Introduction to Docker

Mikhail Dozmorov

Summer 2018

Resources

- Docker Curriculum: A comprehensive tutorial for getting started with Docker. Teaches how to use Docker and deploy dockerized apps on AWS with Elastic Beanstalk and Elastic Container Service. https://docker-curriculum.com/

Install Docker

https://www.docker.com/community-edition#/

Testing your installation

```
$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
ca4f61b1923c: Pull complete
Digest: sha256:97ce6fa4b6cdc0790cda65fe7290b74cfebd9fa0c9b8c38e979330d547d22ce1
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.
...
```

Playing with Busybox

https://en.wikipedia.org/wiki/BusyBox

```
$ docker pull busybox
Using default tag: latest
latest: Pulling from library/busybox
d070b8ef96fc: Pull complete
Digest: sha256:2107a35b58593c58ec5f4e8f2c4a70d195321078aebfadfbfb223a2ff4a4ed21
Status: Downloaded newer image for busybox:latest
```

Docker first checked if this image is available on your computer and since it wasn't it downloaded the image from Docker Hub. So getting an image from Docker Hub works sort of automatically. If you just want to pull the image but not run it, you can also do `docker pull busybox`

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
busybox             latest              f6e427c148a7        3 weeks ago         1.15MB
hello-world         latest              f2a91732366c        4 months ago        1.85kB
```

- REPOSITORY. The name of the repository
- TAG. The label of a specific set point in the repositories' commit history

- IMAGE ID. This is like the primary key for the image
- CREATED. The date the repository was created, as opposed to when it was pulled
- SIZE. The size of the image

Try `docker history busybox` for a more granular view of the layers in the image.

Running containers

A container is a running image that you start with the docker run command, like this:

```
$ docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
```

The command consists of:

- OPTIONS. There are a LOT of options for the run command
- IMAGE. The name of the image that the container should start from
- COMMAND. The command that should run on the container when it starts

```
$ docker run busybox
$
```

Wait, nothing happened! Is that a bug? Well, no. Behind the scenes, a lot of stuff happened. When you call run, the Docker client finds the image (busybox in this case), loads up the container and then runs a command in that container. When we run docker run busybox, we didn't provide a command, so the container booted up, ran an empty command and then exited. Well, yeah - kind of a bummer. Let's try something more exciting.

```
$ docker run busybox echo "hello from busybox"
hello from busybox
```

The `docker ps` command shows you all containers that are currently running.

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              POR
```

Since no containers are running, we see a blank line. Let's try a more useful variant: `docker ps -a`

```
$ docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED              STATUS
0a3c782b5bd9        busybox             "echo 'hello from bu..."   About a minute ago   Exited (0) 2 mi
935297c6c396        busybox             "sh"                     7 minutes ago        Exited (0) 8 minu
bcb097378895        hello-world         "/hello"                 17 hours ago         Exited (0) 17 hou
```

So what we see above is a list of all containers that we ran. Do notice that the STATUS column shows that these containers exited a few minutes ago.

- The CONTAINER_ID, a unique identifier you can use to refer to the container in other commands (this is kind of like a process id in Linux)
- The IMAGE that was used when the container started
- The COMMAND used to start the container
- The time the underlying image was CREATED
- The uptime STATUS
- The exposed PORTS
- A human readable NAME that you can use in place of the ID

You're probably wondering if there is a way to run more than just one command in a container. Do `docker run --help`. Let's try that now:

```
$ docker run -it busybox sh
/ # ls
bin   dev   etc   home  proc  root  sys   tmp   usr   var
```

```
/ # uptime
 05:45:21 up  5:58,  0 users,  load average: 0.00, 0.01, 0.04
```

Running the run command with the `-it` flags attaches us to an interactive tty in the container. Now we can run as many commands in the container as we want. Take some time to run your favorite commands.

Danger Zone: If you're feeling particularly adventurous you can try `rm -rf bin` in the container. Make sure you run this command in the container and not in your laptop. Doing this will not make any other commands like ls, echo work. Once everything stops working, you can exit the container (type exit and press Enter) and then start it up again with the docker run -it busybox sh command. Since Docker creates a new container every time, everything should start working again.

Other useful flags for `docker run`:

- `-d`, `--detach` - Run container in background and print container ID
- `-p`, `--publish list` - Publish a container's port(s) to the host
- `-e`, `--env list` - Set environment variables
- `-u`, `--user string` - Username or UID (format: [:])
- `-w`, `--workdir string` - Working directory inside the container

Attaching storage

```
$ docker run -it -v /Users/mdozmorov/Documents/nobackup:/work busybox sh
```

- The `-v` flag mounts a directory from your host into your Docker container. The flag structure is:

  `[host-path]:[container-path]:[rw|ro]`

- If `[host-path]` or `[container-path]` doesn't exist it is created
- You can control the write status of the volume with the `ro` and `rw` options
- If you don't specify `rw` or `ro`, it will be `rw` by default

Cleaning up

Before we move ahead though, let's quickly talk about deleting containers. We saw above that we can still see remnants of the container even after we've exited by running `docker ps -a`. Throughout this tutorial, you'll run docker run multiple times and leaving stray containers will eat up disk space. Hence, as a rule of thumb, I clean up containers once I'm done with them. To do that, you can run the docker rm command. Just copy the container IDs from above and paste them alongside the command.

```
$ docker rm 690c924270f6 0a3c782b5bd9
690c924270f6
0a3c782b5bd9
```

On deletion, you should see the IDs echoed back to you. If you have a bunch of containers to delete in one go, copy-pasting IDs can be tedious. In that case, you can simply run -

```
$ docker rm $(docker ps -a -q -f status=exited)
```

This command deletes all containers that have a status of exited. In case you're wondering, the `-q` flag, only returns the numeric IDs and `-f` filters output based on conditions provided. One last thing that'll be useful is the `--rm` flag that can be passed to docker run which automatically deletes the container once it's exited from. For one off docker runs, `--rm` flag is very useful. Lastly, you can also delete images that you no longer need by running `docker rmi`.

After working with Docker for some time, you start accumulating other development junk: unused volumes, networks, exited containers and unused images. One Command to "Rule Them All"

```
$ docker system prune
WARNING! This will remove:
        - all stopped containers
        - all networks not used by at least one container
        - all dangling images
        - all build cache
Are you sure you want to continue? [y/N] y
Deleted Containers:
e8a1c3bf03acf115ed13bfb91f8df724a675440943dff0ec6f24f404b2d49c75
d38390d8d08230dcd0f72ea1fbb41460dcb9c89e75cf97d8138ab5a2f1946565
935297c6c3969b7055aee511be7db717be474ffb0bb01048ba247bccf933444b
bcb097378895f3cfa8db661799a69437bf59ff7893cd9238c3b240567ee53dfe

Deleted Networks:
docker_gwbridge

Total reclaimed space: 12B
```

Building images

Creating the first images

- There is a special empty image called `scratch`
- It allows to build from scratch
- The docker import command loads a tarball into Docker
- The imported tarball becomes a standalone image
- That new image has a single layer

Note: you will probably never have to do this yourself

Creating other images

`docker commit`

- Saves all the changes made to a container into a new layer
- Creates a new image (effectively a copy of the container)

`docker build`

- Performs a repeatable build sequence
- This is the preferred method!

Docker commit

First, some useful docker images

`docker-rstudio` - RStudio in Docker with knitr, RMarkdown, and enough TeX to generate PDF

https://github.com/mccahill/docker-rstudio

```
# May take up to an hour to build
docker build -t="r-studio" .

docker run -d -e USERPASS=password  -v /Users/mdozmorov/Documents/nobackup:/home/guest \
    -p 0.0.0.0:8787:8787 -it r-studio
```

rocker - R configurations for Docker

RStudio rocker container instructions: https://github.com/rocker-org/rocker/wiki/Using-the-RStudio-image

```
docker run -d -p 8787:8787 rocker/rstudio

# Avoid changing the permissions in the linked volume on the host
docker run -d -P -v $(pwd):/home/$USER/foo -e USERID=$UID rocker/rstudio

# The interactive method doesn't handle alternate UIDs
docker run --rm -it -e USER=$USER  -e USERID=$UID rocker/rstudio bash
# You are still root, so need to run:
bash /etc/cont-init.d/userconf
su $USER
# From here, you are the non-root $USER, then run, e.g., R
R
```

Installing R packages

Start `rocker/rstudio` as usual, open `http://localhost:8787` in your web-browser. Your username is `guest` and your password is `password`

```
docker run -d -e USERPASS=password  -v /Users/mdozmorov/Documents/nobackup:/home/guest \
    -p 0.0.0.0:8787:8787 -it r-studio
```

Within RStudio, install packages as usual, e.g.,

```
source("https://bioconductor.org/biocLite.R")
biocLite("limma")
```

Installing external dependencies

Many R packages have dependencies external to R, for example GSL - GNU Scientific Library, https://www.gnu.org/software/gsl/. To install these on a running rocker container you need to go to the docker command line (in a new terminal window) and type the following:

```
docker ps # find the ID of the running container you want to add a package to
docker exec -it <container-id> bash # a docker command to start a bash shell in your container
apt-get install libgsl0-dev # install the package, in this case GSL
```

If you get an error message when running apt-get install `libgsl0-dev` try running `apt-get update` first. See "Saving your work" on how to commit/save these changes

Saving your work

To save a Docker image we have to provide a commit message to describe the change that we have made to the image. We do this by passing the `-m` flag followed by the message in quotes. We also need to provide the

specific hash for this version of the container (here `af2e4318ef6d`). Finally, we also provide a new name for the new image. We called this new image `rocker_limma`.

```
docker commit -m "Add limma" af2e4318ef6d rocker_limma

# Check your work
docker images
```

Getting an image to Docker Hub

Imagine you made your own Docker image and would like to share it with the world you can sign up for an account on https://hub.docker.com/. After verifying your email you are ready to go and upload your first docker image.

1. Log in on https://hub.docker.com/
2. Click on *Create Repository*.
3. Choose a name (e.g. `rocker_limma`) and a description for your repository and click *Create*.

Log into the Docker Hub from the command line

```
docker login --username=yourhubusername --email=youremail@company.com
```

just with your own user name and email that you used for the account. Enter your password when prompted. If everything worked you will get a message similar to

```
WARNING: login credentials saved in /home/username/.docker/config.json
Login Succeeded
```

Check the image ID using

```
docker images
```

and what you will see will be similar to

```
REPOSITORY          TAG                 IMAGE ID            CREATED                  SIZE
rocker_limma        latest              e43e2aef81c7        Less than a second ago   5GB
ubuntu              16.04               f975c5035748        2 weeks ago              112MB
busybox             latest              f6e427c148a7        3 weeks ago              1.15MB
hello-world         latest              f2a91732366c        4 months ago             1.85kB
```

and tag your image

```
docker tag e43e2aef81c7 yourhubusername/rocker_limma:firsttry
```

The number must match the image ID and `:firsttry` is the tag. In general, a good choice for a tag is something that will help you understand what this container should be used in conjunction with, or what it represents. If this container contains the analysis for a paper, consider using that paper's DOI or journal-issued serial number; if it's meant for use with a particular version of a code or data version control repo, that's a good choice too - whatever will help you understand what this particular image is intended for.

Push your image to the repository you created

```
docker push yourhubusername/rocker_limma
```

Your image is now available for everyone to use.

Saving and loading images as files

To save a Docker image after you have pulled, committed or built it you use the docker save command. For example, lets save a local copy of the `rocker_limma` docker image we made:

```
docker save rocker_limma > rocker_limma.tar
```

If we want to load that Docker container from the archived tar file in the future, we can use the docker load command:

```
docker load --input rocker_limma.tar
```

Searching for images

- "Stars" indicate the popularity of the image
- "Official" images are those in the root namespace
- "Automated" images are built automatically by the Docker Hub. (This means that their build recipe is always available.)

```
$ docker search figlet
NAME                            DESCRIPTION                                     STARS               OF
hairyhenderson/figlet           Figlet in a container                           3
viveknangal/figlet              This is the Ubuntu base Image having figlet ...   1
mbentley/figlet                                                                 0
mwendler/figlet                 a minimal figlet setup                          0
```

Downloading images

- There are two ways to download images
  - Explicitly, with docker pull
  - Implicitly, when executing docker run and the image is not found locally

Image and tags

- Images can have tags
- Tags define image versions or variants
- `docker pull ubuntu` will refer to `ubuntu:latest`
- The `:latest` tag is generally updated often

docker build

Dockerfiles for building new images

Dockerfiles are a set of instructions on how to add things to a base image. They build custom images up in a series of layers. In a new file called `Dockerfile`, put the following:

```
FROM rocker_limma:latest
```

This tells Docker to start with the `rocker_limma` base image - that's what we've been using so far. The `FROM` command must always be the first thing in your Dockerfile; this is the bottom crust of the pie we are baking.

Next, let's add another layer on top of our base, in order to have gapminder pre-installed and ready to go:

```
RUN R -e "install.packages('gapminder', repos = 'http://cran.us.r-project.org')"
```

`RUN` commands in your Dockerfile execute shell commands to build up your image, like putting the filling in our pie. In this example, we install `gapminder` from the command line using `install.packages`, which does the same thing as if we had done `install.packages('gapminder')` from within RStudio. Save your Dockerfile, and return to your docker terminal; we can now build our image by doing:

```
docker build -t my-r-image .
```

`-t my-r-image` gives our image a name (note image names are always all lower case), and the `.` says all the resources we need to build this image are in our current directory. List your images via:

```
docker images
```

and you should see `my-r-image` in the list. Launch your new image similarly to how we launched the base image:

```
docker run --rm -p 8787:8787 my-r-image
```

Then in the RStudio terminal, try gapminder again:

```
library('gapminder')
gapminder
```

And there it is - `gapminder` is pre-installed and ready to go in your new docker image.

Our pie is almost complete! All we need to finish it is the topping. In addition to R packages like `gapminder`, we may also want some some static files inside our Docker image - such as data. We can do this using the `ADD` command in your Dockerfile:

```
ADD data/gapminder-FiveYearData.csv /home/rstudio/
```

Rebuild your Docker image:

```
docker build -t my-r-image .
```

And launch it again. Go back to RStudio in the browser, and there `gapminder-FiveYearData.csv` will be, present in the files visible to RStudio. In this way, we can capture files as part of our Docker image, so they're always available along with the rest of our image in the exact same state.

Use `ADD` command to add analysis R scripts

What this Dockerfile will do?

```
FROM ubuntu
RUN apt-get update
RUN apt-get install figlet
CMD figlet -f script hello
```

- `FROM` indicates the base image for our build
- Each `RUN` line will be executed by Docker during the build
- Our `RUN` commands must be non-interactive. (No input can be provided to Docker during the build.)
- In many cases, we will add the `-y` flag to `apt-get`
- `CMD` defines a default command to run when none is given
- It can appear at any point in the file
- Each `CMD` will replace and override the previous one. As a result, while you can have multiple `CMD` lines, it is useless

Other useful commands for building images

- `COPY` - will copy the source file in the container, e.g.

  ```
  FROM ubuntu
  RUN apt-get update
  RUN apt-get install -y build-essential
  COPY hello.c /
  RUN make hello
  CMD /hello
  ```

- You can `COPY` whole directories recursively
- Volumes are special directories in a container. Volumes act as passthroughs to the host filesystem
- Declared as `VOLUME /var/lib/postgresql` of using `-v` switch in the command line
- When you `docker commit`, the content of volumes is not brought into the resulting image
- If a RUN instruction in a Dockerfile changes the content of a volume, those changes are not recorded neither
- Volumes can be shared across containers
- Volumes exist independently of containers
- The `docker history <image>` command lists all the layers composing an image
  - For each layer, it shows its creation time, size, and creation command
  - When an image was built with a Dockerfile, each layer corresponds to a line of the Dockerfile
- You can rename containers with `docker rename`

What's not covered

- Networking
- Connecting containers

Webapps with Docker

https://github.com/odewahn/docker-jumpstart/blob/master/public/example.md

https://docker-curriculum.com/ - 2.0 Webapps with Docker

Useful commands

- List all containers (only IDs): `docker ps -aq`
- Stop all running containers: `docker stop $(docker ps -aq)`
- Remove all containers: `docker rm $(docker ps -aq)`
- Remove all images: `docker rmi $(docker images -q)`
- More cleanup tips: https://codefresh.io/docker-tutorial/everyday-hacks-docker/