seulement mis en place le conteneur avec à la fois notre frontend et NGINX. Cela assurait le passage par NGINX pour accéder au frontend (désormais inaccessible par un utilisateur en utilisant le port 4200). Cependant, nous n'assurions pas la sécurité du backend ainsi. Nous avons donc mis en place un conteneur avec *nginx-proxy*, cette fois-ci séparé des autres services, qui redirige les requêtes sur le port 8081 vers le backend. Notre backend est désormais inaccessible par un utilisateur par le port 3000, nous nous assurons donc du passage par NGINX lors de l'accès à notre backend. Nous avons dû modifier les requêtes présentes dans le frontend, qui à l'origine étaient envoyées vers http://localhost:3000, maintenant elles envoient vers http://localhost:8081.
Nous avons ainsi exploré 2 façons de mettre en place un proxy inverse NGINX.

Pour améliorer la sécurité de notre projet, nous aurions pu augmenter le nombre de réseaux afin de les isoler les uns des autres. Nous aurions pu créer un nouveau réseau pour chaque liaison, de sorte que chaque conteneur Docker soit connecté à deux autres conteneurs par deux réseaux distincts.

Conclusion

Nous avons ainsi conteneurisé notre application web. Nous avions à l'origine un front-end et un back-end (la base de données étant présente dans ce dernier). Désormais, nous avons des conteneurs avec notre front-end, notre back-end, notre base de données, mais également NGINX, MOM avec RabbitMQ, et un serveur mail. À l'issue de ce projet, nous avons donc appris à maîtriser Docker, nous avons conteneurisé nos services et nous les faisons communiquer entre eux.