Conteneurisation et Orchestration

Partie 5 : Docker Compose

Le Problème : Gérer Plusieurs Conteneurs

Une application moderne est rarement un seul conteneur. On a souvent :

- Un service web (frontend)
- Une API (backend)
- Une base de données
- Un cache...

Lancer et connecter tout cela avec des commandes `docker run` devient vite complexe et source d'erreurs.

Comment simplifier ?

```
docker run -d --name db postgres # 1. Lancer la base de données
docker run -d --name api --link db:db my-api # 2.Lancer l'API en la liant à la BDD
docker run -d -p 80:80 --link api:api my-frontend # 3. Lancer le frontend en le liant à l'API
```

La Solution : Docker Compose

Docker Compose est un outil pour définir et exécuter des applications Docker multi-conteneurs.

- **Un seul fichier de configuration** : On décrit toute l'architecture de l'application (services, réseaux, volumes) dans un fichier `compose.yml`.
- **Une seule commande** : On gère l'ensemble de l'application avec des commandes simples comme `docker compose up` et `docker compose down`.

C'est l'outil indispensable pour le développement local et les environnements de test.

Le Fichier `compose.yml`

```yaml
services:
  api: # Le service "backend"
    build: ./api
    ports:
      - "5000:5000"
    volumes:
      - ./backend:/app
    environment:
      - DATABASE_URL=postgresql://user:pass@db:5432/mydb
  db: # Le service "base de données"
    image: postgres:13
    volumes:
      - db-data:/var/lib/postgresql/data
    environment:
      - POSTGRES_USER=user
      - POSTGRES_PASSWORD=pass
      - POSTGRES_DB=mydb
volumes:
  db-data:
```

Configuration des Services (1/3)

- `image` vs `build`
  - `image: postgres:13` : Utilise une image existante depuis un registre (comme Docker Hub).
  - `build: ./api` : Construit une image localement à partir d'un `Dockerfile` situé dans le dossier `./api`.
- `ports`
  - Mappe les ports `HOTE:CONTENEUR`.
  - `ports: - "5000:5000"`
- `volumes`
  - **Volume nommé** : `db-data:/var/lib/postgresql/data` (géré par Docker, persistant).

Configuration des Services (2/3)

- `environment` et `.env`
  - Définit des variables d'environnement.
  - Pour éviter de stocker des secrets dans le `compose.yml`, on peut utiliser un fichier `.env` à la racine du projet. Compose le charge automatiquement.

```
# .env
POSTGRES_USER=user
POSTGRES_PASSWORD=secret_pass
```

- `networks`
  - Connecte les services à des réseaux spécifiques. Par défaut, Compose crée un réseau `bridge` unique pour l'application, permettant aux services de communiquer via leur nom de service (ex: `api`, `db`).

Configuration des Services (3/3)

- `depends_on` (slide suivante)
  - Gère l'**ordre de démarrage** des services.
  - **Attention** : `depends_on` attend que le conteneur soit *démarré*, pas que l'application à l'intérieur soit *prête* !
- `restart`
  - Définit la politique de redémarrage d'un service en cas d'échec.
  - `no` : Ne redémarre pas (défaut).
  - `always` : Redémarre toujours.
  - `on-failure` : Redémarre uniquement si le conteneur se termine avec un code d'erreur.
  - `unless-stopped` : Redémarre toujours, sauf si arrêté manuellement.

Gérer les Dépendances avec `healthcheck`

Pour s'assurer qu'un service est réellement prêt, on utilise un `healthcheck`.

```yaml
services:
  api:
    build: ./api
    restart: unless-stopped
    ports:
      - "5000:5000"
    depends_on:
      db:
        condition: service_healthy # Attend que le healthcheck de 'db' passe
  db:
    image: postgres:13
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U user -d mydb"]
      interval: 10s
      timeout: 5s
      retries: 5
```

La CLI Docker Compose

- `docker compose up`
  - Crée et démarre les conteneurs, réseaux et volumes.
  - `-d` : Mode détaché (arrière-plan).
  - `--build` : Force la reconstruction des images.
- `docker compose down`
  - Arrête et supprime les conteneurs et réseaux.
  - `-v` : Supprime également les volumes nommés.
- `docker compose ps` : Liste les conteneurs du projet.
- `docker compose logs` : Affiche les logs des services (`-f` pour suivre en direct).
- `docker compose exec <service> <commande>` : Exécute une commande

Limites de Docker Compose

Docker Compose est excellent pour le développement, mais il a ses limites pour la production :

- **Mono-hôte** : Il n'est pas conçu pour gérer des conteneurs sur plusieurs machines.
- **Pas de Haute Disponibilité** : Ne gère pas automatiquement le redémarrage de conteneurs sur un autre nœud en cas de panne.
- **Pas de Scalabilité Automatique** : Ne peut pas augmenter ou diminuer le nombre de conteneurs en fonction de la charge.

Pour ces besoins, on se tourne vers des orchestrateurs plus complets comme **Kubernetes**.

Conclusion sur Docker Compose

- **Docker Compose** est l'outil standard pour gérer des applications multi-conteneurs en développement.
- Le fichier `docker-compose.yml` permet de définir de manière déclarative l'ensemble de votre stack applicative.
- Il simplifie drastiquement le lancement, l'arrêt et la gestion de vos environnements locaux, tout en offrant des options avancées comme les `healthchecks` pour des démarrages robustes.

Bonus 1 : Les Dev Containers

- **Environnement de Développement dans un Conteneur** : VS Code peut s'exécuter à l'intérieur d'un conteneur Docker.
- **Configuration Partagée** : Un fichier `.devcontainer/devcontainer.json` décrit l'environnement exact : image de base, extensions VS Code à installer, variables d'environnement...
- **Reproductibilité Totale** : Chaque membre de l'équipe a exactement le même environnement de développement, peu importe son OS.

Bonus 2 : Portainer CE

Portainer Community Edition (CE) est une interface graphique web pour gérer vos environnements Docker.

- **Visualisation** : Voyez tous vos conteneurs, images, volumes et réseaux en un coup d'œil.
- **Gestion Simplifiée** : Démarrez, arrêtez, inspectez les logs et accédez à un terminal dans vos conteneurs directement depuis votre navigateur.
- **Déploiement Facile** : Déployez des applications depuis un catalogue ou en collant un fichier `compose.yml` (appelé "Stack" dans Portainer).
- **Plugin Docker Desktop** : S'intègre directement dans Docker Desktop pour une gestion encore plus simple.