In this article, we'll dig into three key topics related to Docker and Jenkins pipeline:

1. First, How to structure a DockerFile 📖
2. Second, Taking advantage of multi-stage builds in Docker for better integration with a Jenkins pipeline. 📚
3. Finally, Try to understand the Docker caching mechanism 🗂

Sounds interesting so far ? Let's jump into it. 🐣🐥

A [DockerFile](https://docs.docker.com/engine/reference/builder/#dockerfile-reference) ✍️ actually contains the instructions that docker will read to build the image. You can say, it is a document with instructional commands.

First take a look over this poorly designed DockerFile below:

# Use an official Python runtime as a base image

FROM python:3.9

# Set the working directory in the container

WORKDIR /app

# Copy the requirements files into the container

COPY requirements.txt dev-requirements.txt /app/

# Install both production and development packages

RUN pip install --no-cache-dir -r requirements.txt

RUN pip install --no-cache-dir -r dev-requirements.txt

# Collect static files

RUN python3 manage.py collectstatic --noinput

# Running migrations

RUN python3 manage.py migrate

# Copy the current directory contents into the container

COPY . /app/

# Run the Django app

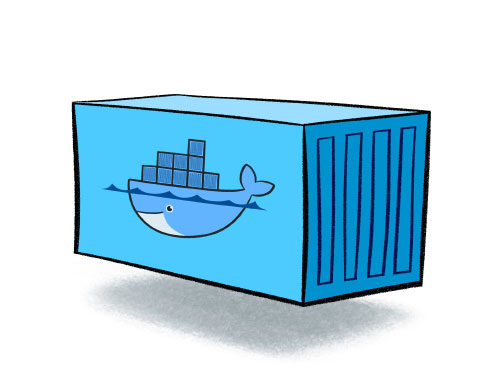
CMD ["gunicorn", "myproject.wsgi:application", "--bind", "0.0.0.0:8000"]

In the above example, the resulting image is far from ideal for production deployment. It includes development packages, dev-requirements.txt, that are unnecessary for a production environment, contributing to image bloat. Additionally, the command

COPY . /app/

make worse the image size by copying everything from the current directory into the container also the command like collectstatic and some other commands like database migrations above will adding a additional time complexities in building process.

These inefficiencies not only slow down the build process but also result in larger image sizes. This complicates the use of this DockerFile in CI/CD pipelines and leads to slower deployment due to longer image pull times. Not cool 😩!!!



A Big Container For Everything

Now let's try to restructure the above DockerFile 📖✍🏻 and try to put a closer look on the different stages below:

# Base stage

FROM python:3.9 as base

# Set the working directory in the container

WORKDIR /app

# Copy the requirements files into the container and install them.

COPY requirements.txt /app/

RUN pip install --no-cache-dir -r requirements.txt

# Build Stage

FROM base as build

# Copy the current directory contents into the container

COPY . /app/

# Collect static files

RUN python3 manage.py collectstatic --noinput

# Dev stage

FROM base as dev

# Copy the dev requirements into the container and install them.

COPY dev-requirements.txt /app/

RUN pip install --no-cache-dir -r dev-requirements.txt

# Deploy stage

FROM base as deploy

COPY --from=build /app/ /app/

# Run the Django app

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

We've broken down our DockerFile into several stages:

* In the **Base Stage**, we utilize the official python3.9 image to set up our primary environment. We first bring in the requirements.txt file and install the regular dependencies.

-no-cache-dir is actually for the pip command. It mean, pip should not include the cache directory for installing the packages but fetch them freshly from the internet to make the image size further smaller. In short terms, pip will not store downloaded packages in its cache directory.

* **Build Stage**: In this step, Docker gathers all the static files. We haven't used the migration command, but we do use the "COPY . /app/" command to tell Docker to add everything from our repository to the image but If we want to be more selective, we can specify what to include in the "COPY" command.

COPY ./src /app/src

COPY ./package.json /app/

However, if we are not sure about what our application really needs to run properly, it's safer to use a .dockerignore file to exclude unnecessary files. This also helps make the image size smaller.

**Example .dockerignore file:**

\_\_pycache\_\_/

.vscode/

.idea/

.DS\_Store

\*.py[cod]

\*.egg-info/

\*.log

\*.sqlite\*

\*.git/

\*.gitignore

\*.github

Dockerfile

\*.md

\*.env

docs/

tests/

.\*\_cache/

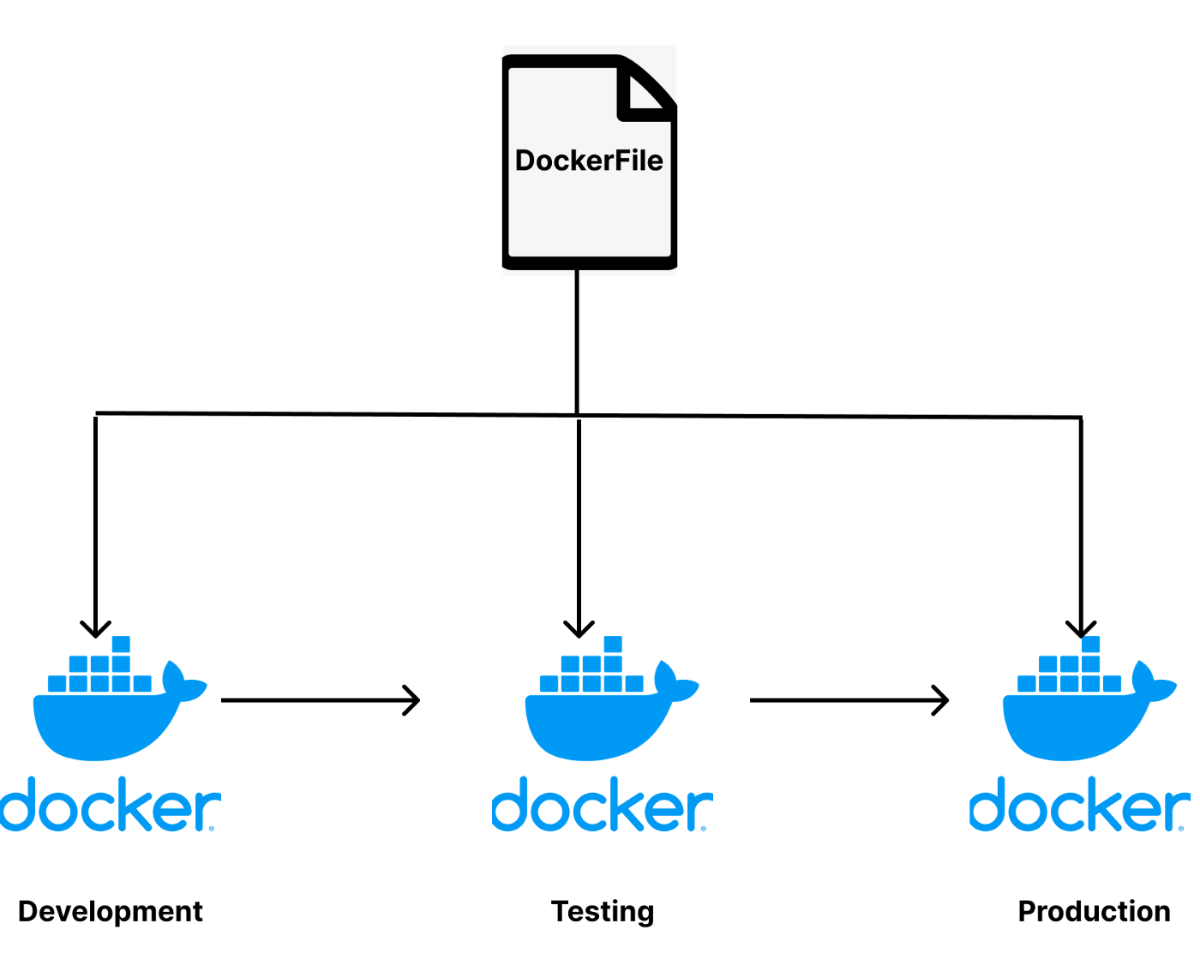
htmlcov/

node\_modules/

Migrations can be executed through scripts or commands separately once the service is started but not in DockerFile that will significantly reduce the image build processing time.

* **Dev Stage**: This stage copy and installs the development dependencies only.
* **Deploy Stage**: We pull everything from the build stage, saving time and reducing the image size for deployment.

So by using a multi-stage approach in the DockerFile, we can make the image smaller by removing unneeded development packages and commands.



This method also lets us use these optimised stages in other places, like in a Jenkins Pipeline or in docker-compose.yml. But how does it do that? 🤷 🤷

**Docker Stages in Jenkins Pipeline:**

We've defined multiple stages in our DockerFile: build , dev, and deploy. Suppose we wish to execute python unit tests prior to deploying into PRD environments. In that case, we can leverage from the dev stage from our DockerFile, which enhances the efficiency of the process.

pipeline {

agent any

stages {

stage('Construct Dev Image') {

steps {

script {

sh "docker build -t $DEV\_IMAGE --target dev ."

}

}

}

stage('Run Pytest') {

steps {

script {

sh "docker run --rm $DEV\_IMAGE pytest"

}

}

}

// Further stages can be added here

...

}

}

After running the pytest stage, we can create the deployment image using the DockerFile. In the JenkinsFile, we can specify which stage from the DockerFile to use by using the --target option, depending on what we need for different stages.

pipeline {

agent any

BASE\_IMAGE="base-image"

DEV\_IMAGE = "dev-image"

PROD\_IMAGE = "prod-image"

stages {

stage('Build Base Image') {

sh "docker build -t $BASE\_IMAGE --target base ."

}

stage('Construct Dev Image') {

sh "docker build -t $DEV\_IMAGE --target dev ."

}

stage('Run Pytest') {

sh "docker run --rm $DEV\_IMAGE pytest"

}

stage('Construct PROD Image') {

sh "docker build -t $PROD\_IMAGE --target deploy ."

}

stage('Push Image') {

steps {

script {

docker push artifactory-abc.com/repo/$PROD\_IMAGE:latest

}

}

}

}

}

**Docker Stage in docker compose:**

We have the opportunity to define the development stage within the DockerFile when we are working with local Docker containers. By doing this, we can build the image specifically for the development environment. [What is Docker Compose](https://docs.docker.com/compose/)

For example:

services:

web:

build:

context: .

target: dev. ## Referring to dev stage here.

image: web-dev-img

**Docker Caching Mechanism:**

Docker provide a very good caching layers mechanism in multi stage build environment that help to speed up the build process.

In JenkinsFile above we have:

* **Build Base Image**: During this stage, Docker builds the base image with production dependencies. Docker will cache these layers. If requirements.txt has not changed, Docker can use these cached layers in the subsequent stages, which speeds up the build process.So by copying and installing the dependencies before copying the rest of the code, we ensure that Docker only invalidates the cache and reinstalls the dependencies when requirements.txt actually changes so Docker will reuse the cache layer and will not install the production requirements again in subsequent stages until the command itself or the requirements.txt is changed , which saves a significant amount of time on subsequent builds.

sh "docker build -t $BASE\_IMAGE --target base ."

* **Build Dev Image**: This stage builds on top of the base image and installs only the development dependencies. If there are no changes in either the development or regular dependencies, Docker can use the cached layers from this stage for subsequent stages.

sh "docker build -t $DEV\_IMAGE --target dev ."

* **Pytest**: This stage runs the tests. Because the image is already built previously and have in cache so it will be a quick to run test.

sh "docker run --rm $DEV\_IMAGE pytest"

* **Build Deploy Image**: Similar to the dev image, the production image will also build on top of the base image and the base layers will be reused from the cache again to speed up the build process.

sh "docker build -t $PROD\_IMAGE --target deploy ."

**Conclusion:**

In short, By using multi-stage DockerFile and a .dockerignore file helps us to make smaller and more efficient Docker images. These methods let us to separate the different jobs in the DockerFile and making it sure only the required files and programs comes up in the final image. This will not only reduce the final image size but also makes things faster, especially in a Jenkins CI/CD pipeline where speed is important. 🙌 🎊🎊🙌