Hands On with Docker

Docker Containre

0. Installing Docker

$ sudo apt-get update

$ sudo apt-get install ca-certificates curl gnupg lsb-release

$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg

$ echo \

"deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] https://download.docker.com/linux/ubuntu \

$(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

$ sudo apt-get update

$ sudo apt-get install docker-ce docker-ce-cli containerd.io

1. Running your first container

Now that you have everything setup, it's time to get our hands dirty. In this section, you are going to run an Alpine Linux container (a lightweight linux distribution) on your system and get a taste of the `docker run` command.

To get started, let's run the following in our terminal:

```
$ docker pull alpine
```

> **Note:** Depending on how you've installed docker on your system, you might see a `permission denied` error after running the above command. Try the commands from the Getting Started tutorial to verify your installation. If you're on Linux, you may need to prefix your `docker` commands with `sudo`. Alternatively you can create a docker group to get rid of this issue.

The `pull` command fetches the alpine **image** from the **Docker registry** and saves it in our system. You can use the `docker images` command to see a list of all images on your system.

```
$ docker images
REPOSITORY              TAG                 IMAGE ID            CREATED
VIRTUAL SIZE
alpine                  latest              c51f86c28340        4 weeks ago
1.109 MB
hello-world             latest              690ed74de00f        5 months ago
960 B
```

1.1 Docker Run

Great! Let's now run a Docker **container** based on this image. To do that you are going to use the `docker run` command.

```
$ docker run alpine ls -l
total 48
drwxr-xr-x    2 root     root          4096 Mar  2 16:20 bin
drwxr-xr-x    5 root     root           360 Mar 18 09:47 dev
drwxr-xr-x   13 root     root          4096 Mar 18 09:47 etc
drwxr-xr-x    2 root     root          4096 Mar  2 16:20 home
drwxr-xr-x    5 root     root          4096 Mar  2 16:20 lib
......
......
```

What happened? Behind the scenes, a lot of stuff happened. When you call `run`, 1. The Docker client contacts the Docker daemon 2. The Docker daemon checks local store if the image (alpine in this case) is available locally, and if not, downloads it from Docker Store. (Since we have issued `docker pull alpine` before, the download step is not necessary) 3. The Docker daemon creates the container and then runs a command in that container. 4. The Docker daemon streams the output of the command to the Docker client

When you run `docker run alpine`, you provided a command (`ls -l`), so Docker started the command specified and you saw the listing.

Let's try something more exciting.

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

OK, that's some actual output. In this case, the Docker client dutifully ran the `echo` command in our alpine container and then exited it. If you've noticed, all of that happened pretty quickly. Imagine booting up a virtual machine, running a command and then killing it. Now you know why they say containers are fast!

Try another command.

```
$ docker run alpine /bin/sh
```

Wait, nothing happened! Is that a bug? Well, no. These interactive shells will exit after running any scripted commands, unless they are run in an interactive terminal - so for this example to not exit, you need to `docker run -it alpine /bin/sh`.

You are now inside the container shell and you can try out a few commands like `ls -l`, `uname -a` and others. Exit out of the container by giving the `exit` command.

Ok, now it's time to see the `docker ps` command. The `docker ps` command shows you all containers that are currently running.

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES
```

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

```
$ docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS                      PORTS               NAMES
36171a5da744        alpine              "/bin/sh"                5 minutes
ago       Exited (0) 2 minutes ago                       fervent_newton
a6a9d46d0b2f        alpine             "echo 'hello from alp"    6 minutes
ago       Exited (0) 6 minutes ago                       lonely_kilby
ff0a5c3750b9        alpine             "ls -l"                  8 minutes
ago       Exited (0) 8 minutes ago                       elated_ramanujan
c317d0a9e3d2        hello-world         "/hello"                34 seconds
ago      Exited (0) 12 minutes ago                      stupefied_mcclintock
```

What you see above is a list of all containers that you ran. Notice that the `STATUS` column shows that these containers exited a few minutes ago. You're probably wondering if there is a way to run more than just one command in a container. Let's try that now:

```
$ docker run -it alpine /bin/sh
/ # ls
bin      dev      etc      home     lib      linuxrc  media    mnt      proc
root     run      sbin     sys      tmp      usr      var
/ # uname -a
Linux 97916e8cb5dc 4.4.27-moby #1 SMP Wed Oct 26 14:01:48 UTC 2016 x86_64
Linux
```

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

That concludes a whirlwind tour of the `docker run` command which would most likely be the command you'll use most often. It makes sense to spend some time getting comfortable with it. To find out more about `run`, use `docker run --help` to see a list of all flags it supports. As you proceed further, we'll see a few more variants of `docker run`. ### 1.2 Terminology In the last section, you saw a lot of Docker-specific jargon which might be confusing to some. So before you go further, let's clarify some terminology that is used frequently in the Docker ecosystem.

- *Images* - The file system and configuration of our application which are used to create containers. To find out more about a Docker image, run `docker inspect alpine`. In the demo above, you used the `docker pull` command to download the **alpine** image. When you executed the command `docker run hello-world`, it also did a `docker pull` behind the scenes to download the **hello-world** image.
- *Containers* - Running instances of Docker images — containers run the actual applications. A container includes an application and all of its dependencies. It shares the kernel with other containers, and runs as an isolated process in user space on the host OS. You created a container using `docker run` which you did using the alpine image that you downloaded. A list of running containers can be seen using the `docker ps` command.

- *Docker daemon* - The background service running on the host that manages building, running and distributing Docker containers.
- *Docker client* - The command line tool that allows the user to interact with the Docker daemon.
- *Docker Store* - A registry of Docker images, where you can find trusted and enterprise ready containers, plugins, and Docker editions. You'll be using this later in this tutorial.

Next Steps

For the next step in the tutorial, head over to Webapps with Docker