Interacting with the container via volumes and ports

Let us get back to yt-dlp. It works yes, but it is quite laborious to get the downloaded videos to the host machine.

We can use Docker volumes to make it easier to store the downloads outside the container's ephemeral storage. With bind mount we can mount a file or directory from our own machine (the host machine) into the container.

Let's start a container with `-v` option, that requires an absolute path. We mount our current folder as `/mydir` in our container, overwriting everything that we have put in that folder in our Dockerfile.

```
$ docker run -v "$(pwd):/mydir" yt-dlp https://www.youtube.com/watch?v=DptFY_MszQs
```

So a volume is simply a folder (or a file) that is shared between the host machine and the container. If a file in volume is modified by a program that's running inside the container the changes are also saved from destruction when the container is shut down as the file exists on the host machine. This is the main use for volumes as otherwise all of the files wouldn't be accessible when restarting the container. Volumes also can be used to share files between containers and run programs that are able to load changed files.

In our yt-dlp we wanted to mount the whole directory since the files are fairly randomly named. If we wish to create a volume with only a single file we could also do that by pointing to it. For example `-v "$(pwd)/material.md:/mydir/material.md"` this way we could edit the material.md locally and have it change in the container (and vice versa). Note also that `-v` creates a directory if the file does not exist.

Exercise 1.9

EXERCISE 1.9: VOLUMES

In this exercise we won't create a new Dockerfile.

Image `devopsdockeruh/simple-web-service` creates a timestamp every two seconds to `/usr/src/app/text.log` when it's not given a command. Start the container with a bind mount so that the logs are created into your filesystem.

Submit the command you used to complete the exercise.

Hint: read the note that was made just before this exercise!

Allowing external connections into containers

This course does not provide an in-depth exploration of inter-program communication mechanisms. If you want to learn that in-depth, you should look at classes about Operating Systems or Networking. Here, you just need to know a few simple things:

- Sending messages: Programs can send messages to URL addresses such as this: http://127.0.0.1:3000 where HTTP is the *protocol*, 127.0.0.1 is an IP address, and 3000 is a *port*. Note the IP part could also be a *hostname*: 127.0.0.1 is also called *localhost* so instead you could use http://localhost:3000.
- Receiving messages: Programs can be assigned to listen to any available port. If a program is listening for traffic on port 3000, and a message is sent to that port, the program will receive and possibly process it.

The address *127.0.0.1* and hostname *localhost* are special ones, they refer to the machine or container itself, so if you are on a container and send a message to *localhost*, the target is the same container. Similarly, if you are sending the request from outside of a container to *localhost*, the target is your machine.

It is possible to **map your host machine port to a container port**. For example, if you map port 1000 on your host machine to port 2000 in the container, and then you send a message to http://localhost:2000 on your computer, the container will get that message if it's listening to its port 2000.

Opening a connection from the outside world to a Docker container happens in two steps:

- Exposing port
- Publishing port

Exposing a container port means telling Docker that the container listens to a certain port. This doesn't do much, except it helps humans with the configuration.

Publishing a port means that Docker will map host ports to the container ports.

To expose a port, add the line `EXPOSE <port>` in your Dockerfile

To publish a port, run the container with `-p <host-port>:<container-port>`

If you leave out the host port and only specify the container port, Docker will automatically choose a free port as the host port:

```
$ docker run -p 4567 app-in-port
```

We could also limit connections to a certain protocol only, e.g. UDP by adding the protocol at the end: `EXPOSE <port>/udp` and `-p <host-port>:<container-port>/udp`.

SECURITY REMINDER: OPENING A DOOR TO THE INTERNET

Since we are opening a port to the application, anyone from the internet could come in and access what you're running.

Don't haphazardly open just any ports - a way for an attacker to get in is by exploiting a port you opened to an insecure server. An easy way to avoid this is by defining the host-side port like this `-p 127.0.0.1:3456:3000`. This will only allow requests from your computer through port 3456 to the application port 3000, with no outside access allowed.

The short syntax, `-p 3456:3000`, will result in the same as `-p 0.0.0.0:3456:3000`, which truly is opening the port to everyone.

Usually, this isn't risky. But depending on the application, it is something you should consider!

Exercise 1.10

EXERCISE 1.10: PORTS OPEN

In this exercise, we won't create a new Dockerfile.

The image `devopsdockeruh/simple-web-service` will start a web service in port `8080` when given the argument "server". In Exercise 1.8 you already did an image that can be used to run the web service without any argument.

Use now the -p flag to access the contents with your browser. The output to your browser should be something like: `{ message: "You connected to the following path: ...`

Submit your used commands for this exercise.

Edit this page