Webのインフラ構築で**Docker**について学んだこと

背景

もともと作成したTabelogの推薦システムはコマンドライン上で動作するものでした。しかし、より多くのユーザーに使いやすく提供するためにWebアプリ化することにしました。この際、開発環境の統一や本番環境での安定稼働を考慮し、Dockerを利用することにしました。

①**Docker**を勉強した理由

- 環境の再現性を確保
    - 開発環境と本番環境を同じコンテナで構築できるため、「自分のPCでは動くのにサーバーでは動かない」といった問題を防止したいと感じた。
- 複数サービスの一元管理
    - Webサーバー（Django/Flaskなど）、データベース（MySQL）、スクレイピング用のPython環境をDocker Composeで一括管理できることに魅力を感じた。
- デプロイの簡便化
    - コンテナ単位での起動・停止が可能で、更新やメンテナンスも容易

②システム構成

Webアプリ化にあたり、以下の構成でDockerを用いたコンテナ環境を構築しました。

- **Frontend:** Next.jsをコンテナ化
    - インストールするものの依存関係をコンテナで管理
- **Backend:** Djangoをコンテナ化
    - REST APIを提供し、フロントエンドとデータベースの橋渡し
    - 開発時の依存関係をコンテナで管理
- **Database:** MySQLをコンテナ化
    - データの永続化やバックアップをコンテナで管理

- MySQL Workbenchを用いて、データベースの管理・操作を視覚的に簡単にしました。

③**Dockerfile**の作成方法**(**主に使用するコマンド**)**

Dockerfileは、コンテナを起動する際に自動的に実行される手順を定義したファイルの作成方法の説明。ここでは、バックエンド用のDockerfileを例に説明します。

各命令の意味

- **FROM**
  - どのベースイメージを使用するかを指定
  - 例: `python:3.12-slim` → Python 3.12をベースにした軽量イメージ
- **WORKDIR**
  - コンテナ内での作業ディレクトリを指定

    →作業ディレクトリを作成することで、Docker Composeで複数コンテナを起動してもディレクトリ構造を統一できる。frontendコンテナを作成した際もapp/以下で作業ができる

  - 例: `WORKDIR /app` → 以降のコマンドは `/app` 内で実行される
- **COPY**
  - ホストのファイルやディレクトリをコンテナ内にコピー
  - 注意

    COPYできる範囲は**dockercompose.yml**の**build->context**で指定したフォルダ以下である。

    context の意味→**Docker Compose** における `context` は、**Docker** が監視・参照する範囲（イメージ作成のルートディレクトリ）を指定する設定。

    `context` 以外の場所にあるファイルは、**Dockerfile** の **COPY** では直接コピーできない**(**ここでのエラーでつまずいた**)**

    - 例1: `COPY requirements.txt ./` →`requirements.txt` を `/app` にコピー
    - 例2: `COPY . ./` → 現在のディレクトリ以下の全ファイルを `/app` にコピー
- **RUN**
    - コンテナイメージ作成時に実行するコマンド
    - 例: `RUN pip install -r requirements.txt` → 必要なパッケージをインストール
- **CMD**
    - コンテナ起動時に実行されるデフォルトコマンド
    - 例: `CMD ["python", "manage.py", "runserver", "0.0.0.0:8000"]`
    - 補足: Docker Composeでコンテナ起動時のコマンドは上書きされるが、バックエンドのコンテナ単体で動かすときもサーバーが起動するように設定できるようにする。

④Docker Compose の作成

既存のコマンドライン版 Tabelog 推薦システムを Web アプリ化するにあたり、複数のコンテナ（Frontend, Backend, Database）をまとめて管理する必要がありました。そのため、Docker Compose を作成しました以下にDocker Composeの作成方法について書きます。

Docker Compose での定義方法

build(各サービスのDFockerfileとの接続)

- `context: ./backend` → Docker がイメージを作るために参照するフォルダ

  requirments.txtをこのフォルダの監視下に置く

- `dockerfile: .devcontainer/Dockerfile` → 使用する Dockerfile を指定

container_name

- コンテナ名を指定

command

- コンテナ起動時に実行するコマンド
- ここでは Django サーバーを 0.0.0.0:8000 で起動

  (Dockerfileで指定したコンテナ起動時のコマンドを上書きできる)

volumes

- ホスト側 `./backend` のコード変更がされたときに即座にコンテナ内 `/app` コンテナに反映されるように設定

working_dir

- コンテナ内での作業ディレクトリを `/app` に設定

  (Dockerfileの設定を上書きできる)

ports

- ホストの 8000 ポートをコンテナの 8000 ポートに接続
- ブラウザからアクセス可能に

environment

- 環境変数を定義
- DB 接続情報や Django 設定を一元管理

depends_on

- このコンテナは `db` コンテナに依存していることを指定
- 依存先が先に起動するよう Docker Compose が調整

⑤コンテナの起動方法

Docker Compose を使った起動
docker-compose.ymlがあるディレクトリで以下のコマンドを実行する

```
docker-compose up -d
```

(自動的にイメージのビルドとコンテナの起動が実行する)

(**注意

Dockerfile や requirements.txt を変更した場合Dockerのイメージを

再ビルドする必要がある

```
docker-compose build --no-cache
```

)

- `-d` オプションでバックグラウンドで起動
- Frontend, Backend, Database をまとめて起動可能

他のパソコンから起動する方法

Docker と Docker Compose のインストールの必要がある

git clone <リポジトリURL>

cd <プロジェクトディレクトリ>

でダウンロード後、wslでUbuntuを開き

docker-compose up -d

をdocker-compose.ymlのディレクトリで実行。

環境変数や volumes は Compose で定義済みなので、ほぼ同じ環境を再現可能

ホスト PC のブラウザから `http://localhost:8000` にアクセスできるようになる

参考記事

https://zenn.dev/code_journey_ys/articles/c875eb92443963

←MySQLのDockerコンテナをDjangoのデータベースに設定しようとしたところ

docker CommandError: You must set settings.ALLOWED_HOSTS if DEBUG is False.

というエラーが発生した。

このエラーに対し、dockercompose.ymlで`environment:DJANGO_SETTINGS_MODULE`

で`config.settings.development`を設定

`django`の開発環境設定用のファイル`(settings/development.py)`で環境設定を上書きした後、`development.py`内で

```
DEBUG = TRUE
```

`ALLOWTED_HOSTS = [*]`を明示したことによりエラーを解消できた