Dockerfile overview

Table of contents

Dockerfile

It all starts with a Dockerfile.

Docker builds images by reading the instructions from a Dockerfile. A Dockerfile is a text file containing instructions for building your source code. The Dockerfile instruction syntax is defined by the specification reference in the Dockerfile reference.

Here are the most common types of instructions:

Instruction	Description
`FROM <image>`	Defines a base for your image.
`RUN <command>`	Executes any commands in a new layer on top of the current image and commits the result. `RUN` also has a shell form for running commands.
`WORKDIR <directory>`	Sets the working directory for any `RUN`, `CMD`, `ENTRYPOINT`, `COPY`, and `ADD` instructions that follow it in the Dockerfile.
`COPY <src> <dest>`	Copies new files or directories from `<src>` and adds them to the filesystem of the container at the path `<dest>`.
`CMD <command>`	Lets you define the default program that is run once you start the container based on this image. Each Dockerfile only has one `CMD`, and only the last `CMD` instance is respected when multiple exist.

Dockerfiles are crucial inputs for image builds and can facilitate automated, multi-layer image builds based on your unique configurations. Dockerfiles can start simple and grow with your needs to support more complex scenarios.

Filename

The default filename to use for a Dockerfile is `Dockerfile`, without a file extension. Using the default name allows you to run the `docker build` command without having to specify additional command flags.

Some projects may need distinct Dockerfiles for specific purposes. A common convention is to name these `<something>.Dockerfile`. You can specify the Dockerfile filename using the `--file` flag for the `docker build` command. Refer to the `docker build` CLI reference to learn about the `--file` flag.

> **Note**
>
> We recommend using the default (`Dockerfile`) for your project's primary Dockerfile.

Docker images

Docker images consist of layers. Each layer is the result of a build instruction in the Dockerfile. Layers are stacked sequentially, and each one is a delta representing the changes applied to the previous layer.

Example

Here's what a typical workflow for building applications with Docker looks like.

The following example code shows a small "Hello World" application written in Python, using the Flask framework.

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello World!"
```

In order to ship and deploy this application without Docker Build, you would need to make sure that:

- The required runtime dependencies are installed on the server
- The Python code gets uploaded to the server's filesystem
- The server starts your application, using the necessary parameters

The following Dockerfile creates a container image, which has all the dependencies installed and that automatically starts your application.

```
# syntax=docker/dockerfile:1
FROM ubuntu:22.04

# install app dependencies
RUN apt-get update && apt-get install -y python3 python3-pip
RUN pip install flask==3.0.*

# install app
```

```
COPY hello.py /

# final configuration
ENV FLASK_APP=hello
EXPOSE 8000
CMD ["flask", "run", "--host", "0.0.0.0", "--port", "8000"]
```

Here's a breakdown of what this Dockerfile does:

- Dockerfile syntax
- Base image
- Environment setup
- Comments
- Installing dependencies
- Copying files
- Setting environment variables
- Exposed ports
- Starting the application

Dockerfile syntax

The first line to add to a Dockerfile is a `# syntax` parser directive. While optional, this directive instructs the Docker builder what syntax to use when parsing the Dockerfile, and allows older Docker versions with BuildKit enabled to use a specific Dockerfile frontend before starting the build. Parser directives must appear before any other comment, whitespace, or Dockerfile instruction in your Dockerfile, and should be the first line in Dockerfiles.

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

Tip

> We recommend using `docker/dockerfile:1`, which always points to the latest release of the version 1 syntax. BuildKit automatically checks for updates of the syntax before building, making sure you are using the most current version.

Base image

The line following the syntax directive defines what base image to use:

```
FROM ubuntu:22.04
```

The `FROM` instruction sets your base image to the 22.04 release of Ubuntu. All instructions that follow are executed in this base image: an Ubuntu environment. The notation `ubuntu:22.04`, follows the `name:tag` standard for naming Docker images. When you build images, you use this notation to name your images. There are many public images you can leverage in your projects, by importing them into your build steps using the Dockerfile `FROM` instruction.

Docker Hub contains a large set of official images that you can use for this purpose.

Environment setup

The following line executes a build command inside the base image.

```
# install app dependencies
RUN apt-get update && apt-get install -y python3 python3-pip
```

This `RUN` instruction executes a shell in Ubuntu that updates the APT package index and installs Python tools in the container.

Comments

Note the `# install app dependencies` line. This is a comment. Comments in Dockerfiles begin with the `#` symbol. As your Dockerfile evolves, comments can be instrumental to document how your Dockerfile works for any future readers and editors of the file, including your future self.

> **Note**
>
> You might've noticed that comments are denoted using the same symbol as the syntax directive on the first line of the file. The symbol is only interpreted as a directive if the pattern matches a directive and appears at the beginning of the Dockerfile. Otherwise, it's treated as a comment.

Installing dependencies

The second `RUN` instruction installs the `flask` dependency required by the Python application.

```
RUN pip install flask==3.0.*
```

A prerequisite for this instruction is that `pip` is installed into the build container. The first `RUN` command installs `pip`, which ensures that we can use the command to install the flask web framework.

Copying files

The next instruction uses the `COPY` instruction to copy the `hello.py` file from the local build context into the root directory of our image.

```
COPY hello.py /
```

A build context is the set of files that you can access in Dockerfile instructions such as `COPY` and `ADD`.

After the `COPY` instruction, the `hello.py` file is added to the filesystem of the build container.

Setting environment variables

If your application uses environment variables, you can set environment variables in your Docker build using the `ENV` instruction.

```

```

```
ENV FLASK_APP=hello
```

This sets a Linux environment variable we'll need later. Flask, the framework used in this example, uses this variable to start the application. Without this, flask wouldn't know where to find our application to be able to run it.

Exposed ports

The `EXPOSE` instruction marks that our final image has a service listening on port `8000`.

```
EXPOSE 8000
```

This instruction isn't required, but it is a good practice and helps tools and team members understand what this application is doing.

Starting the application

Finally, `CMD` instruction sets the command that is run when the user starts a container based on this image.

```
CMD ["flask", "run", "--host", "0.0.0.0", "--port", "8000"]
```

This command starts the flask development server listening on all addresses on port `8000`. The example here uses the "exec form" version of `CMD`. It's also possible to use the "shell form":

```
CMD flask run --host 0.0.0.0 --port 8000
```

There are subtle differences between these two versions, for example in how they trap signals like `SIGTERM` and `SIGKILL`. For more information about these differences, see Shell and exec form

Building

To build a container image using the Dockerfile example from the previous section, you use the `docker build` command:

```
$ docker build -t test:latest .
```

The `-t test:latest` option specifies the name and tag of the image.

The single dot (`.`) at the end of the command sets the build context to the current directory. This means that the build expects to find the Dockerfile and the `hello.py` file in the directory where the command is invoked. If those files aren't there, the build fails.

After the image has been built, you can run the application as a container with `docker run`, specifying the image name:

```
$ docker run -p 127.0.0.1:8000:8000 test:latest
```

This publishes the container's port 8000 to `http://localhost:8000` on the Docker host.

> **Tip**
>
> To improve linting, code navigation, and vulnerability scanning of your Dockerfiles in Visual Studio Code see Docker VS Code Extension.