Developing Python projects in local environments can get pretty challenging if more than one project is being developed at the same time. Bootstrapping a project may take time as we need to manage versions, set up dependencies and configurations for it. Before, we used to install all project requirements directly in our local environment and then focus on writing the code. But having several projects in progress in the same environment quickly becomes a problem as we may get into configuration or dependency conflicts. Moreover, when sharing a project with teammates we would need to also coordinate our environments. For this we have to define our project environment in such a way that makes it easily shareable.

A good way to do this is to create isolated development environments for each project. This can be easily done by using containers and Docker Compose to manage them.

Here we cover how to containerize a Python service/tool and the best practices for it.

*Requirements*

To easily exercise what we discuss here, we need to install a minimal set of tools required to manage containerized environments locally:

* Windows or macOS: [Install Docker Desktop](https://www.docker.com/get-started)
* Linux: Install [Docker](https://docs.docker.com/get-docker/) and then [Docker Compose](https://docs.docker.com/compose/install/)

## Containerize a Python service

We show how to do this with a simple Flask service such that we can run it standalone without needing to set up other components.

*server.py*

from flask import Flask

server = Flask(\_\_name\_\_)

@server.route("/")

def hello():

return "Hello World!"

if \_\_name\_\_ == "\_\_main\_\_":

server.run(host='0.0.0.0')

In order to run this program, we need to make sure we have all the required dependencies installed first. One way to manage dependencies is by using a package installer such as pip. For this we need to create a requirements.txt file and write the dependencies in it. An example of such a file for our simple *server.py* is the following:

*requirements.txt*

Flask==1.1.1

We have now the following structure:

app

├─── requirements.txt

└─── src

└─── server.py

We create a dedicated directory for the source code to isolate it from other configuration files. We will see later why we do this.

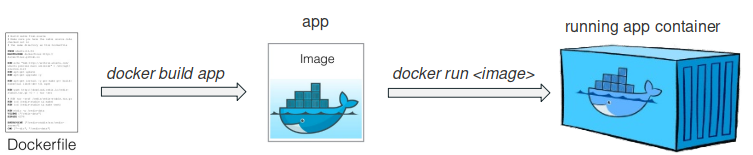
To execute our Python program, all that is left to do is to install a Python interpreter and run it.

We could run this program locally. But, this goes against the purpose of containerizing our development which is to keep a clean standard development environment that allows us to easily switch between projects with different conflicting requirements.

Let’s have a look next on how we can easily containerize this Python service.

### Dockerfile

The way to get our Python code running in a container is to pack it as a Docker image and then run a container based on it. The steps are sketched below.



To generate a Docker image we need to create a Dockerfile which contains instructions needed to build the image. The Dockerfile is then processed by the Docker builder which generates the Docker image. Then, with a simple *docker run* command, we create and run a container with the Python service.

#### Analysis of a Dockerfile

An example of a Dockerfile containing instructions for assembling a Docker image for our *hello world* Python service is the following:

*Dockerfile*

# set base image (host OS)

FROM python:3.8

# set the working directory in the container

WORKDIR /code

# copy the dependencies file to the working directory

COPY requirements.txt .

# install dependencies

RUN pip install -r requirements.txt

# copy the content of the local src directory to the working directory

COPY src/ .

# command to run on container start

CMD [ "python", "./server.py" ]

For each instruction or command from the Dockerfile, the Docker builder generates an image layer and stacks it upon the previous ones. Therefore, the Docker image resulting from the process is simply a read-only stack of different layers.

We can also observe in the output of the build command the Dockerfile instructions being executed as steps.

$ docker build -t myimage .

Sending build context to Docker daemon 6.144kB

Step 1/6 : FROM python:3.8

3.8.3-alpine: Pulling from library/python

…

Status: Downloaded newer image for python:3.8.3-alpine

---> 8ecf5a48c789

Step 2/6 : WORKDIR /code

---> Running in 9313cd5d834d

Removing intermediate container 9313cd5d834d

---> c852f099c2f9

Step 3/6 : COPY requirements.txt .

---> 2c375052ccd6

Step 4/6 : RUN pip install -r requirements.txt

---> Running in 3ee13f767d05

…

Removing intermediate container 3ee13f767d05

---> 8dd7f46dddf0

Step 5/6 : COPY ./src .

---> 6ab2d97e4aa1

Step 6/6 : CMD python server.py

---> Running in fbbbb21349be

Removing intermediate container fbbbb21349be

---> 27084556702b

Successfully built 70a92e92f3b5

Successfully tagged myimage:latest

Then, we can check the image is in the local image store:

$ docker images

REPOSITORY TAG IMAGE ID CREATED SIZE

myimage latest 70a92e92f3b5 8 seconds ago 991MB

During development, we may need to rebuild the image for our Python service multiple times and we want this to take as little time as possible. We next analyze some best practices that may help us with this.

### Development Best Practices for Dockerfiles

#### Base Image

The first instruction from the Dockerfile specifies the base image on which we add new layers for our application. The choice of the base image is pretty important as the features it ships may impact the quality of the layers built on top of it.

When possible, we should always use official images which are in general frequently updated and may have less security concerns.

The choice of a base image can impact the size of the final one. If we prefer size over other considerations, we can use some of the base images of a very small size and low overhead. These images are usually based on the *alpine* distribution and are tagged accordingly. However, for Python applications, the slim variant of the official Docker Python image works well for most cases (eg. python:3.8-slim).

#### Instruction order matters for leveraging build cache

When building an image frequently, we definitely want to use the builder cache mechanism to speed up subsequent builds. As mentioned previously, the Dockerfile instructions are executed in the order specified. For each instruction, the builder checks first its cache for an image to reuse. When a change in a layer is detected, that layer and all the ones coming after are being rebuilt.

For an efficient use of the caching mechanism , we need to place the instructions for layers that change frequently after the ones that incur less changes.

Let’s check our Dockerfile example to understand how the instruction order impacts caching. The interesting lines are the ones below.

...

# copy the dependencies file to the working directory

COPY requirements.txt .

# install dependencies

RUN pip install -r requirements.txt

# copy the content of the local src directory to the working directory

COPY src/ .

...

During development, our application’s dependencies change less frequently than the Python code. Because of this, we choose to install the dependencies in a layer preceding the code one. Therefore we copy the dependencies file and install them and then we copy the source code. This is the main reason why we isolated the source code to a dedicated directory in our project structure.

#### Multi-stage builds

Although this may not be really useful during development time, we cover it quickly as it is interesting for shipping the containerized Python application once development is done.

What we seek in using multi-stage builds is to strip the final application image of all unnecessary files and software packages and to deliver only the files needed to run our Python code. A quick example of a multi-stage Dockerfile for our previous example is the following:

# first stage

FROM python:3.8 AS builder

COPY requirements.txt .

# install dependencies to the local user directory (eg. /root/.local)

RUN pip install --user -r requirements.txt

# second unnamed stage

FROM python:3.8-slim

WORKDIR /code

# copy only the dependencies installation from the 1st stage image

COPY --from=builder /root/.local /root/.local

COPY ./src .

# update PATH environment variable

ENV PATH=/root/.local:$PATH

CMD [ "python", "./server.py" ]

Notice that we have a two stage build where we name only the first one as *builder*. We name a stage by adding an *AS <NAME>* to the *FROM* instruction and we use this name in the *COPY i*nstruction where we want to copy only the necessary files to the final image.

The result of this is a slimmer final image for our application:

$ docker images

REPOSITORY TAG IMAGE ID CREATED SIZE

myimage latest 70a92e92f3b5 2 hours ago 991MB

multistage latest e598271edefa 6 minutes ago 197MB

…

In this example we relied on the *pip’s –user* option to install dependencies to the local user directory and copy that directory to the final image. There are however other solutions available such as virtualenv or building packages as wheels and copy and install them to the final image.

### Run the container

After writing the Dockerfile and building the image from it, we can run the container with our Python service.

$ docker images

REPOSITORY TAG IMAGE ID CREATED SIZE

myimage latest 70a92e92f3b5 2 hours ago 991MB

...

$ docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

$ docker run -d -p 5000:5000 myimage

befb1477c1c7fc31e8e8bb8459fe05bcbdee2df417ae1d7c1d37f371b6fbf77f

We now containerized our *hello world* server and we can query the port mapped to localhost.

$ docker ps

CONTAINER ID IMAGE COMMAND PORTS ...

befb1477c1c7 myimage "/bin/sh -c 'python ..." 0.0.0.0:5000->5000/tcp ...

$ curl http://localhost:5000

"Hello World!"