What is Docker?

Docker is a platform for developing, shipping, and running applications in lightweight, portable containers. Containers package an application with all its dependencies, ensuring consistency across different environments.

---

1. Install Docker

Steps:

1. **Download Docker:**
   - Go to Docker's official website and download Docker Desktop for your operating system (Windows, macOS, or Linux).
2. **Install Docker:**
   - Follow the installation steps specific to your OS.
   - On Linux, you may need to install Docker Engine using your package manager.
3. **Verify Installation:**
   - Open a terminal or command prompt and run:

     ```
     docker --version
     ```

     This should display the Docker version installed on your system.

---

2. Docker Basics

Key Concepts:

- **Image**: A lightweight, standalone, and executable package containing everything needed to run a piece of software (code, runtime, libraries, etc.).
- **Container**: A running instance of a Docker image.
- **Dockerfile**: A script to build a Docker image.
- **Docker Hub**: A repository where Docker images are stored.

---

3. Your First Docker Commands

Run a Container:

Start a simple container using the `hello-world` image:

```
docker run hello-world
```

This command: 1. Pulls the `hello-world` image from Docker Hub (if not already present locally). 2. Creates and runs a container using the image.

List Running Containers:

```
docker ps
```

List All Containers:

```
docker ps -a
```

Stop a Running Container:

```
docker stop <container_id>
```

Remove a Container:

```
docker rm <container_id>
```

Remove an Image:

```
docker rmi <image_name>
```

---

4. Managing Docker Containers and Images

View Images:

```
docker images
```

Remove Unused Images and Containers:

```
docker system prune
```

---

6. Docker Compose

Docker Compose allows you to manage multi-container applications using a single YAML file.

Commands:

- Start the application:

  ```
  docker-compose up
  ```

- Stop the application:

  ```
  docker-compose down
  ```

Here’s a step-by-step guide to build a Docker image for a **Django application**:

---

1. Prerequisites

- Install Docker on your machine.
- Have a Django application ready. If you don’t have one, create a simple Django project:

  ```
  django-admin startproject myproject
  cd myproject
  ```

---

2. Create a `Dockerfile`

The `Dockerfile` specifies how the Docker image for your Django app will be built.

Example `Dockerfile`:

```
# Use the official Python image as a base
FROM python:3.10-slim
```

```
# Set environment variables
ENV PYTHONDONTWRITEBYTECODE=1
ENV PYTHONUNBUFFERED=1

# Set the working directory in the container
WORKDIR /app

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

# Copy the Django application code into the container
COPY . /app/

# Expose the port Django runs on
EXPOSE 8000

# Command to run the application
CMD ["python", "manage.py", "runserver", "0.0.0.0:8000"]
```

---

3. Create `requirements.txt`

List all dependencies of your Django project in a `requirements.txt` file. For example:

```
Django>=4.2
gunicorn
psycopg2-binary
```

Generate it from your virtual environment using:

```
pip freeze > requirements.txt
```

---

4. Build the Docker Image

Run the following command in the directory containing the `Dockerfile`:

```
docker build -t django-app .
```

This command: - Builds the Docker image. - Tags it with the name `django-app`.

---

5. Run the Docker Container

Start the container with:

```
docker run -p 8000:8000 django-app
```

This maps port `8000` on your machine to port `8000` in the container. Visit http://localhost:8000 to see your Django app.

---

6. Using Docker Compose (Optional)

If your Django app depends on other services (like a database), use Docker Compose to define and run multi-container applications.

Example `docker-compose.yml`:

```
version: "3.8"
```

```yaml
services:
  web:
    build: .
    command: python manage.py runserver 0.0.0.0:8000
    volumes:
      - .:/app
    ports:
      - "8000:8000"
    depends_on:
      - db

  db:
    image: postgres:13
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
      POSTGRES_DB: mydatabase
```

Steps to Run:

1. Start the services:

   ```
   docker-compose up
   ```

2. Access the Django app at http://localhost:8000.

---

7. Add `.dockerignore`

Prevent unnecessary files from being copied into the Docker image by adding a `.dockerignore` file:

```
__pycache__/
*.pyc
*.pyo
*.log
*.sqlite3
.env
```

---

8. Next Steps

- **Static Files:** Collect static files during the build by adding this to the `Dockerfile`:

  ```
  RUN python manage.py collectstatic --noinput
  ```

- **Production Server:** Use a production-ready server like Gunicorn in the `CMD`:

  ```
  CMD ["gunicorn", "myproject.wsgi:application", "--bind", "0.0.0.0:8000"]
  ```

- **Environment Variables:** Use a `.env` file to manage secrets and configurations, and load them using `django-environ` or similar.

Setup with .env

Using `.env` files in Docker Compose is a best practice for managing sensitive or environment-specific configuration variables. Here's how you can use them:

---

1. Create a `.env` File

Create a `.env` file in the root directory of your project to store environment variables.

Example `.env`:

```
POSTGRES_USER=user
POSTGRES_PASSWORD=password
POSTGRES_DB=mydatabase
DJANGO_SECRET_KEY=your-secret-key
```

---

2. Reference `.env` Variables in `docker-compose.yml`

Update the `docker-compose.yml` file to reference variables from the `.env` file.

Example `docker-compose.yml`:

```yaml
version: "3.8"

services:
  web:
    build: .
    command: python manage.py runserver 0.0.0.0:8000
    ports:
      - "8000:8000"
    depends_on:
      - db
    environment:
      - DJANGO_SECRET_KEY=${DJANGO_SECRET_KEY}

  db:
    image: postgres:13
    environment:
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
      POSTGRES_DB: ${POSTGRES_DB}
```

In this configuration: - `${VAR_NAME}` references the variable `VAR_NAME` in the `.env` file.

---

3. Verify the `.env` File is Loaded

By default, Docker Compose automatically loads the `.env` file from the current directory. No additional configuration is required if the `.env` file is in the same directory as the `docker-compose.yml` file.

If your `.env` file is in a different location or you want to use a specific file, pass it explicitly:

```
docker-compose --env-file /path/to/.env up
```

---

4. Access Variables in Django

In your Django project, use a library like django-environ to access environment variables.

Install `django-environ`:

```
pip install django-environ
```

Update `settings.py`:

```
import environ
import os

# Initialize environment variables
env = environ.Env()
environ.Env.read_env(os.path.join(BASE_DIR, '.env'))

# Example: Use environment variables in settings
SECRET_KEY = env('DJANGO_SECRET_KEY')
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': env('POSTGRES_DB'),
        'USER': env('POSTGRES_USER'),
        'PASSWORD': env('POSTGRES_PASSWORD'),
        'HOST': 'db',  # Service name in docker-compose.yml
        'PORT': 5432,
    }
}
```

---

5. Secure `.env`

- **Exclude it from Version Control:** Add `.env` to `.gitignore`:

  ```
  .env
  ```

- **Use Secret Management for Production:** For production environments, consider using secret management tools like AWS Secrets Manager, Azure Key Vault, or Docker Secrets.

---

6. Run Docker Compose

Start your services as usual:

```
docker-compose up
```

Steps to Manually Specify an Environment File

1. Create an Environment File

Create a file to store environment variables (e.g., `custom.env`).

Example `custom.env`:

```
POSTGRES_USER=user
POSTGRES_PASSWORD=password
POSTGRES_DB=mydatabase
DJANGO_SECRET_KEY=your-secret-key
```

---

2. Specify the Environment File in `docker-compose.yml`

Explicitly link the file to the service in `docker-compose.yml` using the `env_file` directive.

Updated `docker-compose.yml`:

```
version: "3.8"
```

```yaml
services:
  web:
    build: .
    command: python manage.py runserver 0.0.0.0:8000
    ports:
      - "8000:8000"
    depends_on:
      - db
    env_file:
      - custom.env

  db:
    image: postgres:13
    env_file:
      - custom.env
```

In this configuration: - `env_file` points to `custom.env` for both the `web` and `db` services.

---

3. Start Services

Run Docker Compose normally:

```
docker-compose up
```

Docker Compose will now load the variables from `custom.env` without relying on the `.env` file loading behavior.

This method: 1. Gives you control over which environment file is used. 2. Avoids ambiguity if multiple `.env` files exist.

Reference Links

DockerFile

Docker Compose

Docker CLI