Projet de Virtualisation
INFO0803

DARVILLE Killian

30 mai 2024

Table des matières

1 Introduction

Dans le cadre de notre projet en virtualisation, nous avons développé une application web en utilisant Docker. L'objectif principal de ce projet est de concevoir et déployer une solution qui exploite les avantages de la conteneurisation pour assurer une gestion efficace et flexible des applications. Notre application vise à gérer une liste de contacts, en permettant : l'ajout, la suppression, l'affichage et la sélection de contacts en fonction de critères spécifiques.

Pour atteindre cet objectif, nous avons divisé notre application en trois principaux composants, chacun étant déployé dans un conteneur Docker distinct :

- Un serveur web responsable de la visualisation des contacts.
- Une base de données stockant l'ensemble des informations sur les contacts.
- Un serveur d'API faisant office d'intermédiaire entre la base de données et le serveur web.

L'utilisation de Docker pour containeriser ces composants permet de garantir une séparation claire des différentes taches et une plus grande flexibilité dans la gestion des mises à jour. De plus, cela simplifie grandement les processus de déploiement, avec un environnement facilement reproductible.

Ce rapport détaille les différentes étapes de la conception, du développement et du déploiement de notre solution, en mettant en avant les aspects techniques liés à la virtualisation et à l'orchestration des conteneurs Docker. Nous présenterons l'architecture de l'application, les fonctionnalités implémentées, ainsi que le processus de déploiement automatisé via Docker Compose.

2 Architecture des conteneurs

Pour ce projet, nous utilisons une architecture Docker composée de trois conteneurs principaux, chacun ayant un rôle distinct et essentiel pour le fonctionnement de l'application.

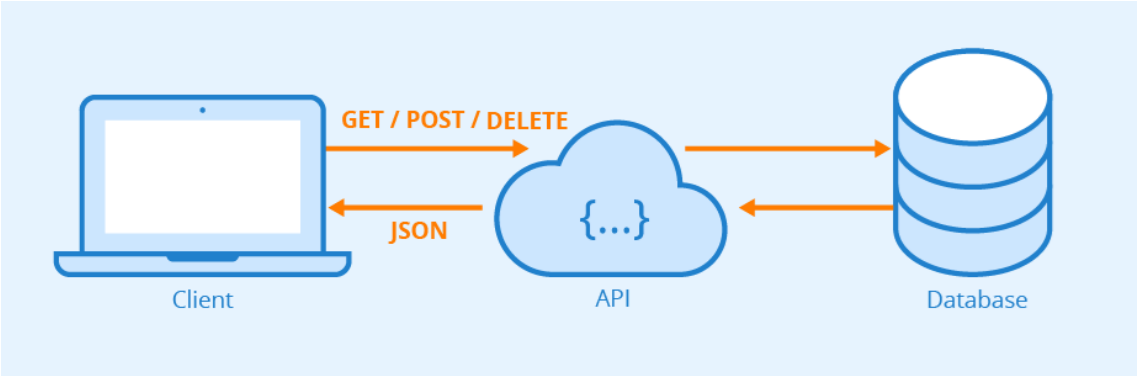

FIGURE 1 – Schéma des communications entre conteneurs

2.1 Serveur web de visualisation

Le premier conteneur est le serveur web, développé en utilisant Flask, un framework Python. Ce serveur a pour fonction de fournir l'interface utilisateur permettant aux utilisateurs de visualiser, ajouter, supprimer et filtrer les contacts. Ce conteneur communique avec le serveur d'API pour effectuer ces actions. L'utilisation de Flask pour ce conteneur permet une création rapide et simple de l'interface utilisateur. Ce conteneur expose le port 5000, par lequel les utilisateurs accèdent à l'application web grâce au lien suivant : **http ://localhost :5000**.

2.2 API

Le deuxième conteneur est le serveur d'API, également développé en Flask. Ce serveur a pour rôle de gérer les requêtes de données et de formater les données pour le serveur web. Il expose une API RESTful qui permet de réaliser des opérations CRUD (Create, Read, Update, Delete) sur la base de données des contacts.

En centralisant la logique de l'application et les opérations de gestion des données dans un serveur d'API distinct, nous obtenons une séparation claire des responsabilités entre conteneurs. Ce conteneur est accessible via le port 5001.

2.3 Base de données

Le troisième conteneur est la base de données, qui utilise MongoDB pour stocker l'ensemble des données des contacts. MongoDB a été choisi pour sa flexibilité et sa capacité à gérer simplement les différentes structures de données. Ce conteneur assure que toutes les informations sont accessibles de manière rapide et sécurisée. Le conteneur de la base de données expose le port 27017
(‘mongodb ://root :example@AdresseIP :27017/contact_db ?authSource=admin‘), permettant aux autres conteneurs d'interagir avec lui de manière efficace.

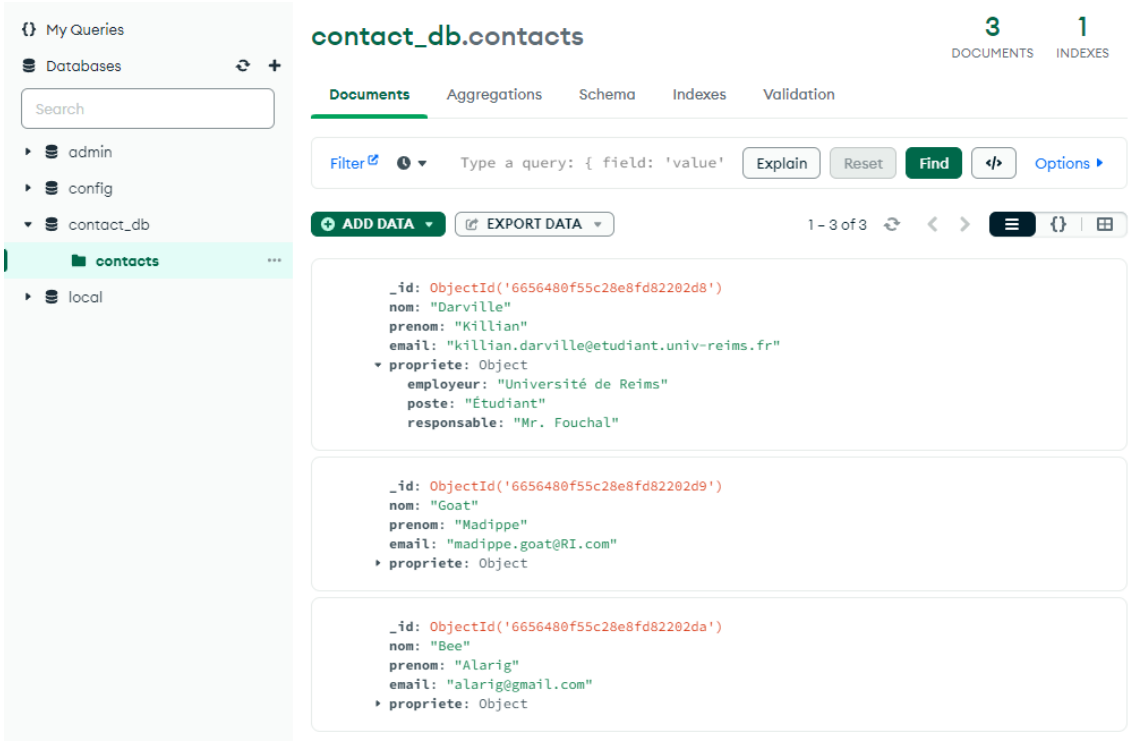

FIGURE 2 – Image de la base de données MongoDB

3 Fonctionnalités demandées

Les fonctionnalités principales de l'application sont centrées autour de la gestion des contacts, permettant aux utilisateurs d'ajouter, de supprimer, d'afficher et de sélectionner des contacts selon des critères spécifiques. Voici en détail les interactions entre les conteneurs Docker pour chaque fonctionnalité :

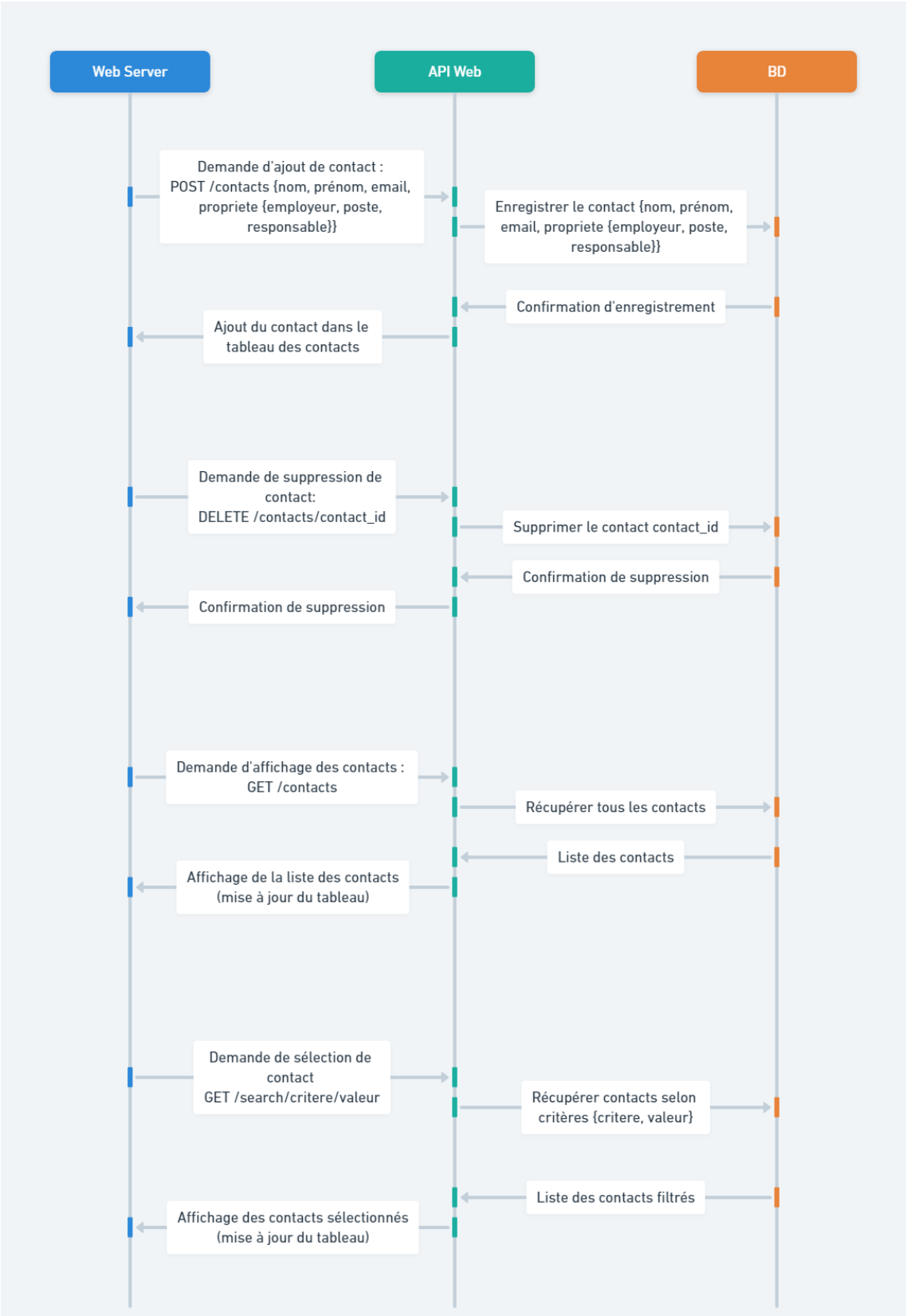

FIGURE 3 – Diagramme des échanges entre les différents conteneurs pour chaque fonctionnalité

3.1 Ajout d'un contact

L'ajout d'un nouveau contact est réalisé via une requête POST vers l'API depuis le serveur de visualisation ('POST /contacts'). Le serveur de visualisation envoie un objet JSON contenant les informations du contact telles que le nom, le prénom, l'e-mail et les propriétés. L'API reçoit cette requête et insère les données du nouveau contact dans la base MongoDB. Cette interaction permet une mise à jour en temps réel de la liste de contacts disponible dans l'application.

3.2 Suppression d'un contact

La suppression d'un contact se fait à travers une requête DELETE dirigée vers l'API, avec comme argument l'identifiant unique du contact à supprimer ('DELETE /contacts/<contact_id>'). Formée par le serveur web, cette requête permet à l'API de retirer le contact spécifié de la base de données MongoDB simplement avec son ID.

3.3 Affichage de tous les contacts

Pour afficher l'ensemble des contacts disponibles, le serveur de visualisation envoie une requête GET à l'API ('GET /contacts'). Cette requête ne nécessite aucun argument et permet à l'API de récupérer toutes les entrées de la base de données MongoDB. Une fois les données récupérées, l'API retourne la liste complète des contacts au serveur de visualisation, qui se charge ensuite de les afficher de manière structurée dans un tableau.

3.4 Sélection d'un contact à partir de critères

La fonction de sélection des contacts selon un critère spécifique est réalisée grâce à une requête GET envoyée par le serveur de visualisation à l'API ('GET /search/<critere>/<valeur>'). Les critères de recherche sont spécifiés dans l'URL de la requête GET, avec le champ à filtrer (critère) et la valeur correspondante (valeur). L'API traite cette requête en effectuant une recherche dans la base de données MongoDB pour identifier et renvoyer tous les contacts répondant aux critères spécifiés grâce à une expression régulière basée sur la valeur (Attention aux accents, ils sont pris en compte lors du filtrage). Cette fonctionnalité permet de trouver rapidement des informations spécifiques parmi les contacts enregistrés, selon les critères disponibles : 'Employeur', 'Poste' et 'Responsable'.

3.5 Interface utilisateur

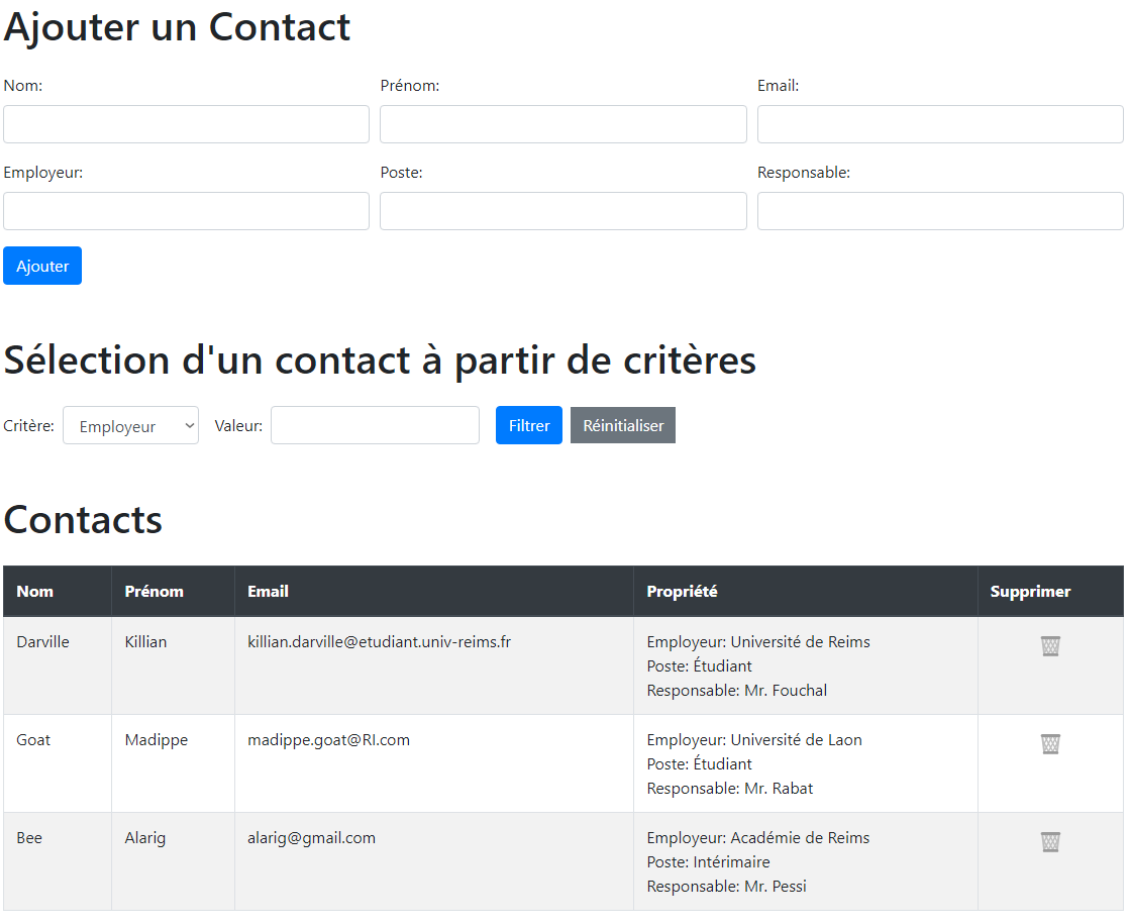

FIGURE 4 – Image de la page web

Sur la figure 4, nous pouvons voir 3 parties distinctes.

— L'utilisateur peut ajouter un contact en remplissant les différents champs du formulaire : Nom, Prénom, Email, Employeur, Poste et Responsable, puis en cliquant sur le bouton 'Ajouter'.

— L'utilisateur peut ensuite sélectionner le ou les contacts à partir d'un critère, en spécifiant une valeur pour un critère choisie entre Employeur, Poste et Responsable. Le bouton 'Filtrer' permet d'appliquer le filtre et de mettre à jour le tableau, tandis que le bouton 'Réinitialiser' permet de supprimer le filtre et affiche la totalité des contacts dans le tableau.

— Pour finir, l'utilisateur peut visualiser chaque contact ajouté dans le tableau, et supprimer les contacts en cliquant sur l'icône 'poubelle' sur la dernière colonne de chaque contact, puis en confirmant cette suppression.

4 Déploiement

L'objectif du déploiement de notre application est de permettre sa création et son lancement à partir d'un unique fichier 'docker-compose.yml'. Ce fichier définit les services nécessaires, récupère le code source depuis GitHub et initialise les conteneurs Docker correspondants.

4.1 Création des conteneurs

Pour créer et démarrer les conteneurs Docker définis dans notre application, nous utilisons l'unique commande suivante : '**docker-compose up**'. Cette commande utilise le fichier docker-compose.yml pour lancer et orchestrer les conteneurs Docker.

4.2 Volumes

Les volumes Docker jouent un rôle crucial dans la persistance des données et du code source entre les redémarrages des conteneurs. Voici comment ils sont configurés dans notre projet :

— **api_data** : Utilisé pour stocker le code de l'application cloné depuis GitHub. Ce volume est monté dans les conteneurs 'serveur' et 'api' pour permettre l'accès au code.
— **mongo_data** : Utilisé pour stocker les données persistantes de MongoDB. Cela permet que les données des contacts soient conservées même après un redémarrage des conteneurs MongoDB.

Grâce à ces volumes, lorsque la commande **docker-compose up** est exécutée à nouveau, le dépôt GitHub n'est pas cloné une seconde fois, car le code source est déjà présent dans le volume **api_data**. De même, la base de données MongoDB n'est pas recréée ni réinitialisée, car le volume **mongo_data** conserve toutes les données précédemment stockées. Cette persistance des données et du code source assure que les informations des contacts et les configurations restent intactes, permettant un redémarrage rapide et sans perte de données.

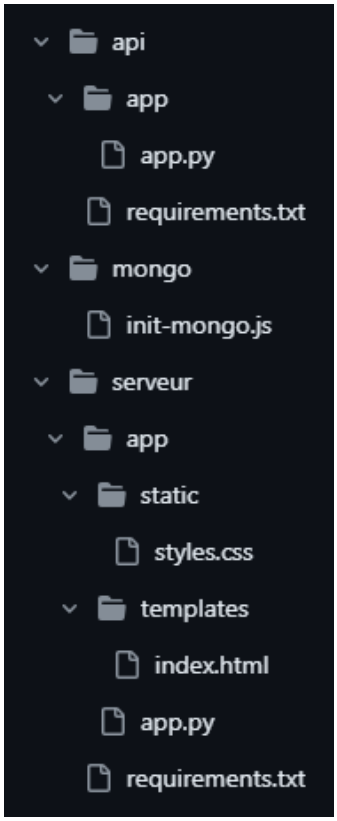

Figure 5 – Image de l'architecture du projet

5 Détails du Docker Compose

Le fichier docker-compose.yml est au coeur de notre projet, orchestrant la création et l'exécution de tous les conteneurs nécessaires. Il définit trois services principaux : serveur, api, et mongo, chacun configuré pour exécuter des tâches spécifiques dans notre architecture.

5.1 Service 'serveur'

Le service serveur est responsable de l'interface utilisateur, qui permet d'agir sur les contacts. Voici les détails de sa configuration :

- **Image** : Utilise l'image python :3.9-slim, une version plus légère de Python 3.9.
- **Volumes** : Monte le volume api_data en lecture seule (ro) qui contient le code de l'application. Cela permet au conteneur d'accéder au code source cloné depuis GitHub.
- **Entrypoint** : L'entrypoint exécute plusieurs étapes :
  - Mise à jour des paquets.
  - Attente de la fin de l'initialisation du conteneur api (indiquée par la présence du fichier api_initialized).
  - Installation des dépendances Python.
  - Démarrage du serveur Flask pour l'interface utilisateur.
- **Ports** : Expose le port 5000 pour permettre l'accès à l'application web à l'adresse suivante : **http ://localhost :5000**.
- **depends_on** : Dépend du service mongo pour s'assurer que MongoDB soit prêt avant de démarrer.

5.2 Service 'api'

Le service api est l'intermédiaire entre la base de données et le serveur web. Il expose une API RESTful pour effectuer les opérations CRUD. Voici ses spécifications :

- **Image** : Utilise l'image python :3.9-slim.
- **Volumes** : Monte le volume api_data pour accéder au code de l'application.
- **Entrypoint** : L'entrypoint exécute les étapes suivantes :
  - Mise à jour des paquets.
  - Installation de git.
  - Clone le dépôt GitHub si le répertoire n'existe pas (donc uniquement lors du premier lancement).
  - Création du fichier api_initialized pour indiquer la fin de l'initialisation.
  - Installation des dépendances Python.
  - Démarrage du serveur Flask pour l'API.
- **Ports** : Expose le port 5001 pour l'API RESTful à l'adresse suivante : **http ://localhost :5001**.
- **depends_on** : Dépend du service mongo pour s'assurer que MongoDB soit prêt avant de démarrer.

5.3 Service 'mongo'

Le service mongo gère la base de données MongoDB, stockant les informations des contacts. Voici ses détails :

- **Image** : Utilise l'image officielle de MongoDB.
- **Environment** : Configure l'utilisateur root, le mot de passe, et la base de données initiale :
  - MONGO_INITDB_ROOT_USERNAME : root
  - MONGO_INITDB_ROOT_PASSWORD : example
  - MONGO_INITDB_DATABASE : contact_db
- **Volumes** : Monte deux volumes :
  - mongo_data : Pour stocker les données persistantes de MongoDB.
  - api_data : En mode lecture seule (ro) pour accéder au script d'initialisation de la base de données.
- **Entrypoint** : L'entrypoint suit les étapes suivantes :
  - Attente que le conteneur api termine son initialisation.
  - Copie du script d'initialisation dans le répertoire approprié de MongoDB.
  - Démarrage du service MongoDB.
- **Ports** : Expose le port 27017 pour permettre l'accès à la base de données MongoDB.

Notre fichier '**docker-compose.yml**' permet que notre application web soit entièrement automatisée et très facilement déployable avec la simple commande '**docker-compose up**'.

6 Conclusion

En conclusion, notre projet de virtualisation a démontré l'efficacité de Docker et Docker Compose pour le déploiement d'applications web. En containerisant les différents composants de notre application, nous avons pu assurer une séparation des différentes taches et un processus de déploiement simple.

Le fichier '**docker-compose.yml**' joue un rôle central dans notre projet, orchestrant la création et l'exécution des conteneurs pour le serveur web, l'API et la base de données. Grâce à ce fichier unique, l'ensemble du processus de déploiement est automatisé, depuis le clonage du dépôt privé GitHub, jusqu'à l'initialisation de la base de données et le démarrage des services.

Ce projet a donc mis en valeur l'importance de la conteneurisation et de l'orchestration des conteneurs dans la gestion des applications, offrant des solutions robustes, évolutives et faciles à déployer.