Pourquoi conteneuriser son projet avec Docker

Pourquoi conteneuriser son projet avec Docker, en développement comme en production ?

La conteneurisation avec Docker est aujourd'hui une pratique incontournable dans le développement logiciel moderne. Elle permet de créer un environnement isolé, reproductible et cohérent pour une application, quel que soit le système hôte. Voici pourquoi Docker est utile **autant en développement qu'en production** :

En développement

- **Réplicabilité** : Tous les membres de l'équipe utilisent le même environnement, ce qui évite les fameux "chez moi ça marche".
- **Isolation** : Chaque projet peut tourner avec ses dépendances spécifiques sans interférer avec les autres.
- **Facilité de démarrage** : Un nouveau développeur peut lancer tout le projet avec une seule commande (via Docker Compose), sans se soucier d'installer manuellement les dépendances.

En production

- **Portabilité** : Les conteneurs Docker tournent de manière identique sur n'importe quel serveur compatible (cloud, local, cluster, etc.).
- **Déploiement simplifié** : Intégration aisée avec des outils CI/CD et orchestrateurs (comme Kubernetes).
- **Cohérence** : Même image déployée que celle utilisée pendant les tests.

En bref, Docker réduit les écarts entre les environnements, accélère les déploiements et fiabilise l'exécution des applications.

Exemple : conteneuriser un projet Python avec PostgreSQL

Prenons l'exemple d'une application Python (ex: Flask ou FastAPI) qui se connecte à une base de données PostgreSQL. Voici comment la conteneuriser avec Docker et Docker Compose.

Structure du projet

```
mon-projet/
│
├── app/
│   ├── main.py
│   └── requirements.txt
│
├── Dockerfile
└── docker-compose.yml
```

Contenu du *Dockerfile*

```dockerfile
FROM python:3.11-slim

WORKDIR /app

COPY app/requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY app/ .

CMD ["python", "main.py"]
```

Contenu du *app/main.py*

```python
import psycopg2

conn = psycopg2.connect(
    host="db",
    dbname="mydb",
    user="myuser",
    password="mypassword"
)

cur = conn.cursor()
cur.execute("SELECT version();")
version = cur.fetchone()
print("Connected to:", version)

cur.close()
conn.close()
```

Contenu du *app/requirements.txt*

```
psycopg2-binary
```

Contenu du *docker-compose.yml*

```yaml
version: '3.9'

services:
  app:
    build: .
    depends_on:
      - db
    environment:
      - PYTHONUNBUFFERED=1
    networks:
      - backend

  db:
    image: postgres:15
    environment:
      POSTGRES_DB: mydb
      POSTGRES_USER: myuser
      POSTGRES_PASSWORD: mypassword
    volumes:
      - pgdata:/var/lib/postgresql/data
    networks:
      - backend

volumes:
  pgdata:

networks:
  backend:
```

Lancement du projet

```
git:(master) docker-compose up --build
```

Et voilà : une base PostgreSQL et une application Python sont lancées ensemble, automatiquement connectées via le réseau Docker.

Conclusion

Conteneuriser un projet avec Docker permet de garantir un environnement stable, cohérent et portable. C’est un gain de temps en développement, une garantie de fiabilité en production, et un atout majeur dans les pipelines DevOps. Grâce à Docker Compose, la gestion des dépendances devient aussi simple qu’un fichier de configuration.