USING DOCKER CONTAINERS

Getting assistance on Docker command line

- All the examples that follow in this course are written using the Docker command line. That might be a challenge no matter how experienced you are in Linux command line.
- Because of that, Docker provides a bundled help system within its command line interface.
- You can use it whenever you need assistance by typing the command followed by `--help`. For example, `docker images --help`
- If you need help on Docker itself you can type `docker --help`, which will list all the available commands with Docker.

LAB: create a service monitor

- Target: we have a server running an a Busybox image(a tiny Linux kernel with the most basic functionality). We need to check whether or not this server is up and running on periodic intervals.
- Procedure:

– Pull and run `afakharany/server` container from Docker hub. The container needs to be run in the background (daemon mode) and it needs to have a name by which it can be referenced from other containers. We'll name it "server". The command is as follows:
`docker run --detach --name server afakharany/server`

– Pull and run `afakharany/checker` container from Docker hub. This container contains a script that issues a ping request to the server every five seconds. If the server is running it will print "Server is up", otherwise it'll print "Server is down". But to do that, you must "link" containers with each other. Consider the following:
`docker run --link server:server -it afakharany/checker`
This command pulls and runs `afakharany/checker` container. It also links it to the already running container "server". Notice the use of -it options (interactive and tty), which makes the container run in the foreground, and accept tty signals (like CTRL-c to exit). As soon as the command runs it should print "Server is running".

– Now press CTRL-c to exit the container, and run the container in the background mode:
`docker run --detach --name checker --link server:server afakharany/checker`
Accordingly the container will keep running in the background doing its job.

Listing the running containers

- Once one or more containers are up and running, you can use docker ps to list all the currently running containers. The following information is displayed:
  - *The ID of the container*
  - *The container's name*
  - *The image used*
  - *The command that the container is executing*
  - *The amount of time elapsed since the container started running*
  - *Any network ports exposed by the container*

More ways to interact with containers

- After running a Docker container, you can control its state by issuing the following commands:
  - `docker stop` *`name/id`* ***this will stop the container***
  - `docker start` *`name/id`* ***this will start a stopped container***
  - `docker restart` *`name/id`* ***this will restart (stop then start) a container.***
- When a container is up and running in the background, you won't see any output of its command(s). If you want to examine this output, you can use the docker logs command followed by the container name or id. For example: `docker logs server`
- One useful option here is –f `docker logs -f server`. This will continuously update the output of the logs command, displaying new content as soon as it is available.
- For example, if you stop the "server" container using docker stop server, the checker container will start sensing that the server is down as no ping requests are successful anymore. Accordingly it will start printing "Server is down". Such a message can be viewed with `docker logs checker`.

The PID namespace

- In Linux, each and every process running on the system must have a unique number identifying it called the process ID or the PID. Process ID's start from 1 (often assigned to the init process) and increment as more and more processes are spawned by the system.
- In the following example, we are creating two different containers based on the Busybox image:
```
docker run -d --name server1 busybox sh -c "nc -l -p 0.0.0.0:7070"
docker run -d --name server1 busybox sh -c "nc -l -p 0.0.0.0:8080"
```
- Now lets see what processes does each container posses by running the ps command on each of them. You can run additional commands against the container in addition to the main command that it runs by using Docker's exec command:
docker exec command as follows:
```
root@DockerDemo:~# docker exec server1 ps
PID   USER     TIME   COMMAND
    1 root       0:00 sh -c nc -l -p 0.0.0.0:7070
    5 root       0:00 nc -l -p 0.0.0.0:7070
   10 root       0:00 ps
root@DockerDemo:~# docker exec server2 ps
PID   USER     TIME   COMMAND
    1 root       0:00 sh -c nc -l -p 0.0.0.0:8080
    5 root       0:00 nc -l -p 0.0.0.0:8080
   10 root       0:00 ps
```
- As you can see from the output, each container has its own PID namespace, which makes processes start from 1 onwards as if it is a completely separate OS.
- Such an isolation level is one of the strength points Docker; as it prevents the container from guessing the PID's of the host operating system.
- You can manually override this behavior by adding `--pid host` to the `docker create` or `docker run` commands.

The problem with multiple containers

- One of the most challenging issues in development environments is the need to create more than one server (service) all using the same resources. For example, say you are developing a web application, and you need to create more than one webserver to test different environments. Once the first webserver is up and running, port 80 can no longer be used by another webserver process.
- Docker provides a number of workarounds for this problem, which are similar to namespace isolation.
- But building more and more containers for the same purpose may lead to naming conflicts, which may cause difficulty identifying the right container when needed. For example, think of tens of webservers, using the --name flag to specify a human-friendly name for the container becomes a challenge quickly.
- In the following slides let's explore the mechanism by which Docker identifies its containers and how you can make use of it for your own needs.

Container ID's

- You've previously seen the container ID appearing in the output of docker images. Actually, this is not the whole ID, this is the first twelve characters of it. A container id consists of 1024 hexadecimal-encoded numbers resulting in a combination of sixty five letters and numbers.
- Because of the large length of this ID, it is very less likely that a collision may occur (two containers having the same ID). Actually, it is less likely that the first twelve characters will be repeated. Accordingly, Docker uses those first twelve characters as a "shorthand" for the full container ID.
- You can obtain the full container ID when you run it in a detached mode (background), as Docker responds with the container ID. For example:

```
root@DockerDemo:~# docker run -d busybox
a833efa0c99f5a6e9ecc883a7f6a0823b8bae86a68d7910c98012927524bda7b
```

- You can also let Docker generate the container ID for you without having to actually run the container by using the docker create command:

```
root@DockerDemo:~# docker create
busybox1ec198ff028a3a5f98f93b88adc57418715c9e53121bba0ac7a724a068f3a0
32
```

- You can use either of the two ID's to refer to a container. For example

```
root@DockerDemo:~# docker start
1ec198ff028a3a5f98f93b88adc57418715c9e53121bba0ac7a724a068f3a032
```

More ways to get the container ID

- You can save the container ID to a shell variable. But note that this method is only valid if you are working in a POSIX-compatible shells (like BASH for example):

```
root@DockerDemo:~# CID=`docker create busybox`
root@DockerDemo:~# echo $CID
35f5c754723a112ecc53fde4294d8e57ee8f0fac6a769506510000064389d7
75
```

- Another more elegant way of storing container ID's is using the --cidfile flag. This flag is available for docker create and docker run commands. For example:

```
root@DockerDemo:~# docker create --cidfile bbox.cide busybox
a293fed72141060a2f62c88bb55b2979dbb6b079e5f6985c71a28293a7f2e4
53
root@DockerDemo:~# cat bbox.cide
a293fed72141060a2f62c88bb55b2979dbb6b079e5f6985c71a28293a7f2e4
53
```

- Notice that Docker will fail to create a new container when the --cidfile is specified if that file already exists.

Docker human-friendly names

- Naturally, container ID's are hard to remember and are hard to read. Also having to manually issue a new name for each container may be a headache.
- Consequently, docker uses a naming convention to assign names to new containers. This consists of:
  an adjective + _ + the last name of a famous scientist/engineer
  Examples: kickass_heisenberg, naughty_babbage and so on
- Combining the name you assigned to the container, the auto-generated name, and the CID helps identify containers.

Creating neutral systems

- Systems depend – one way or another – on other systems or on the host.
- Examples of this dependence include known filesystem locations, environment variables, and so on.
- It is a best practice that you minimize those dependencies as much as possible in order to create systems that are easily portable and require the least amount of maintenance.
- Docker provides three methods for achieving this:
  - *Read-only filesystems*
  - *Environment variable injection*
  - *Volumes*

Read-only filesystems

- As the name suggests, a read-only filesystem will prevent any changes to the underlying system.
- This is helpful when dealing with applications that use databases massively to store and manage their data. For example, content management systems (WordPress, Joomla, Drupal…etc.)
- It also helps provide a level of protection to the container: if it gets compromised, the attacker will not have a chance to do any damage.
- You can specify that a container will work in read-only mode by using the --read-only flag. As a matter of fact, Docker does have a ready-made image for WordPress:

```
docker run –d –-read-only --name wpress wordpress
```

LAB: create a WordPress container

- The previous command is enough to create a container from the WordPress public image. Let's see whether or not the instance is functioning properly:
  `docker inspect -format="{{.State.Running}}" wpress`
  The inspect command is used to print a JSON-formatted string containing the container's metadata. The format flag is used to filter that output. In our case, it displays the value of the state of the container.
- The inspect command shows a running state of `false`, which means that our WordPress instance is not running. To determine the reason, run `docker logs wordpress`, which shows a missing dependency for WordPress, which is the MySQL database. Let's create a container for the database:
  `docker run -d --name wpressDB -e MYSQL_ROOT_PASSWORD=admin mysql:latest`
- Now we need to create a new container for WordPress, and link it to the running MySQL database (you may have to delete the first wordpress container before continuing):
  `docker run --name wpress --link wpressDB:mysql -p 80 --read-only wordpress`
- Inspecting the container again proves that it is still not running. That's because although WordPress uses the database to store most of its data, it still needs to make some changes to the underlying filesystems. Now remove the old wordpress instance and create a new one, this time we are using volumes (covered later) to exempt some filesystems from the read-only protection:
  `docker run -d --name wpress --link wpressDB:mysql -p 80 --read-only -v /run/lock/apache2 -v /run/apache2 -v /tmp wordpress`
- To test that everything is working, check the container status and running port by running `docker ps`, then open your browser and navigate to http://localhost:*port*. You should find WordPress startup page that will allow you to start customization. You have a fully functional WordPress installation.

Bundling environment variables

- In the previous lab, WordPress was created successfully. But if you've worked with WordPress before, you'd know that a lot of variables and customization settings are done outside the browser in wp-config.php file. The previous method assumed default values for those environment variables. But Docker can let you pass on those variables in the container-creation process itself.
- Environment variables are a core part of an operating system. One of the well-known variables is PATH. In Linux/UNIX you can examine the currently defined variables by issuing `env` command. In other operating systems the method may differ but the concept is the same.
- Using Docker you can define a new environment variable, overwrite an existing one, or do both. Any scripts or applications inside your containers can be configured to expect and interpret the variables that you pass to them.
- You inject environment variables to a container using the --env (or –e) option. Consider the following example:
  ```
  docker run -it -e MYVAR='Hello' busybox env
  ```
  Executing the env command on the container will display all the configured environment variables, including the newly defined $MAYVAR.
- Accordingly, the previous docker command could have contained some wordpress settings as follows:
  ```
  docker run -d --name wpress --link wpressDB:mysql -p 80 --read-only -v
  /run/lock/apache2 -v /run/apache2 -v /tmp -e WORDPRESS_DB_HOST=database_hostname -e
  WORDPRESS_DB_USER=database_user -e WORDPRSS_DB_PASSWORD=the_password -e
  WORDPRESS_DB_NAME=the_database_name wordpress
  ```

Recovering from failures – restarting

- Any given Docker container can be in one of four states: running, paused, restarting, and exited (also refers to the stopped state).
- As mentioned before, a container will keep on running as long as the command(s) it executes are. When all the commands exits, the container is stopped. Also when the command fails from some reason, the container will exit as well. Docker provides a restart mechanism to help recover the container from failures.
- The `--restart` option can be added at creation time to instruct Docker to restart the container. It has one of the four following:
  - Do not restart at all (the default behavior) - `no`
  - Try to restart when a failure happens – `on-failure[:max-retries]`
  - Always restart the container but do not start it when the daemon starts if it was previously put to the stopped state – `unless-stopped`
  - Always restart the container when it is down – `always`
- Docker uses a back-off policy to avoid restarting the container too quickly. This policy works as follows: the first time the system will wait for a specified period (say one second), the second attempt will have to wait double the period (two seconds), the third will wait four seconds, the fourth will be eight seconds and so on.
- You can check this behavior using the following example:

```
root@DockerDemo:~# docker run -d --name restart-test --restart always busybox date
8b240bc03373096e3d67adc5aaa1ed139e81c1bb3d293a7c296b043843625f59
root@DockerDemo:~# docker logs -f restart-test
Sun Jul 17 20:31:32 UTC 2016
Sun Jul 17 20:31:33 UTC 2016
Sun Jul 17 20:31:36 UTC 2016
```

  As you can see, after the `date` command is done, the container gets restarted. Each time it restarts it prints the current date and time (the output of the `date` command). Observer the incrementing amount of wait periods before Docker attempts to restart the container.

- Notice that when the docker "attempts" to restart the container, the container is in the "restart" state, where it cannot accept any new commands. This may become a problem if you want to run some diagnostic or troubleshooting tools on the container to determine the root cause of the failure. A better approach is to use an init process (process manager) with the container.

The process manager (init)

- The Linux/UNIX system model contains a main process, with the PID of 1. it is always the first process that gets launched when the system boots. It is responsible for launching all other processes on the system, and respawns (restarts) some of them if they fail.
- In a container design, it is a recommended practice to use the same architecture to ensure that processes get cleanly restarted if they failed.
- To see an example, run the following command
  `docker run -d --name baseimage phusion/baseimage`
  which will pull and run a baseimage container from phusion. This is an Ubuntu 16.04 image that uses an init process to control child processes.
- Now have a look at the running processes inside the container by issuing the following command:
  `docker exec baseimage ps –ef`
  to determine the processes currently running on the system.
- Obtain the process id for one of the running processes (say syslog), and kill this process using the kill command like this:
  `docker exec baseimage kill -9 PID`
- Examine the container logs to see what happened by issuing the following command:
  `docker logs baseimage`
  you will see that the process was respawned again by the process manager.

Container's entry point

- Another useful measure taken to ensure – as much as possible – that the container and the processes within run smoothly without errors, is the "startup script".
- A startup script is simply a shell script that does a quick check on the all the prerequisites needed to run the application inside the container.
- For example, we can examine the startup script of the MySQL database container we created earlier by issuing the following command:
  `docker exec wpressDB cat /usr/local/bin/docker-entrypoint.sh`
- Docker also has an "entrypoint" command. This is by default sh –c, which ensures that every command run directly using the docker command is correctly passed on to the shell to get executed. But that can be altered by using the --entrypoint flag. For example:
  `docker run --entrypoint="cat" mysql /usr/local/bin/docker-entrypoint.sh`
  Here I've overridden the default program that gets executed to be "cat" instead of sh –c. Now I can just pass in the path to the file to be displayed instead of the passing in the whole cat command; as cat will get executed by default.

Removing containers

- After working with so many examples in this section, you probably have a large number of containers sitting in your system. Some of them have already exited while others are still running. Using `docker ps -a`, you can have a complete listing with all the containers regardless of their state.
- To remove a container, you have to stop it first. Then you can use the command `docker rm name/id`. But you can override the warning and force a running container to stop by either issuing the command `docker rm -f name/id` or using docker kill name/id. The key difference here is that the docker stop will send a SIGHUP signal to the container, giving the process a chance to make necessary cleanups, while the second method will send a SIGKILL signal, which causes the process to immediately stop while running, which may lead to file corruption and other adverse consequences.
- If you want to run a container for testing purposes after which you are going to immediately remove, you can use the --rm flag while creation to automate the cleanup process. For example:
  `docker run --rm busybox echo "Hello World"`
  This command will print "Hello World" to the screen using a Busybox container. After the container exits, it will get cleanly removed from the system automatically.