In-depth dive into images

Images are the basic building blocks for containers and other images. When you "containerize" an application you work towards creating the image.

By learning what images are and how to create them you are ready to start utilizing containers in your own projects.

Where do the images come from?

When running a command such as `docker run hello-world`, Docker will automatically search Docker Hub for the image if it is not found locally.

This means that we can pull and run any public image from Docker's servers. For example, if we wanted to start an instance of the PostgreSQL database, we could just run `docker run postgres`, which would pull and run https://hub.docker.com/_/postgres/.

We can search for images in the Docker Hub with `docker search`. Try running `docker search hello-world`.

The search finds plenty of results, and prints each image's name, short description, amount of stars, and "official" and "automated" statuses.

```
$ docker search hello-world

  NAME                      DESCRIPTION    STARS   OFFICIAL   AUTOMATED
  hello-world               Hello World!…  1988     [OK]
  kitematic/hello-world-ngi…  A light-weig…  153
  tutum/hello-world         Image to tes…  90                 [OK]
  ...
```

Let's examine the list.

The first result, `hello-world`, is an official image. Official images are curated and reviewed by Docker, Inc. and are usually actively maintained by the authors. They are built from repositories in the docker-library.

When browsing the CLI's search results, you can recognize an official image from the "[OK]" in the "OFFICIAL" column and also from the fact that the image's name has no prefix (aka organization/user). When browsing Docker Hub, the page will show "Docker Official Images" as the repository, instead of a user or organization. For example, see the Docker Hub page of the `hello-world` image.

The third result, `tutum/hello-world`, is marked as "automated". This means that the image is automatically built from the source repository. Its Docker Hub page shows its previous "Builds" and a link to the image's "Source Repository" (in this case, to GitHub) from which Docker Hub builds the image.

The second result, `kitematic/hello-world-nginx`, is neither an official nor an automated image. We can't know what the image is built from, since its Docker Hub page has no links to any repositories. The only thing its Docker Hub page reveals is that the image is 9 years old. Even if the image's "Overview" section had links to a repository, we would have no guarantees that the published image was built from that source.

There are also other Docker registries competing with Docker Hub, such as Quay. By default, `docker search` will only search from Docker Hub, but to a search different registry, you can add the registry address before the search term, for example, `docker search quay.io/hello`. Alternatively, you can use the registry's web pages to search for images. Take a look at the page of the `nordstrom/hello-world` image on Quay. The page shows the command to use to pull the image, which reveals that we can also pull images from hosts other than Docker Hub:

`docker pull quay.io/nordstrom/hello-world`

So, if the host's name (here: `quay.io`) is omitted, it will pull from Docker Hub by default.

NOTE: Trying the above command may fail giving manifest errors as the default tag latest is not present in quay.io/nordstrom/hello-world image. Specifying a correct tag for a image will pull the image without any errors, for ex. `docker pull quay.io/nordstrom/hello-world:2.0`

A detailed look into an image

Let's go back to a more relevant image than 'hello-world', the Ubuntu image, one of the most common Docker images to use as a base for your own image.

Let's pull Ubuntu and look at the first lines:

```
$ docker pull ubuntu
  Using default tag: latest
  latest: Pulling from library/ubuntu
```

Since we didn't specify a tag, Docker defaulted to `latest`, which is usually the latest image built and pushed to the registry. **However**, in this case, the repository's README says that the `ubuntu:latest` tag points to the "latest LTS" instead since that's the version recommended for general use.

Images can be tagged to save different versions of the same image. You define an image's tag by adding `:<tag>` after the image's name.

Ubuntu's Docker Hub page reveals that there's a tag named 22.04 which promises us that the image is based on Ubuntu 22.04. Let's pull that as well:

```
$ docker pull ubuntu:22.04

  22.04: Pulling from library/ubuntu
  c2ca09a1934b: Downloading [==========================================>      ]  34.2SMB/38.64MB
  d6c3619d2153: Download complete
  0efe07335a04: Download complete
  6b1bb01b3a3b: Download complete
  43a98c187399: Download complete
```

Images are composed of different layers that are downloaded in parallel to speed up the download. Images being made of layers also have other aspects and we will talk about them in part 3.

We can also tag images locally for convenience, for example, `docker tag ubuntu:22.04 ubuntu:jammy_jellyfish` creates the tag `ubuntu:jammy_jellyfish` which refers to `ubuntu:22.04`.

Tagging is also a way to "rename" images. Run `docker tag ubuntu:22.04 fav_distro:jammy_jellyfish` and check `docker image ls` to see what effects the command had.

To summarize, an image name may consist of 3 parts plus a tag. Usually like the following: `registry/organisation/image:tag`. But may be as short as `ubuntu`, then the registry will default to Docker hub, organisation to *library* and tag to *latest*. The organisation may also be a user, but calling it an organisation may be more clear.

Exercises 1.5 - 1.6

> **EXERCISE 1.5: SIZES OF IMAGES**
>
> In the Exercise 1.3 we used `devopsdockeruh/simple-web-service:ubuntu`.
>
> Here is the same application but instead of Ubuntu is using Alpine Linux: `devopsdockeruh/simple-web-service:alpine`.
>
> Pull both images and compare the image sizes. Go inside the Alpine container and make sure the secret message functionality is the same. Alpine version doesn't have `bash` but it has `sh`, a more bare-bones shell.

> **EXERCISE 1.6: HELLO DOCKER HUB**
>
> Run `docker run -it devopsdockeruh/pull_exercise`.
>
> The command will wait for your input.
>
> Navigate through the Docker hub to find the docs and Dockerfile that was used to create the image.
>
> Read the Dockerfile and/or docs to learn what input will get the application to answer a "secret message".

Submit the secret message and command(s) given to get it as your answer.

Building images

Finally, we get to build our own images and get to talk about `Dockerfile` and why it's so great.

Dockerfile is simply a file that contains the build instructions for an image. You define what should be included in the image with different instructions. We'll learn about the best practices here by creating one.

Let's take a most simple application and containerize it first. Here is a script called "hello.sh"

hello.sh

```
#!/bin/sh

echo "Hello, docker!"
```

First, we will test that it even works. Create the file, add execution permissions and run it:

```
$ chmod +x hello.sh

$ ./hello.sh
  Hello, docker!
```

- If you're using Windows you can skip these two and add chmod +x hello.sh to the Dockerfile.

And now to create an image from it. We'll have to create the `Dockerfile` that declares all of the required dependencies. At least it depends on something that can run shell scripts. We will choose Alpine, a small Linux distribution that is often used to create small images.

Even though we're using Alpine here, you can use Ubuntu during exercises. Ubuntu images by default contain more tools to debug what is wrong when something doesn't work. In part 3 we will talk more about why small images are important.

We will choose exactly which version of a given image we want to use. This guarantees that we don't accidentally update through a breaking change, and we know which images need updating when there are known security vulnerabilities in old images.

Now create a file and name it "Dockerfile" and put the following instructions inside it:

Dockerfile

```
# Start from the alpine image that is smaller but no fancy tools
FROM alpine:3.19

# Use /usr/src/app as our workdir. The following instructions will be executed in this location.
WORKDIR /usr/src/app

# Copy the hello.sh file from this directory to /usr/src/app/ creating /usr/src/app/hello.sh
COPY hello.sh .

# Alternatively, if we skipped chmod earlier, we can add execution permissions during the build.
# RUN chmod +x hello.sh

# When running docker run the command will be ./hello.sh
CMD ./hello.sh
```

Great! We can use the command docker build to turn the Dockerfile to an image.

By default `docker build` will look for a file named Dockerfile. Now we can run `docker build` with instructions where to build (`.`) and give it a name (`-t <name>`):

```
$ docker build . -t hello-docker
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 478B
 => [internal] load metadata for docker.io/library/alpine:3.19
 => [auth] library/alpine:pull token for registry-1.docker.io
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [1/3] FROM docker.io/library/alpine:3.19@sha256:c5b1261d6d3e43071626931fc004f70149baeba2c8ec672bd4f
 => [internal] load build context
 => => transferring context: 68B
 => [2/3] WORKDIR /usr/src/app
 => [3/3] COPY hello.sh .
 => exporting to image
 => => exporting layers
 => => writing image sha256:5f8f5d7445f34b0bcfaaa4d685a068cdccc1ed79e65068337a3a228c79ea69c8
 => => naming to docker.io/library/hello-docker
```

Let us ensure that the image exists:

```
$ docker image ls
  REPOSITORY           TAG          IMAGE ID       CREATED         SIZE
  hello-docker         latest       5f8f5d7445f3   4 minutes ago   7.73MB
```

> **PERMISSION DENIED**
>
> If you're now getting "/bin/sh: ./hello.sh: Permission denied" it's because the `chmod +x hello.sh` was skipped earlier. You can simply uncomment the RUN instruction between COPY and CMD instructions

> **NOT FOUND**
>
> If you're now getting "/bin/sh: ./hello.sh: not found" and you're using Windows it might be because by default Windows uses CRLF as line ending. Unix, in our case Alpine, uses just LF which makes the copying of our `hello.sh` invalid bash script in the build phase. To overcome this error change the line endings to LF before running `docker build`

Now executing the application is as simple as running `docker run hello-docker`. Try it!

During the build we see from the output that there are three steps: [1/3], [2/3] and [3/3]. The steps here represent layers of the image so that each step is a new layer on top of the base image (alpine:3.19 in our case).

Layers have multiple functions. We often try to limit the number of layers to save on storage space but layers can work as a cache during build time. If we just edit the last lines of Dockerfile the build command can start from the previous layer and skip straight to the section that has changed. COPY automatically detects changes in the files, so if we change the hello.sh it'll run from step 3/3, skipping 1 and 2. This can be used to create faster build pipelines. We'll talk more about optimization in part 3.

It is also possible to manually create new layers on top of a image. Let us now create a new file called `additional.txt` and copy it inside a container.

We'll need two terminals, that shall be called 1 and 2 in the following listings. Let us start by running the image:

```
# do this in terminal 1
$ docker run -it hello-docker sh
/usr/src/app #
```

Now we're inside of the container. We replaced the CMD we defined earlier with `sh` and used -i and -t to start the container so that we can interact with it.

In the second terminal we will copy the file inside the container:

```
# do this in terminal 2
$ docker ps
  CONTAINER ID   IMAGE          COMMAND   CREATED         STATUS         PORTS     NAMES
  9c06b95e3e85   hello-docker   "sh"      4 minutes ago   Up 4 minutes             zen_rosalind

$ touch additional.txt
$ docker cp ./additional.txt zen_rosalind:/usr/src/app/
```

The file is created with command `touch` right before copying it in.

Let us ensure that the file is copied inside the container:

```
# do this in terminal 1
/usr/src/app # ls
additional.txt  hello.sh
```

Great! Now we've made a change to the container. We can use command docker diff to check what has changed

```
# do this in terminal 2
$ docker diff zen_rosalind
  C /usr
  C /usr/src
  C /usr/src/app
  A /usr/src/app/additional.txt
  C /root
  A /root/.ash_history
```

The character in front of the file name indicates the type of the change in the container's filesystem: A = added, D = deleted, C = changed. The additional.txt was created and our `ls` created .ash_history.

Next we will save the changes as *a new image* with the command docker commit:

```
# do this in terminal 2
$ docker commit zen_rosalind hello-docker-additional
  sha256:2f63baa355ce5976bf89fe6000b92717f25dd91172aed71620e784315bfc4fd
$ docker image ls
  REPOSITORY                  TAG          IMAGE ID       CREATED          SIZE
  hello-docker-additional     latest       2f63baa355ce   3 seconds ago    7.73MB
  hello-docker                latest       444f21cf7bd5   31 minutes ago   7.73MB
```

Technically the command `docker commit` added a new layer on top of the image `hello-docker`, and the resulting image was given the name `hello-docker-additional`.

We will actually not use the command `docker commit` again during this course. This is because defining the changes to the Dockerfile is much more sustainable method of managing changes. No magic actions or scripts, just a Dockerfile that can be version controlled.

Let's do just that and create hello-docker with v2 tag that includes the file additional.txt. The new file can be added with a RUN instruction:

Dockerfile

```
# Start from the alpine image
FROM alpine:3.19

# Use /usr/src/app as our workdir. The following instructions will be executed in this location.
WORKDIR /usr/src/app

# Copy the hello.sh file from this location to /usr/src/app/ creating /usr/src/app/hello.sh.
COPY hello.sh .

# Execute a command with `/bin/sh -c` prefix.
RUN touch additional.txt

# When running Docker run the command will be ./hello.sh
CMD ./hello.sh
```

Now we used the RUN instruction to execute the command `touch additional.txt` which creates a file inside the resulting image. Pretty much anything that can be executed in the container based on the created image, can be instructed to be run with the RUN instruction during the build of a Dockerfile.

Build now the Dockerfile with `docker build . -t hello-docker:v2` and we are done! Let's compare the output of ls:

```
$ docker run hello-docker-additional ls
  additional.txt
  hello.sh

$ docker run hello-docker:v2 ls
  additional.txt
  hello.sh
```

Now we know that all instructions in a Dockerfile **except** CMD (and one other that we will learn about soon) are executed during build time. **CMD** is executed when we call docker run, unless we overwrite it.

Exercises 1.7 - 1.8

EXERCISE 1.7: IMAGE FOR SCRIPT

We can improve our previous solutions now that we know how to create and build a Dockerfile.

Let us now get back to Exercise 1.4.

Create a new file `script.sh` on your local machine with the following contents:

```
while true
do
  echo "Input website:"
  read website; echo "Searching.."
  sleep 1; curl http://$website
done
```

Create a Dockerfile for a new image that starts from *ubuntu:22.04* and add instructions to install `curl` into that image. Then add instructions to copy the script file into that image and finally set it to run on container start using CMD.

After you have filled the Dockerfile, build the image with the name "curler".

- If you are getting permission denied, use `chmod` to give permission to run the script.

The following should now work:

```
$ docker run -it curler

  Input website:
  helsinki.fi
  Searching..
  <!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
  <html><head>
  <title>301 Moved Permanently</title>
  </head><body>
  <h1>Moved Permanently</h1>
  <p>The document has moved <a href="https://www.helsinki.fi/">here</a>.</p>
  </body></html>
```

Remember that RUN can be used to execute commands while building the image!

Submit the Dockerfile.

EXERCISE 1.8: TWO LINE DOCKERFILE

By default our `devopsdockeruh/simple-web-service:alpine` doesn't have a CMD. Instead, it uses *ENTRYPOINT* to declare which application is run.

We'll talk more about *ENTRYPOINT* in the next section, but you already know that the last argument in `docker run` can be used to give a command or an argument.

As you might've noticed it doesn't start the web service even though the name is "simple-web-service". A suitable argument is needed to start the server!

Try `docker run devopsdockeruh/simple-web-service:alpine hello`. The application reads the argument "hello" but will inform that hello isn't accepted.

In this exercise create a Dockerfile and use FROM and CMD to create a brand new image that automatically runs `server`.

The Docker documentation CMD says a bit indirectly that if a image has ENTRYPOINT defined, CMD is used to define it the default arguments.

Tag the new image as "web-server"

Return the Dockerfile and the command you used to run the container.

Running the built "web-server" image should look like this:

```
$ docker run web-server
[GIN-debug] [WARNING] Creating an Engine instance with the Logger and Recovery middleware already a

[GIN-debug] [WARNING] Running in "debug" mode. Switch to "release" mode in production.
- using env:   export GIN_MODE=release
- using code:  gin.SetMode(gin.ReleaseMode)

[GIN-debug] GET    /*path                    --> server.Start.func1 (3 handlers)
[GIN-debug] Listening and serving HTTP on :8080
```

- We don't have any method of accessing the web service yet. As such confirming that the console output is the same will suffice.
- The exercise title may be a useful hint here.

Edit this page

Previous
« Running and stopping containers

Next
Defining start conditions for the container »