Running the App Image - Debug and Production Modes

Running images in Docker

We'll cover the following

In the last chapter, we built our app image. Now, it's time to run that image using `docker run` command. This command has multiple options. If you type `docker run --help` in the terminal, you'll get plenty of options for this command, but the right syntax is below.

Usage: docker run [OPTIONS] IMAGE [COMMAND] [ARG...]

[OPTIONS] - any argument related to running the image

[COMMAND] - any command to override CMD if ENTRYPOINT is implemented in Dockerfile

[ARG] - arguments for container shell

Running the app image

If you remember, we previously built our image using flask_app:1.0 tag. Type `docker images` into the terminal and verify that your images exist on your system.

The output will be similar to

```
REPOSITORY                                  TAG                 IMAGE ID
            CREATED              SIZE
flask_app                                   1.0                 3edb8369bc8
3        8 minutes ago       952MB
```

The debug mode

It's time to run our app using Docker.

Type `docker run flask_app:1.0`

You'll get output something like so:

```
* Serving Flask app "app.py" (lazy loading)
* Environment: development
* Debug mode: on
* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 132-810-040
```

At this point, you won't be able to access the app from your browser.

Now, stop the app using ctrl + c and rerun using `docker run -p 5000:5000 flask_app:1.0`. There will be no change in output, but try accessing your app in the browser using **localhost:5000**.

Cheers!! You got it running.

What just happened? What makes the app accessible?

`-p 5000:5000` option in run command

> Before adding this option, the app was running in the container on the container's 5000 port. There is no way the host machine will reach into the container. However, the -p 5000:5000 option will create a pipe/tunnel on the host machine's 5000 port to the container's 5000 port.

This is called port mapping. You can create interfaces to establish communication between host and container. We will soon see how to make the host's file system accessible from the container.

Notice a few things in the output:

1.

```
```

```
Debug mode: on
```

2.

```
Environment: development
```

Since we set FLASK_ENV=development in .flaskrun file. Our app is running in debug mode.

```
<h1>Sign In</h1>
<form action="/" method="post" novalidate>
    <input type="hidden">
    <p>
        <label>Username</label> <br>
        <input type="text" id="username" name="username">
    </p>
    <p>
        <label>Password</label>
        <br>
        <input type="password" id="password" name="password">
    </p>
    <p> <input type="submit" value="Submit"> </p>
</form>
```

Debug mode off

Now, change the 'FLASK_ENV' environment variable in .flaskenv file to 'production' and rebuild the image.

`docker build -t flask_app:1.0 .`

Next, run it using `docker run -p 5000:5000 flask_run:1.0`.

You'll see the following output:

```
 * Serving Flask app "app.py"
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deploy
ment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

Notice,

```
* Environment: production
and
```

```
* Debug mode: off
```

Although it is a production environment, it is not used in production. We will see in the bonus section how to deploy Flask to a real production environment.

```
<h1>Sign In</h1>
<form action="/" method="post" novalidate>
    <input type="hidden">
    <p>
        <label>Username</label> <br>
        <input type="text" id="username" name="username">
    </p>
    <p>
        <label>Password</label>
        <br>
        <input type="password" id="password" name="password">
    </p>
    <p> <input type="submit" value="Submit"> </p>
</form>
```

What is the significance of the development mode?

While developing the app, any changes you make in files are detected by Flask and it restarts the server to reflect the changes automatically. This significantly reduces development time. You can try that out by running the app in venv for now.

You'll get something like:

```
* Detected change in '/Users/venkateshachintalwar/Documents/Online_Projects/Docker/app.py', reloading
```

Even so, this will not work with Docker for now, because there is a separation between the container's file system and the host's file system.

> When you code, you are making changes in host's file system and that's why it will detect the change in venv. To reflect the change in a container, you'll have to rebuild the image.

Rebuilding the whole image just to reflect a line of code can be annoying. How can we fix this?

In the "**Managing Data for Container**" lesson, we saw how to mount a filesystem onto the container. Let's make use of that.

Type `pwd` to get the absolute path to 'Docker' directory. Next, add one more option to the `docker run` command,

`docker run -it -v <host_absolute_path>:<folder path in container> -p 5000:5000 flask_app:1.0`

For example,

`docker run -it -v /Users/venkateshachintalwar/Documents/Online_Projects/Docker:/code -p 5000:5000 flask_app:1.0`

Now, if you run the app in dev mode, changes will be instantly reflected.

```
* Detected change in '/code/app.py', reloading
```

Daemon mode

Instead of running a container in interactive shell mode (`-it` option stands for `-i` - Interactive, `-t` TTY terminal) of the container, it can be run as a background process. Means, docker container will run in the background completely detached from the shell of the host machine.

This way, the shell or the terminal will not be locked and other commands can be executed on the same shell. To run the container in daemon mode, use `-d` option while running or creating the container from the image.

Example: `docker run -d -v <path_to_code_directory>:/code -p 5000:5000 flask_app:1.0`

This will print the container id something like this,

```
9ffc383cf193af15ff4daa744db5a58f1cdc85b9b51d09f096aa4c591dc24d6b
```

And the container will continue running in the background. The id is a full SHA256 digest of the container.

Type `docker ps` to see whether the container is running in the background or not.

```
CONTAINER ID        IMAGE                    COMMAND             CREATED             STATUS              PORTS                    NAMES
33221bd2d378        venky8283/flask_app:1.0  "flask run"         12 minutes ago      Up 12 minutes       0.0.0.0:5000->5000/tcp   eloquent_b
haskara
```

Under the Image column, you will see flask_app:1.0. This confirms that the image is started and running in the background. (For any errors refer docker

troubleshooting section.)

To get back into the container, Type `docker exec -it <container id> bash`. This will attach the shell of the container to the host terminal. `docker exec` is used to execute commands from the running container.

You can also directly attach the container running in detached mode to the shell using `docker attach <container id>`. But you will not see any output as no process in the container is printing to stdout of the container.

Once you type `ctrl + c` to get back to host shell, the container will be stopped. And this is the reason you should avoid `docker attach` command to get into the detached container.

Either use `docker exec -it <container_id> bash` command or `docker logs` command to interact with the detached containers.

That was interesting. Now, take a break. Grab a coffee. We will see more about containers in the next chapter.