Getting Started
Frequently Asked Questions
Part 1
Introduction to Part 1
Definitions and basic concepts
Running and stopping containers
In-depth dive into images
Defining start conditions for the container
Interacting with the container via volumes and ports
Utilizing tools from the Registry
Summary
Part 2
Part 3

Part 1 › Running and stopping containers

Running and stopping containers

Next we will start using a more useful image than hello-world. We can run Ubuntu just with `docker run ubuntu`.

```
$ docker run ubuntu
  Unable to find image 'ubuntu:latest' locally
  latest: Pulling from library/ubuntu
  83ee3a23efb7: Pull complete
  db98fc6f11f0: Pull complete
  f611acd52c6c: Pull complete
  Digest: sha256:703218c0465075f4425e58fac086e09e1de5c340b12976ab9eb8ad26615c3715
  Status: Downloaded newer image for ubuntu:latest
```

Anticlimactic as nothing really happened. The image was downloaded and ran and that was the end of that. It actually tried to open a shell but we will need to add a few flags to interact with it. `-t` will create a tty.

```
$ docker run -t ubuntu
  root@f83969ce2cd1:/#
```

Now we're inside the container and if we input `ls` and press enter... nothing happens. Because our terminal is not sending the messages into the container. The `-i` flag will instruct to pass the STDIN to the container. If you're stuck with the other terminal you can just stop the container.

```
$ docker run -it ubuntu
  root@2eb70ecf5789:/# ls
  bin  boot  dev  etc  home  lib  lib32  lib64  libx32  media  mnt  opt  proc  root  run  sbin  srv  sy
```

Great! Now we know at least 3 useful flags. `-i` (interactive), `-t` (tty) and `-d` (detached).

Let's throw in a few more and run a container in the background:

```
$ docker run -d -it --name looper ubuntu sh -c 'while true; do date; sleep 1; done'
```

> **QUOTES**
>
> If you are command prompt (Windows) user you must use double quotes around the script i.e. `docker run -d -it --name looper ubuntu sh -c "while true; do date; sleep 1; done"`. The quote or double-quote may haunt you later during the course.

- The first part, `docker run -d`. Should be familiar by now, run container detached.
- Followed by `-it` is short for `-i` and `-t`. Also familiar, `-it` allows you to interact with the container by using the command line.
- Because we ran the container with `--name looper`, we can now reference it easily.
- The image is `ubuntu` and what follows it is the command given to the container.

And to check that it's running, run `docker container ls`

Let's follow `-f` the output of logs with

```
$ docker logs -f looper
  Thu Mar  1 15:51:29 UTC 2023
  Thu Mar  1 15:51:30 UTC 2023
  Thu Mar  1 15:51:31 UTC 2023
  ...
```

Let's test pausing the looper without exiting or stopping it. In another terminal run `docker pause looper`. Notice how the logs output has paused in the first terminal. To unpause run `docker unpause looper`.

Keep the logs open and attach to the running container from the second terminal using 'attach':

```
$ docker attach looper
  Thu Mar  1 15:54:38 UTC 2023
  Thu Mar  1 15:54:39 UTC 2023
  ...
```

Now you have process logs (STDOUT) running in two terminals. Now press control+c in the attached window. The container is stopped because the process is no longer running.

If we want to attach to a container while making sure we don't close it from the other terminal we can specify to not attach STDIN with `--no-stdin` option. Let's start the stopped container with `docker start looper` and attach to it with `--no-stdin`.

Then try control+c.

```
$ docker start looper

$ docker attach --no-stdin looper
  Thu Mar  1 15:56:11 UTC 2023
  Thu Mar  1 15:56:12 UTC 2023
  ^C
```

The container will continue running. Control+c now only disconnects you from the STDOUT.

Running processes inside a container with docker exec

We often encounter situations where we need to execute commands within a running container. This can be achieved using the `docker exec` command.

We could e.g. list all the files inside the container default directory (which is the root) as follows:

```
$ docker exec looper ls -la
total 56
drwxr-xr-x   1 root root 4096 Mar  6 10:24 .
drwxr-xr-x   1 root root 4096 Mar  6 10:24 ..
-rwxr-xr-x   1 root root    0 Mar  6 10:24 .dockerenv
lrwxrwxrwx   1 root root    7 Feb 27 16:01 bin -> usr/bin
drwxr-xr-x   2 root root 4096 Apr 18  2022 boot
drwxr-xr-x   5 root root  360 Mar  6 10:24 dev
drwxr-xr-x   1 root root 4096 Mar  6 10:24 etc
drwxr-xr-x   2 root root 4096 Apr 18  2022 home
lrwxrwxrwx   1 root root    7 Feb 27 16:01 lib -> usr/lib
drwxr-xr-x   2 root root 4096 Feb 27 16:01 media
drwxr-xr-x   2 root root 4096 Feb 27 16:01 mnt
drwxr-xr-x   2 root root 4096 Feb 27 16:01 opt
dr-xr-xr-x 293 root root    0 Mar  6 10:24 proc
drwx------   2 root root 4096 Feb 27 16:08 root
drwxr-xr-x   5 root root 4096 Feb 27 16:08 run
lrwxrwxrwx   1 root root    8 Feb 27 16:01 sbin -> usr/sbin
drwxr-xr-x   2 root root 4096 Feb 27 16:01 srv
dr-xr-xr-x  13 root root    0 Mar  6 10:24 sys
drwxrwxrwt   2 root root 4096 Feb 27 16:08 tmp
drwxr-xr-x  11 root root 4096 Feb 27 16:01 usr
drwxr-xr-x  11 root root 4096 Feb 27 16:08 var
```

We can execute the Bash shell in the container in interactive mode and then run any commands within that Bash session:

```
$ docker exec -it looper bash

  root@2a49df3ba735:/# ps aux

  USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
  root         1  0.2  0.0   2612  1512 pts/0    Ss+  12:36   0:00 sh -c while true; do date; sleep 1;
  root        64  1.5  0.0   4112  3460 pts/1    Ss   12:36   0:00 bash
  root        79  0.0  0.0   2512   584 pts/0    S+   12:36   0:00 sleep 1
  root        80  0.0  0.0   5900  2844 pts/1    R+   12:36   0:00 ps aux
```

From the `ps aux` listing we can see that our `bash` process got PID (process ID) of 64.

Now that we're inside the container it behaves as you'd expect from Ubuntu, and we can exit the container with `exit` and then either kill or stop the container.

Our looper won't stop for a SIGTERM signal sent by a stop command. To terminate the process, stop follows the SIGTERM with a SIGKILL after a grace period. In this case, it's simply faster to use kill.

```
$ docker kill looper
$ docker rm looper
```

Running the previous two commands is basically equivalent to running `docker rm --force looper`

Let's start another process with `-it` and add `--rm` in order to remove it automatically after it has exited. The `--rm` ensures that there are no garbage containers left behind. It also means that `docker start` can not be used to start the container after it has exited.

```
$ docker run -d --rm -it --name looper-it ubuntu sh -c 'while true; do date; sleep 1; done'
```

Now let's attach to the container and hit control+p, control+q to detach us from the STDOUT.

```
$ docker attach looper-it

  Mon Jan 15 19:50:42 UTC 2018
  Mon Jan 15 19:50:43 UTC 2018
  ^P^Qread escape sequence
```

Instead, if we had used ctrl+c, it would have sent a kill signal followed by removing the container as we specified `--rm` in `docker run` command.

Exercise 1.3

> ⓘ **EXERCISE 1.3: SECRET MESSAGE**
>
> Now that we've warmed up it's time to get inside a container while it's running!
>
> Image `devopsdockeruh/simple-web-service:ubuntu` will start a container that outputs logs into a file. Go inside the running container and use `tail -f ./text.log` to follow the logs. Every 10 seconds the clock will send you a "secret message".
>
> Submit the secret message and command(s) given as your answer.

Nonmatching host platform

If you are working with M1/M2 Mac, you quite likely end up with the following warning when running the image *devopsdockeruh/simple-web-service:ubuntu*:

```
WARNING: The requested image's platform (linux/amd64) does not match the detected
host platform (linux/arm64/v8) and no specific platform was requested
```

Despite this warning, you can run the container. The warning basically says what's wrong, the image uses a different processor architecture than your machine.

The image can be used because Docker Desktop for Mac employs an emulator by default when the image's processor architecture does not match the host's. However, it's important to note that emulated execution may be less efficient in terms of performance than running the image on a compatible native processor architecture.

When you run `docker run ubuntu` for example, you don't get a warning, why is that? Quite a few popular images are so-called multi platform images, which means that one image contains variations for different architectures. When you are about to pull or run such an image, Docker will detect the host architecture and give you the correct type of image.

Ubuntu in a container is just... Ubuntu

A container that is running a Ubuntu image works quite like a normal Ubuntu:

```
$ docker run -it ubuntu
root@881a1d4ecff2:/# ls
bin   dev  home  media  opt   root  sbin  sys  usr
boot  etc  lib   mnt    proc  run   srv   tmp  var
root@881a1d4ecff2:/# ps
  PID TTY          TIME CMD
    1 pts/0    00:00:00 bash
   13 pts/0    00:00:00 ps
root@881a1d4ecff2:/# date
Wed Mar  1 12:08:24 UTC 2023
root@881a1d4ecff2:/#
```

An image like Ubuntu contains already a nice set of tools but sometimes just the one that we need is not within the standard distribution. Let us assume that we would like to edit some files inside the container. The good old Nano editor is a perfect fit for our purposes. We can install it in the container by using apt-get:

```
$ docker run -it ubuntu
root@881a1d4ecff2:/# apt-get update
root@881a1d4ecff2:/# apt-get -y install nano
root@881a1d4ecff2:/# cd tmp/
root@881a1d4ecff2:/tmp# nano temp_file.txt
```

As can be seen, installing a program or library to a container happens just like the installation is done in "normal" Ubuntu. The remarkable difference is that the installation of Nano is not permanent, that is, if we remove our container, all is gone. We shall soon see how to get a more permanent solution for building images that are perfect to our purposes.

Exercise 1.4

> ⓘ **EXERCISE 1.4: MISSING DEPENDENCIES**
>
> Start a Ubuntu image with the process `sh -c 'while true; do echo "Input website:"; read website; echo "Searching.."; sleep 1; curl http://$website; done'`
>
> If you're on Windows, you'll want to switch the `'` and `"` around: `sh -c "while true; do echo 'Input website:'; read website; echo 'Searching..'; sleep 1; curl http://$website; done"`.
>
> You will notice that a few things required for proper execution are missing. Be sure to remind yourself which flags to use so that the container actually waits for input.
>
> > Note also that curl is NOT installed in the container yet. You will have to install it from inside of the container.
>
> Test inputting `helsinki.fi` into the application. It should respond with something like
>
> ```
> <html>
>   <head>
>     <title>301 Moved Permanently</title>
>   </head>
>
>   <body>
>     <h1>Moved Permanently</h1>
>     <p>The document has moved <a href="http://www.helsinki.fi/">here</a>.</p>
>   </body>
> </html>
> ```
>
> This time return the command you used to start process and the command(s) you used to fix the ensuing problems.
>
> **Hint** for installing the missing dependencies you could start a new process with `docker exec`.
>
> - This exercise has multiple solutions, if the curl for helsinki.fi works then it's done. Can you figure out other (smart) solutions?

Edit this page

Previous
« Definitions and basic concepts

Next
In-depth dive into images »