Build your Python image

Estimated reading time: 13 minutes

Prerequisites

Work through the orientation and setup in Get started Part 1 (/get-started/) to understand Docker concepts.

Enable BuildKit

Before we start building images, ensure you have enabled BuildKit on your machine. BuildKit allows you to build Docker images efficiently. For more information, see Building images with BuildKit (/develop/develop-images/build_enhancements/).

BuildKit is enabled by default for all users on Docker Desktop. If you have installed Docker Desktop, you don't have to manually enable BuildKit. If you are running Docker on Linux, you can enable BuildKit either by using an environment variable or by making BuildKit the default setting.

To set the BuildKit environment variable when running the `docker build` command, run:

```
$ DOCKER_BUILDKIT=1 docker build .
```

To enable docker BuildKit by default, set daemon configuration in `/etc/docker/daemon.json` feature to `true` and restart the daemon. If the `daemon.json` file doesn't exist, create new file called `daemon.json` and then add the following to the file.

```
{
  "features":{"buildkit" : true}
}
```

Restart the Docker daemon.

Overview

Now that we have a good overview of containers and the Docker platform, let's take a look at building our first image. An image includes everything needed to run an application - the code or binary, runtime, dependencies, and any other file system objects required.

To complete this tutorial, you need the following:

- Python version 3.8 or later. Download Python (https://www.python.org/downloads/)
- Docker running locally. Follow the instructions to download and install Docker (/desktop/)
- An IDE or a text editor to edit files. We recommend using Visual Studio Code (https://code.visualstudio.com/Download).

Sample application

Let's create a simple Python application using the Flask framework that we'll use as our example. Create a directory in your local machine named `python-docker` and follow the steps below to create a simple web server.

```
$ cd /path/to/python-docker
$ pip3 install Flask
$ pip3 freeze | grep Flask >> requirements.txt
$ touch app.py
```

Now, let's add some code to handle simple web requests. Open this working directory in your favorite IDE and enter the following code into the `app.py` file.

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

@app.route('/')
def hello_world():
    return 'Hello, Docker!'
```

Test the application

Let's start our application and make sure it's running properly. Open your terminal and navigate to the working directory you created.

```
$ python3 -m flask run
```

To test that the application is working properly, open a new browser and navigate to `http://localhost:5000` .

Switch back to the terminal where our server is running and you should see the following requests in the server logs. The data and timestamp will be different on your machine.

```
127.0.0.1 - - [22/Sep/2020 11:07:41] "GET / HTTP/1.1" 200 -
```

Create a Dockerfile for Python

Now that our application is running properly, let's take a look at creating a Dockerfile.

A Dockerfile is a text document that contains the instructions to assemble a Docker image. When we tell Docker to build our image by executing the `docker build` command, Docker reads these instructions, executes them, and creates a Docker image as a result.

Let's walk through the process of creating a Dockerfile for our application. In the root of your project, create a file named `Dockerfile` and open this file in your text editor.

> **What to name your Dockerfile?**
>
> 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 `Dockerfile.<something>` or `<something>.Dockerfile` . Such Dockerfiles can then be used through the `--file` (or `-f` shorthand) option on the `docker build` command. Refer to the "Specify a Dockerfile" section (/engine/reference/commandline/build/#specify-a-dockerfile--f) in the `docker build` reference to learn about the `--file` option.
>
> We recommend using the default (`Dockerfile`) for your project's primary Dockerfile, which is what we'll use for most examples in this guide.

The first line to add to a Dockerfile is a `# syntax` parser directive (/engine/reference/builder/#syntax). 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

upgrade the parser before starting the build. Parser directives (/engine/reference/builder/#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
```

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.

Next, we need to add a line in our Dockerfile that tells Docker what base image we would like to use for our application.

```
# syntax=docker/dockerfile:1

FROM python:3.8-slim-buster
```

Docker images can be inherited from other images. Therefore, instead of creating our own base image, we'll use the official Python image that already has all the tools and packages that we need to run a Python application.

> **Note**
>
> To learn more about creating your own base images, see Creating base images (/develop/develop-images/baseimages/).

To make things easier when running the rest of our commands, let's create a working directory. This instructs Docker to use this path as the default location for all subsequent commands. By doing this, we do not have to type out full file paths but can use relative paths based on the working directory.

```
WORKDIR /app
```

Usually, the very first thing you do once you've downloaded a project written in Python is to install `pip` packages. This ensures that your application has all its dependencies installed.

Before we can run `pip3 install`, we need to get our `requirements.txt` file into our image. We'll use the `COPY` command to do this. The `COPY` command takes two parameters. The first parameter tells Docker what file(s) you would like to copy into the image. The second parameter tells Docker where you want that file(s) to be copied to. We'll copy the `requirements.txt` file into our working directory `/app`.

```
COPY requirements.txt requirements.txt
```

Once we have our `requirements.txt` file inside the image, we can use the `RUN` command to execute the command `pip3 install` . This works exactly the same as if we were running `pip3 install` locally on our machine, but this time the modules are installed into the image.

```
RUN pip3 install -r requirements.txt
```

At this point, we have an image that is based on Python version 3.8 and we have installed our dependencies. The next step is to add our source code into the image. We'll use the `COPY` command just like we did with our `requirements.txt` file above.

```
COPY . .
```

This `COPY` command takes all the files located in the current directory and copies them into the image. Now, all we have to do is to tell Docker what command we want to run when our image is executed inside a container. We do this using the `CMD` command. Note that we need to make the application externally visible (i.e. from outside the container) by specifying `--host=0.0.0.0` .

```
CMD [ "python3", "-m" , "flask", "run", "--host=0.0.0.0"]
```

Here's the complete Dockerfile.

```
# syntax=docker/dockerfile:1

FROM python:3.8-slim-buster

WORKDIR /app

COPY requirements.txt requirements.txt
RUN pip3 install -r requirements.txt

COPY . .

CMD [ "python3", "-m" , "flask", "run", "--host=0.0.0.0"]
```

Directory structure

Just to recap, we created a directory in our local machine called `python-docker` and created a simple Python application using the Flask framework. We also used the `requirements.txt` file to gather our requirements, and created a Dockerfile containing the commands to build an image. The Python application directory structure would now look like:

```
python-docker
|____ app.py
|____ requirements.txt
|____ Dockerfile
```

Build an image

Now that we've created our Dockerfile, let's build our image. To do this, we use the `docker build` command. The `docker build` command builds Docker images from a Dockerfile and a "context". A build's context is the set of files located in the specified PATH or URL. The Docker build process can access any of the files located in this context.

The build command optionally takes a `--tag` flag. The tag is used to set the name of the image and an optional tag in the format `name:tag`. We'll leave off the optional `tag` for now to help simplify things. If you do not pass a tag, Docker uses "latest" as its default tag.

Let's build our first Docker image.

```
$ docker build --tag python-docker .
[+] Building 2.7s (10/10) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 203B
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load metadata for docker.io/library/python:3.8-slim-buster
 => [1/6] FROM docker.io/library/python:3.8-slim-buster
 => [internal] load build context
 => => transferring context: 953B
 => CACHED [2/6] WORKDIR /app
 => [3/6] COPY requirements.txt requirements.txt
 => [4/6] RUN pip3 install -r requirements.txt
 => [5/6] COPY . .
 => [6/6] CMD [ "python3", "-m", "flask", "run", "--host=0.0.0.0"]
 => exporting to image
 => => exporting layers
 => => writing image sha256:8cae92a8fbd6d091ce687b71b31252056944b09760438905b72662
 => => naming to docker.io/library/python-docker
```

View local images

To see a list of images we have on our local machine, we have two options. One is to use the CLI and the other is to use Docker Desktop (/desktop/dashboard/#explore-your-images). As we are currently working in the terminal let's take a look at listing images using the CLI.

To list images, simply run the `docker images` command.

```
$ docker images
REPOSITORY      TAG               IMAGE ID       CREATED         SIZE
python-docker   latest            8cae92a8fbd6   3 minutes ago   123MB
python          3.8-slim-buster   be5d294735c6   9 days ago      113MB
```

You should see at least two images listed. One for the base image `3.8-slim-buster` and the other for the image we just built `python-docker:latest`.

Tag images

As mentioned earlier, an image name is made up of slash-separated name components. Name components may contain lowercase letters, digits and separators. A separator is defined as a period, one or two underscores, or one or more dashes. A name component may not start or end with a separator.

An image is made up of a manifest and a list of layers. Do not worry too much about manifests and layers at this point other than a "tag" points to a combination of these artifacts. You can have multiple tags for an image. Let's create a second tag for the image we built and take a look at its layers.

To create a new tag for the image we've built above, run the following command.

```
$ docker tag python-docker:latest python-docker:v1.0.0
```

The `docker tag` command creates a new tag for an image. It does not create a new image. The tag points to the same image and is just another way to reference the image.

Now, run the `docker images` command to see a list of our local images.

```
$ docker images
REPOSITORY      TAG               IMAGE ID       CREATED         SIZE
python-docker   latest            8cae92a8fbd6   4 minutes ago   123MB
python-docker   v1.0.0            8cae92a8fbd6   4 minutes ago   123MB
python          3.8-slim-buster   be5d294735c6   9 days ago      113MB
```

You can see that we have two images that start with `python-docker`. We know they are the same image because if you take a look at the `IMAGE ID` column, you can see that the values are the same for the two images.

Let's remove the tag that we just created. To do this, we'll use the `rmi` command. The `rmi` command stands for remove image.

```
$ docker rmi python-docker:v1.0.0
Untagged: python-docker:v1.0.0
```

Note that the response from Docker tells us that the image has not been removed but only "untagged". You can check this by running the `docker images` command.

```
$ docker images
REPOSITORY      TAG               IMAGE ID       CREATED          SIZE
python-docker   latest            8cae92a8fbd6   6 minutes ago    123MB
python          3.8-slim-buster   be5d294735c6   9 days ago       113MB
```

Our image that was tagged with `:v1.0.0` has been removed, but we still have the `python-docker:latest` tag available on our machine.

Next steps

In this module, we took a look at setting up our example Python application that we will use for the rest of the tutorial. We also created a Dockerfile that we used to build our Docker image. Then, we took a look at tagging our images and removing images. In the next module we'll take a look at how to:

Run your image as a container (/language/python/run-containers/)

Feedback

Help us improve this topic by providing your feedback. Let us know what you think by creating an issue in the Docker Docs (https://github.com/docker/docker.github.io/issues/new?title=[Python%20docs%20feedback]) GitHub repository. Alternatively, create a PR (https://github.com/docker/docker.github.io/pulls) to suggest updates.

python (/search/?q=python), build (/search/?q=build), images (/search/?q=images), dockerfile (/search/?q=dockerfile)