Containerize a Django application

- Overview
- Create the Django project
- Create a production Dockerfile
  - Run the application
- Set up a development environment
  - Update the Dockerfile
  - Update the Compose file
  - Add the PostgreSQL driver
  - Run with Compose Watch
- Summary

- You have installed the latest version of Docker Desktop.
- You have uv installed, or you can use Docker to scaffold the project without a local Python or uv installation.

> If you're new to Docker, start with the Docker basics guide to get familiar with key concepts like images, containers, and Dockerfiles.

Overview

This guide walks you through containerizing a Django application with Docker. By the end, you will:

- Initialize a Django project using uv, either locally or inside a Docker Hardened Image container.

- Create a production-ready Dockerfile using Docker Hardened Images (DHI).
- Add a `development` stage to your Dockerfile and configure Compose Watch for automatic code syncing.

Create the Django project

You can bootstrap the project with a local uv installation, or entirely inside a container using the same DHI image the Dockerfile uses, with no local Python required.

{{< tabs >}} {{< tab name="Local (uv)" >}}

1. Initialize the project pinned to Python 3.14, then navigate into it:

   ```
   $ uv init --python 3.14 django-docker
   $ cd django-docker
   ```

2. Add Django and Gunicorn, then scaffold the Django project:

   ```
   $ uv add django gunicorn
   $ uv run django-admin startproject myapp .
   ```

{{< /tab >}} {{< tab name="Container (DHI)" >}}

The DHI dev image already has Python 3.14, so the bootstrapped project will match the Dockerfile exactly.

1. Create the project directory and navigate into it:

   ```
   $ mkdir django-docker && cd django-docker
   ```

2. Initialize the project, add dependencies, and scaffold. All in one container run:

   ```
   $ docker run --rm -v $PWD:$PWD -w $PWD \
     -e UV_LINK_MODE=copy \
     dhi.io/python:3.14-alpine3.23-dev \
     sh -c "pip install --quiet --root-user-action=ignore uv && uv init --name django-docker --python 3.14 . && uv add django gunicorn && uv run django-admin startproject myapp ."
   ```

   > The previous command uses Mac/Linux shell syntax. On Windows, adjust the path: PowerShell uses `${PWD}`, Command Prompt uses `%cd%`, Git Bash requires `MSYS_NO_PATHCONV=1`

> with `$(pwd -W)`.

{{< /tab >}} {{< /tabs >}}

Your directory should now contain the following files:

```
├── .python-version
├── main.py
├── manage.py
├── myapp/
│   ├── __init__.py
│   ├── asgi.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
├── pyproject.toml
├── uv.lock
└── README.md
```

Create a production Dockerfile

Docker Hardened Images are production-ready base images maintained by Docker that minimize attack surface. For more details, see Docker Hardened Images.

1. Sign in to the DHI registry:

   ```
   $ docker login dhi.io
   ```

2. Create a `.dockerignore` file to exclude local artifacts from the build context:

   ```
   .venv/
   __pycache__/
   *.pyc
   .git/
   ```

3. Create a `Dockerfile` with the following contents:

   ```
   # syntax=docker/dockerfile:1
   ```

```
# Build stage: the -dev image includes tools needed to install packages.
FROM dhi.io/python:3.14-alpine3.23-dev AS builder

# Prevent Python from writing .pyc files to disk.
ENV PYTHONDONTWRITEBYTECODE=1
# Prevent Python from buffering stdout/stderr so logs appear immediately.
ENV PYTHONUNBUFFERED=1

RUN pip install --quiet --root-user-action=ignore uv
# Use copy mode since the cache and build filesystem are on different volumes.
ENV UV_LINK_MODE=copy

WORKDIR /app

# Install dependencies into a virtual environment using cache and bind mounts
# so neither uv nor the lock files need to be copied into the image.
RUN --mount=type=cache,target=/root/.cache/uv \
    --mount=type=bind,source=uv.lock,target=uv.lock \
    --mount=type=bind,source=pyproject.toml,target=pyproject.toml \
    uv sync --frozen --no-install-project

# Runtime stage: minimal DHI image with no shell or package manager,
# already runs as the nonroot user.
FROM dhi.io/python:3.14-alpine3.23

# Prevent Python from buffering stdout/stderr so logs appear immediately.
ENV PYTHONUNBUFFERED=1
# Activate the virtual environment copied from the build stage.
ENV PATH="/app/.venv/bin:$PATH"

WORKDIR /app

# Copy the pre-built virtual environment and application source code.
COPY --from=builder /app/.venv /app/.venv
COPY . .

EXPOSE 8000

# Run Gunicorn as the production WSGI server.
```

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

4. Create a `compose.yaml` file:

```
services:
  web:
    build: .
    ports:
      - "8000:8000"
```

Run the application

From the `django-docker` directory, run:

```
$ docker compose up --build
```

Open a browser and navigate to http://localhost:8000. You should see the Django welcome page.

Press `ctrl`+`c` to stop the application.

Set up a development environment

The production setup uses Gunicorn and requires a full image rebuild to pick up code changes. For development, you can add a `development` stage to your Dockerfile that uses Django's built-in server, and configure Compose Watch to automatically sync code changes into the running container without a rebuild.

Update the Dockerfile

Replace your `Dockerfile` with a multi-stage version that adds a `development` stage alongside `production`:

```
# syntax=docker/dockerfile:1
```

```
# Build stage: the -dev image includes tools needed to install packages.
FROM dhi.io/python:3.14-alpine3.23-dev AS builder

# Prevent Python from writing .pyc files to disk.
ENV PYTHONDONTWRITEBYTECODE=1
# Prevent Python from buffering stdout/stderr so logs appear immediately.
ENV PYTHONUNBUFFERED=1

RUN pip install --quiet --root-user-action=ignore uv
# Use copy mode since the cache and build filesystem are on different volumes.
ENV UV_LINK_MODE=copy

WORKDIR /app

# Install dependencies into a virtual environment using cache and bind mounts
# so neither uv nor the lock files need to be copied into the image.
RUN --mount=type=cache,target=/root/.cache/uv \
    --mount=type=bind,source=uv.lock,target=uv.lock \
    --mount=type=bind,source=pyproject.toml,target=pyproject.toml \
    uv sync --frozen --no-install-project

# The development stage inherits the -dev image and virtual environment from
# the builder. Django's built-in server reloads when Compose Watch syncs files.
FROM builder AS development

ENV PATH="/app/.venv/bin:$PATH"

COPY . .
EXPOSE 8000
CMD ["python", "manage.py", "runserver", "0.0.0.0:8000"]

# The production stage uses the minimal runtime image, which has no shell,
# no package manager, and already runs as the nonroot user.
FROM dhi.io/python:3.14-alpine3.23 AS production

# Prevent Python from buffering stdout/stderr so logs appear immediately.
ENV PYTHONUNBUFFERED=1
# Activate the virtual environment copied from the build stage.
ENV PATH="/app/.venv/bin:$PATH"
```

```
WORKDIR /app

# Copy only the pre-built virtual environment and application source code.
COPY --from=builder /app/.venv /app/.venv
COPY . .

EXPOSE 8000

# Run Gunicorn as the production WSGI server.
CMD ["gunicorn", "myapp.wsgi:application", "--bind", "0.0.0.0:8000"]
```

Update the Compose file

Replace your `compose.yaml` with the following. It targets the `development` stage, adds a PostgreSQL database, and configures Compose Watch:

```
services:
  web:
    build:
      context: .
      # Build the development stage from the multi-stage Dockerfile.
      target: development
    ports:
      - "8000:8000"
    environment:
      # Enable Django's verbose debug error pages (the dev server always auto-reloads).
      - DEBUG=1
      # Database connection settings passed to Django via environment variables.
      - POSTGRES_DB=myapp
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=password
      - POSTGRES_HOST=db
      - POSTGRES_PORT=5432
    # Wait for the database to pass its healthcheck before starting the web service.
    depends_on:
      db:
        condition: service_healthy
    develop:
```

```
      watch:
        # Sync source file changes directly into the container so Django's
        # dev server can reload them without a full image rebuild.
        - action: sync
          path: .
          target: /app
          ignore:
            - __pycache__/
            - "*.pyc"
            - .git/
            - .venv/
        # Rebuild the image when dependencies change.
        - action: rebuild
          path: pyproject.toml
        - action: rebuild
          path: uv.lock
  db:
    image: dhi.io/postgres:18
    restart: always
    volumes:
      # Persist database data across container restarts.
      - db-data:/var/lib/postgresql
    environment:
      - POSTGRES_DB=myapp
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=password
    # Expose the port only to other services on the Compose network,
    # not to the host machine.
    expose:
      - 5432
    # Only report healthy once PostgreSQL is ready to accept connections,
    # so the web service doesn't start before the database is available.
    healthcheck:
      test: ["CMD", "pg_isready"]
      interval: 10s
      timeout: 5s
      retries: 5
volumes:
  db-data:
```

The `sync` action pushes file changes directly into the running container so Django's dev server reloads them automatically. A change to `pyproject.toml` or `uv.lock` triggers a full image rebuild instead.

> To learn more about Compose Watch, see Use Compose Watch.

Add the PostgreSQL driver

Add the `psycopg` adapter to your project:

{{< tabs >}} {{< tab name="Local (uv)" >}}

```
$ uv add 'psycopg[binary]'
```

{{< /tab >}} {{< tab name="Container (DHI)" >}}

```
$ docker run --rm -v $PWD:$PWD -w $PWD \
  -e UV_LINK_MODE=copy \
  dhi.io/python:3.14-alpine3.23-dev \
  sh -c "pip install --quiet --root-user-action=ignore uv && uv add 'psycopg[binary]'"
```

{{< /tab >}} {{< /tabs >}}

Then update `myapp/settings.py` to read `DEBUG` and `DATABASES` from environment variables:

```
import os

DEBUG = os.environ.get('DEBUG', '0') == '1'

DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.postgresql",
        "NAME": os.environ.get("POSTGRES_DB", "myapp"),
        "USER": os.environ.get("POSTGRES_USER", "postgres"),
        "PASSWORD": os.environ.get("POSTGRES_PASSWORD", "password"),
        "HOST": os.environ.get("POSTGRES_HOST", "localhost"),
        "PORT": os.environ.get("POSTGRES_PORT", "5432"),
    }
}
```

Run with Compose Watch

Start the development stack:

```
$ docker compose watch
```

Open a browser and navigate to http://localhost:8000.

Try editing a file, for example add a view to `myapp/views.py`. Compose Watch syncs the change into the container and Django's dev server reloads automatically. If you update `pyproject.toml` or `uv.lock`, Compose Watch triggers a full image rebuild.

Press `ctrl` + `c` to stop.

Summary

In this guide, you:

- Bootstrapped a Django project using uv, with options for both local and containerized setup.
- Created a production-ready Dockerfile using Docker Hardened Images and uv for dependency management.
- Added a `development` stage to the `Dockerfile` and configured Compose Watch for fast iterative development with a PostgreSQL database.

Related information:

- Dockerfile reference
- Compose file reference
- Use Compose Watch
- Docker Hardened Images
- Multi-stage builds
- uv documentation

□□□ #2
□ □□□ □□□ 2026-10-31 18:45:17 UTC
□ □□□ □□□ 2026-10-31 18:45:19 UTC