Docker Containers
Creating, managing and running containers

---

1 Building a minimal Flask-Python Docker container

The simple Python app will run in a minimal container that runs on Ubuntu. The container will be based on the most recent image of Python (from DockerHub) and will install Flask and a Hello World web service.
This guide covers the most important operations regarding the creation, management and execution of containers. An overview is available at https://docs.docker.com/get-started/overview/ .

1.1 Check that Docker is installed

First check that Docker is installed with:

```
$ docker -v
```

1.2 Install Docker

If Docker isn't installed, follow the official guide:
https://docs.docker.com/engine/install/ubuntu/

1.3 Test installation

The Docker installation can be checked with:

```
$ docker run hello-world
```

If it fails because it cannot connect to the Docker daemon, start the daemon with:

```
$ sudo dockerd
```

1.4 Add current user to Docker group

To avoid having to type `sudo` for building and running containers you can do as follows:

```
$ sudo groupadd docker
$ sudo usermod -aG docker $USER
$ newgrp docker
```

The first command will probably return an info saying that the group already exists. The last command is just to make the second command have an immediate effect.

Now check again if you can run the `docker run hello-world` without `sudo`. It should work!

1.5 Play with Docker

If Docker can't be installed on your machine you can use a temporary **online Docker playground** like: `https://labs.play-with-docker.com/`

You need a free DockerHub account (`https://hub.docker.com/`) to login to the playground.

If after starting the environment, the screen goes black just refresh the web page.

Now you have a temporary “VM” session with Alpine Linux that last up to 4h and is mapped to a public IP. This instance already has Docker installed. You can add more instances, each with its own running shell. You can also create a “cluster” with a master and a chosen number of slaves by using the wrench tool.
You can use a basic GUI editor that launches as a separate browser window.

You can **transfer files** between your machine and that instance by using a service like `https://transfer.sh/`

1.6 Check running containers

To check the containers that are running execute:

```
$ docker ps
```

or

```
$ docker ps --all
```

This last command also shows the exit code of containers that ran previously.

1.7 Check local images

Local images can be checked with:

```
$ docker images
```

or

```
$ docker image ls
```

If you have not removed the `hello-world` from 1.3, you’ll see it listed.

1.8 Build a container (build)

Execute the following commands to prepare the environment for creating a new container:

```
$ mkdir hello_flask
$ cd hello_flask
```

Use a text editor to add the following to a file named `app.py` (available at Moodle)

```
# app.py - a minimal flask api using flask_restful
from flask import Flask
from flask_restful import Resource, Api

app = Flask(__name__)
api = Api(app)

class HelloWorld(Resource):
    def get(self):
        return {'hello': 'world'}

api.add_resource(HelloWorld, '/')

if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0')
```

The address ‘`0.0.0.0`’ allows the program to receive requests from any source IP.
Use a text editor to add the following to a file named `requirements.txt` (available at Moodle)

```
flask
flask_restful
```

Use a text editor to add the following to a file named `Dockerfile` (available at Moodle)

```
FROM python:3
WORKDIR /app
COPY requirements.txt /app
RUN pip install -r requirements.txt
COPY . /app
EXPOSE 5000
ENTRYPOINT ["python"]
CMD ["app.py"]
```

This `Dockerfile` copies our current folder into our container folder. It sets that folder as the working directory, installs our requirements and then runs the file using `python app.py`.

You can learn more about writing a `Dockerfile` at https://docs.docker.com/develop/develop-images/dockerfile_best-practices/ .

While remaining in the hello_flask directory, now we can build our docker image:

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

If all went well you should see at the end "`Successfully built …`". This last command built an image with the tag `hello_flask:latest` that includes everything in the current directory.

Check that it was created:

```
$ docker images
```

Now that the container was created it's time to run it.

1.9 Running a container (run)

Docker runs processes in isolated containers. A container is a process which runs on a host. The host may be local or remote. When an operator executes `docker run`, the container process that runs is isolated in that it has its own file system, its own networking, and its own isolated process tree separate from the host. We can run the container with:

```
$ docker run --name hello_flask -p 5000:5000 hello_flask:latest
```

The parameter `--name` allows the use of the specified name in future commands instead of the CONTAINER_ID. The parameter `-p` specifies the port it is going to run on. This is required because the program did not specify a port. This is a more flexible approach.

Let's make sure that the container is up and running:

```
$ docker container ls -l
```

If we want to list all containers that are defined on our system, then we do as follow:

```
$ docker container ls -a
```

Sometimes, we want to just list the IDs of all containers (useful for command composing):

```
$ docker container ls -q
```

You can see any output produced from the programs running in the container. Check that it's working by sending an HTTP GET request to `localhost:port` from another console window:

```
$ curl http://localhost:5000
```
or
```
$ curl http://127.0.0.1:5000
```

You should see both the HTTP response from the command and the log from Flask at the console connected to the container.

To get only the headers use the next command:

```
$ curl -I -X GET http://localhost:5000
```

1.10 Stopping a container

You can stop the container with a graceful shutdown (sending a `SIGTERM` and after the grace period the `SIGKILL`)

```
$ docker stop hello_flask
```

or in a forceful way (sending a `SIGKILL`)

```
$ docker kill hello_flask
```

In the preceding command, we have used the name of the container to specify which container we want to stop. But we could have also used the container ID instead.

How do we get the ID of a container? There are several ways of doing so. The manual approach is to list all running containers and find the one that we're looking for in the list. From there, we copy its ID. A more automated way is to use some shell scripting and environment variables. If, for example, we want to get the ID of the trivia container, we can use this expression:

```
$ export CONTAINER_ID=$(docker container ls -a | grep hello_flask | awk '{print $1}')
```

Now, instead of using the container name, we can use the `$CONTAINER_ID` variable in our expression:

```
$ docker container stop $CONTAINER_ID
```

1.11 Running a container (run) and cleaning (rm)

If you try to re-run the same container by giving it the same name, you'll get an error. Even though each execution of a container will always generate a new `CONTAINER_ID` and the *`hello_flask`* container is not running anymore it still exists in docker. Check this with:

```
$ docker ps --all
```

You'll see that container *`hello_flask`* has exited (with error code 0) but its remains are still accessible.

You can remove the *`hello_flask`* container:

```
$ docker rm hello_flask
```

Or you can remove all your containers (at any state):

```
$ docker container rm -f $(docker container ls -a -q)
```

By default, a container's file system persists even after the container exits. This makes debugging a lot easier (since you can inspect the final state) and you retain all your data by default. But if you are running short-term **foreground** processes, these container file systems can really pile up. If instead you'd like Docker to **automatically clean up the container and remove the file system when the container exits**, you can add the `--rm` flag.

```
$ docker run --name hello_flask --rm -p 5000:5000 hello_flask:latest
```

If we wanted to mute the container output, we can detach the run with a different command:

```
$ docker run --name hello_flask --rm -d -p 5000:5000 hello_flask:latest
```

The `-d` parameter runs the process as detached (background process), also known as a Daemon. You'll see that this command returns the UUID long identifier provided by the Docker daemon.

You can get the UUID short identifier (`CONTAINER ID`) by running

```
$ docker ps
```

Now, you know that a container is identified in three ways: **UUID long identifier**, **UUID short identifier** and, if set on the run command, a **name**.
Check that the container is running and that is operating correctly.

You can read more about running containers at https://docs.docker.com/engine/reference/run/ .

1.12 Inspecting containers

Containers are runtime instances of an image and have a lot of associated data that characterizes their behavior. To get more information about a specific container, we can use the `inspect` command:

```
$ docker container inspect hello_flask
```

You should see information such as the following:

- The ID of the container
- The creation date and time of the container
- The image from which the container is built
- Mounts
- Network settings
- …

Sometimes, we need just a tiny bit of the overall information, and to achieve this, we can either use the grep tool or a filter. Let's use a filter to see only the state part of the whole output in JSON:

```
$ docker container inspect -f "{{json .State}}" hello_flask | jq .
```

1.13 Executing commands in a container (exec)

The way to execute commands in a running container is by using the command `exec`. The command issued will run in the default directory of the container. If the underlying image has a custom directory specified with the WORKDIR directive in its `Dockerfile`, this will be used instead. The command should be an executable like the following that lists the directory contents of WORKDIR

```
$ docker exec hello_flask ls
```

Chained or quoted commands will not work. An example of a command that fails is

```
$ docker exec hello_flask "echo a && echo b"
```

Such a command can be executed successfully by running bash and providing it the string as input

```
$ docker exec hello_flask bash -c "echo a && echo b"
```

1.14 Open a command shell in a container

You can open a command shell inside the container with

```
$ docker exec -it hello_flask /bin/bash
```

Now you are inside the container in directory /app. Check its contents with:

```
$ ls
```

You'll find the same 3 original files. Why? Check the versions of python and pip with:

```
$ python --version
$ pip --version
```

We can define environment variables using the -e flag, as follows:

```
$ docker exec -it -e MY_VAR="Hello World" hello_flask /bin/bash
# echo $MY_VAR
Hello World
```

When you're done you can exit the shell with

```
exit
```

Or you can press `Ctrl + D`

Finally stop the container and check that there are no containers running.

1.15 Attaching to a running container

We can use the `attach` command to attach our Terminal's standard input, output, and error (or any combination of the three) to a running container using the ID or name of the container. Let's do this for our hello_flask container after starting it as a daemon (`-d` flag) and accessing it with `curl`:

```
$ docker run --name hello_flask --rm -d -p 5000:5000 hello_flask:latest
$ curl http://localhost:5000
$ docker attach hello_flask
```

Now we can check the logs written to the console by Flask.

Quit the container by pressing `Ctrl+C`. This will detach your Terminal and, at the same time, stop the `nginx` container.

1.16 Remove stopped containers (prune)

You can remove all stopped containers with

```
$ docker container prune
```

1.17 Cleaning all containers and images

If you want to remove all containers, networks, images and build cache, you do:

```
$ docker system prune -a
```

1.18 Saving and loading images

Another way to create a new container image is by importing or loading it from a file. A container image is nothing more than a tarball. To demonstrate this, we can use the `docker image save` command to export an existing image to a tarball, like this:

```
$ docker image save -o ./backup/hello_flask.tar hello_flask
```

If, on the other hand, we have an existing tarball and want to import it as an image into our system, we can use the `docker image load` command, as follows:

```
$ docker image load -i ./backup/hello_flask.tar
```

2 Logs

2.1 Retrieving container logs

It is a best practice for any good application to generate some logging information that developers and operators alike can use to find out what the application is doing at a given time, and whether there are any problems to help to pinpoint the root cause of the issue.
When running inside a container, the application should preferably output the log items to STDOUT and STDERR and not into a file. If the logging output is directed to STDOUT and STDERR, then Docker can collect this information and keep it ready for consumption by a user or any other external system:

1. To access the logs of a given container, we can use the `docker container logs` command. If, for example, we want to retrieve the logs of our `trivia` container, we can use the following expression:

```
$ docker container logs hello_flask
```

This will retrieve the whole log produced by the application from the very beginning of its existence.

2. If we want to only get a few of the latest entries, we can use the `-t` or `–tail` parameter, as follows:

```
$ docker container logs –tail 5 hello_flask
```

Sometimes, we want to follow the log that is produced by a container. This is possible when using the `-f` or `–follow` parameter.

```
$ docker container logs –tail 5 –follow hello_flask
```

2.2 Instrumenting a Python application

To instrument a Python application, we can use the logging package as follows:

```
import logging

logger = logging.getLogger("Sample App")
logger.setLevel(logging.WARN)
ch = logging.StreamHandler()
formatter = logging.Formater('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
ch.setFormatter(formatter)
logger.addHandler
(…)
logger.info("Accessing endpoint '/'"
```

Our logging message is formatted with the date and time at the beginning, then the name of the logger, the log level, and finally, our actual message defined on the last line of the application.

3 Data Volumes

3.1 Creating volumes

To create a new data volume, we can use the `docker volume create` command. This will create a named volume that can then be mounted into a container and used for persistent data access or storage. The following command creates a volume called `sample`, using the default volume driver:

```
$ docker volume create sample
```

The default volume driver is the so-called local driver, which stores the data locally in the host filesystem.

The easiest way to find out where the data is stored on the host is by using the `docker volume inspect` command on the volume we just created. The actual location can differ from system to system, and so, this is the safest way to find the target folder. You can see this command in the following code block:

```
$ docker volume inspect sample
```

There are other volume drivers available from third parties, in the form of plugins. We can use the `-driver` parameter in the `create` command to select a different volume driver. Other volume drivers use different types of storage systems to back a volume, such as cloud storage, **Network File System** (**NFS**) drives, software-defined storage, and more.

3.2 Mounting a volume

Once we have created a named volume, we can mount it into a container by following these steps:

```
$ docker run --name test -it -v sample:/data alpine /bin/sh
```

The preceding command mounts the `sample` volume to the `/data` folder inside the container.

Inside the container, we can now create files in the `/data` folder and then exit, as follows:

```
# cd /data
# echo "Test" > data.txt
# exit
```

If we navigate to the host folder that contains the data of the volume and list its content, we should see the file we just created inside the container.

3.3 Removing volumes

Volumes can be removed using the `docker volume rm` command. It is important to remember that **removing a volume destroys the containing data irreversibly**, and thus is to be considered a dangerous command. Docker helps us a bit in this regard, as it does not allow us to delete a volume that is still in use by a container. Always make sure before you remove or delete a volume that you either have a backup of its data or you really don't need this data anymore.

```
$ docker volume rm sample
```

3.4 Sharing data between containers

Containers are like sandboxes for the applications running inside them. This is mostly beneficial and wanted, in order to protect applications running in different containers from each other. It also means that the whole filesystem visible to an application running inside a container is private to this application, and no other application running in a different container can interfere with it.

At times, though, we want to share data between containers. Say an application running in container A produces some data that will be consumed by another application running in container B. How can

we achieve this? Well, I'm sure you've already guessed it—we can use Docker volumes for this purpose. We can create a volume and mount it to container A, as well as to container B. In this way, both applications A and B have access to the same data.

Now, as always when multiple applications or processes concurrently access data, we have to be very careful to avoid inconsistencies. To avoid concurrency problems such as race conditions, we ideally have only one application or process that is creating or modifying data, while all other processes concurrently accessing this data only read it. We can enforce a process running in a container to only be able to read the data in a volume by mounting this volume as read-only. Have a look at the following command:

```
$ docker run --name writer -it -v shared-data:/data alpine /bin/sh
```

Try to create a file inside this container, like this:

```
# echo "I can create a file" > /data/sample.txt
```

It should succeed.
Exit this container, and then execute the following command:

```
$ docker run --name reader -it -v shared-data:/app/data:ro hello_flask /bin/sh
```

Check that you can read the file /data/sample.txt. Now try to create a new file:

```
# echo "I try to create another file" > /app/data/data.txt
```

It will fail.

3.5 Using host volumes

In certain scenarios, such as when developing new containerized applications or when a containerized application needs to consume data from a certain folder produced—say—by a legacy application, it is very useful to use volumes that mount a specific host folder. Let's look at the following example:

```
$ docker run --rm -it -v $(pwd):/app/src alpine:latest /bin/sh
```

The preceding expression interactively starts an `alpine` container with a shell and mounts the `src` subfolder of the current directory into the container at `/app/src`. We need to use `$(pwd)` (or `` `pwd` ``, for that matter), which is the current directory, as when working with volumes, we always need to use absolute paths.
Developers use these techniques all the time when they are working on their application that runs in a container and want to make sure that the container always contains the latest changes they make to the code, without the need to rebuild the image and rerun the container after each change.

4 Configuring containers

More often than not, we need to provide some configuration to the application running inside a container. The configuration is often used to allow one and the same container to run in very different environments, such as in development, test, staging, or production environments.

In Linux, configuration values are often provided via environment variables.

We have learned that an application running inside a container is completely shielded from its host environment. Thus, the environment variables that we see on the host are different from the ones that we see from within a container.

Now, the good thing is that we can actually pass some configuration values into the container at start time. We can use the `--env` (or the short form, `-e`) parameter in the form `--env <key>=<value>` to do so, where `<key>` is the name of the environment variable and `<value>` represents the value to be associated with that variable.

5 Registries and repositories

5.1 Docker Registry (Docker Trusted Registry)

Docker registry is an enterprise-grade storage solution for Docker images. In other words, it's an image storage service. Well known cloud registries are for example: Docker Hub (https://hub.docker.com/), Quay (https://quay.io/), Google Container Registry (https://cloud.google.com/container-registry/) and Amazon Elastic Container Service (https://aws.amazon.com/ecr/).
One docker registry can contain many different docker repositories.

5.2 Docker Repository

Docker Repository is a collection of Docker images with the same name and different tags. Each image has its own tag. For example a tag can be:

```
python:3
```

5.3 Docker Hub

Besides providing a centralized resource for image discovery and distribution, Docker Hub's functionality extends to:

- Automated builds of images on source code changes and parallel builds
- Webhooks on image creation and push
- Groups and organizations management
- GitHub and BitBucket integration

In case you don't have a Doker Hub account, you can create one now (https://hub.docker.com/).
To push the image from the local machine to Docker Hub you login with:

```
$ docker login
```

… and enter the credentials of your account in the prompt.
Now re-tag the image with your username prefix (replace ACCOUNTNAME with the username) with:

```
$ docker tag hello_flask ACCOUNTNAME/hello_flask
```

After that, you can easily push the image by typing:

```
$ docker push ACCOUNTNAME/hello_flask
```

If we don't specify the tag, Docker will apply the `:latest` tag to it.
This might take a minute but now `hello_flask` is available from your public docker hub. You can set this up private if you would like.
If we want to pull the image from the Docker Hub to the local machine (not necessary in this case), we need to type

```
$ docker pull ACCOUNTNAME/hello_flask
```

If you don't specify the tag, you are going to pull the image tagged `:latest`.