Docker Fundamentals Exercises

Contents

1 Running & Inspecting a Container

1.1 Running Containers

1. First, let's start a container, and observe the output:

```
$ docker container run ubuntu:16.04 echo "hello world"
```

The `ubuntu:16.04` part indicates the *image* we want to use to define this container; it includes the underlying operating system and its entire filesystem. `echo "hello world"` is the command we want to execute inside the context of the container defined by that image.

2. Now create another container from the same image, and run a different command inside of it:

```
$ docker container run ubuntu:16.04 ps -ef
```

3. `ps -ef` was PID 1 inside the container in the last step; try doing `ps -ef` at the host prompt and see what process is PID 1 here.

1.2 Listing Containers

1. Try listing all your currently running containers:

```
$ docker container ls
```

There's nothing listed, since the containers you ran executed a single command, and shut down when finished.

2. List stopped as well as running containers with the `-a` flag:

```
$ docker container ls -a
```

1.3 Conclusion

In this exercise you ran your first container using `docker container run`, and explored the importance of the PID 1 process in a container; this process is a member of the host's PID tree like any other, but is 'containerized' via tools like kernel namespaces, making this process and its children behave as if it was the root of a PID tree, with its own user ID spectrum, filesystem, and network stack. Furthermore, this process defines the state of the container; if the process exits, the container stops.

2 Interactive Containers

2.1 Writing to Containers

1. Create a container using the `ubuntu:16.04` image, and connect to its bash shell in interactive mode using the `-i` flag (also the `-t` flag, to request a TTY connection):

```
$ docker container run -it ubuntu:16.04 bash
```

2. Explore your container's filesystem with `ls`, and then create a new file:

```
$ ls
$ touch test.dat
$ ls
```

3. Exit the connection to the container:

```
$ exit
```

4. Run the same command as above to start a container in the same way:

```
$ docker container run -it ubuntu:16.04 bash
```

5. Try finding your `test.dat` file inside this new container; it is nowhere to be found. Exit this container for now in the same way you did above.

2.2 Reconnecting to Containers

1. We'd like to recover the information written to our container in the first example, but starting a new continer didn't get us there; instead, we need to restart our original container, and reconnect to it. List all your stopped containers:


```
$ docker container ls -a
```

2. We can restart a container via the Container ID listed in the first column. Use the container ID for the first `ubuntu:16.04` container you created with `bash` as its command:

```
$ docker container start <Container ID>
```

3. Reconnect to your container with `docker container exec` (this can be used to execute *any* command in a running Docker container):

```
$ docker container exec -it <Container ID> bash
```

4. List the contents of the container's filesystem again with `ls`; your `test.dat` should be where you left it. Exit the container again by typing `exit`.

2.3 More Container Listing Options

1. In the last step, we saw how to get the short container ID of all our containers using `docker container ls -a`. Try adding the `--no-trunc` flag to see the entire container ID:

```
$ docker container ls -a --no-trunc
```

This long ID is the same as the string that is returned after starting a container with `docker container run`.

2. List only the container ID:

```
$ docker container ls -aq
```

3. List the last container to have started:

```
$ docker container ls -l
```

4. Finally, you can also filter results with the `--filter` flag; for example, try filtering by exit code:

```
$ docker container ls -a --filter "exited=1"
$ docker container ls -a --filter "exited=0"
```

2.4 Conclusion

In this demo, you saw that files added to a container's filesystem do not get added to all containers created from the same image; changes to a container's filesystem are local to itself, and exist only in that particular container's R/W layer, of which there is one per container. You also learned how to restart a stopped Docker container using `docker container start`, how to run a command in a running container using `docker container exec`, and also saw some more options for listing containers via `docker container ls`.

3 Detached Containers and Logging

3.1 Running a Container in the Background

1. First try running a container as usual; the STDOUT and STDERR streams from whatever is PID 1 inside the container is directed to the terminal:

```
$ docker container run ubuntu:14.04 ping 127.0.0.1 -c 10
```

2. The same process can be run in the background with the `-d` flag:

```
$ docker container run -d ubuntu:14.04 ping 127.0.0.1
```

3. Find this second container's ID, and use it to inspect the logs it generated:


```
$ docker container logs <container ID>
```

These logs correspond to STDOUT and STDERR from the container's PID 1.

3.2 Attaching to Container Output

1. We can attach a terminal to a container's PID 1 output with the `attach` command; try it with the last container you made in the previous step:

   ```
   $ docker container attach <container ID>
   ```

2. We can leave attached mode by then pressing `CTRL+C`. After doing so, list your running containers; you should see that the container you attached to has been killed, since the `CTRL+C` issued killed PID 1 in the container, and therefore the container itself.

3. Try running the same thing in detached interactive mode:

   ```
   $ docker container run -d -it ubuntu:14.04 ping 127.0.0.1
   ```

4. Attach to this container like you did the first one, but this time detach with `CTRL+P+Q`, and list your running containers. In this case, the container should still be happily running in the background after detaching from it.

3.3 Logging Options

1. We saw previously how to read the entire log of a container's PID 1; we can also use a couple of flags to control what logs are displayed. `--tail n` limits the display to the last `n` lines; try it with the container that should be running from the last step:

   ```
   $ docker container logs --tail 5 <container ID>
   ```

2. We can also follow the logs as they are generated with `-f`:

   ```
   $ docker container logs -f <containerID>
   ```

   (`CTRL+C` to break out of following mode).

3. Finally, try combining the tail and follow flags to begin following the logs from a point further back in history.

3.4 Conclusion

In this scenario, we saw how to run processes in the background, attach to them, and inspect their logs. We also saw an explicit example of how killing PID 1 in a container kills the container itself.

4 Starting, Stopping, Inspecting and Deleting Containers

4.1 Starting and Restarting Containers

1. Start by running a tomcat server in the background, and check that it's really running:

   ```
   $ docker container run -d tomcat
   $ docker container ls
   ```

2. Stop the container using `docker container stop`, and check that the container is indeed stopped:

   ```
   $ docker container stop <container ID>
   $ docker container ls -a
   ```


3. Start the container again with `docker container start`, and attach to it at the same time:

```
$ docker container start -a <containerID>
```

4. Detach and stop the container with CTRL+C, then restart the container without attaching and follow the logs starting from 10 lines previous.

5. Finally, stop the container with `docker container kill`:

```
$ docker container kill <containerID>
```

Both `stop` and `kill` send a SIGKILL to PID 1 in the container; the difference is that `stop` first sends a SIGTERM, then waits for a grace period (default 10 seconds) before sending the SIGKILL, while `kill` fires the SIGKILL immediately.

4.2 Inspecting a Container with `docker container inspect`

1. Start your tomcat server again, then inspect the container details using `docker container inspect`:

```
$ docker container start <containerID>
$ docker container inspect <container ID>
```

2. Find the container's IP and long ID in the JSON output of `inspect`. If you know the key name of the property you're looking for, try piping to grep:

```
$ docker container inspect <container ID> | grep IPAddress
```

3. Now try grepping for `Cmd`, the PID 1 command being run by this container. `grep`'s simple text search doesn't always return helpful results.

4. Another way to filter this JSON is with the `--format` flag. Syntax follows Go's text/template package: http://golang.org/pkg/text/template/. For example, to find the `Cmd` value we tried to grep for above, instead try:

```
$ docker container inspect --format='{{.Config.Cmd}}' <container ID>
```

5. Keys nested in the JSON returned by `docker container inspect` can be chained together in this fashion. Try modifying this example to return the IP address you grepped for previously.

6. Finally, we can extract all the key/value pairs for a given object using the `json` function:

```
$ docker container inspect --format='{{json .Config}}' <container ID>
```

4.3 Deleting Containers

1. Start three containers in background mode, then stop the first one.

2. List only exited containers using the `--filter` flag we learned earlier, and the option `status=exited`.

3. Delete the container you stopped above with `docker container rm`, and do the same listing operation as above to confirm that it has been removed:

```
$ docker container rm <container ID>
$ docker container ls ...
```

4. Now do the same to one of the containers that's still running; notice `docker container rm` won't delete a container that's still running, unless we pass it the force flag `-f`. Delete the second container you started above:

```
$ docker container rm -f <container ID>
```

5. Try using the `docker container ls` flags we learned previously to remove the last container that was run, or all stopped containers. Recall that you can pass the output of one shell command `cmd-A` into a variable of another command `cmd-B` with syntax like `cmd-B $(cmd-A)`.

4.4 Conclusion

In this scenario, you learned how to use `docker container start`, `stop`, `rm` and `kill` to start, stop and delete containers. You also saw the `docker container inspect` command, which returns metadata about a given container.

5 Interactive Image Creation

5.1 Modifying a Container

1. Start a bash terminal in an ubuntu container:

```
$ docker container run -it ubuntu:16.04 bash
```

2. Install a couple pieces of software in this container - there's nothing special about `vim` and `wget`, any changes to the filesystem will do. Afterwards, exit the container:

```
$ apt-get update
$ apt-get install -y wget vim
$ exit
```

3. Finally, try `docker container diff` to see what's changed about a container relative to its image; you'll need to get the container ID via `docker container ls -a` first:

```
$ docker container ls -a
$ docker container diff <container id>
```

   Make sure the results of the diff make sense to you before moving on.

5.2 Capturing Container State as an Image with `docker container commit`

1. Installing wget and vim in the last step wrote information to the container's read/write layer; now let's save that read/write layer as a new read-only image layer in order to create a new image that reflects our additions, via the `docker container commit`:

```
$ docker container commit <container id> <your docker store ID>/myapp:1.0
```

2. Check that you can see your new image by listing all your images:

```
$ docker image ls
```

3. Create a container running bash using your new image, and check that vim and wget are installed:

```
$ docker container run -it <your docker store ID>/myapp:1.0 bash
$ which vim
$ which wget
```

4. Create a file in your container and commit that as a new image. Use the same image name but tag it as 1.1.
5. Finally, run `docker container diff` on your most recent container; does the output make sense? What do you guess the prefixes A, C and D at the start of each line mean?

5.3 Conclusion

In this exercise, you saw how to inspect the contents of a container's read / write later with `docker container diff`, and commit those changes to a new image layer with `docker container commit`.

6 Creating Images with Dockerfiles (1/2)

6.1 Writing and Building a Dockerfile

1. Create a folder called `myimage`, and a text file called `Dockerfile` within that folder. In `Dockerfile`, include the following instructions:

```
FROM ubuntu:16.04

RUN apt-get update
RUN apt-get install -y iputils-ping
```

2. Build your image with the `build` command:

```
$ docker image build -t <username>/myimage .
```

3. Verify that your new image exists with `docker image ls`, then use it to run a container and ping something from within that container.

6.2 The Build Cache

1. In order to speed up image builds, Docker preserves a cache of previous build steps. Try running the exact same build command you did previously:

```
$ docker image build -t <username>/myimage .
```

   Each step should execute immediately and report `using cache` to indicate that the step was not repeated, but fetched from the build cache.

2. Open your Dockerfile and add another `RUN` step at the end to install `vim`.
3. Build the image again as above; which steps is the cache used for?
4. Build the image again; which steps use the cache this time?
5. Finally, swap the order of the two `RUN` commands for installing `ping` and `vim` in the Dockerfile, and build one last time. Which steps are cached this time?

6.3 The `history` Command

1. The `docker image history` command allows us to inspect the build cache history of an image. Try it with your new image:

```
$ docker image history <image id>
```

   Note the image id of the layer built for the `apt-get update` command.

2. Replace the two `RUN` commands that installed `iputils-ping` and `vim` with a single command:

```
...
RUN apt-get install -y iputils-ping vim
```

3. Build the image again, and run `docker image history` on this new image. How has the history changed?

6.4 Conclusion

So far, we've seen how to write a basic Dockerfile using `FROM` and `RUN` commands, some basics of how image chaching works, and seen the `docker image history` command. After some discussion, we'll see how to define some defualt commands and options for running as the PID 1 of containers created from an image defined by our Dockerfile.

7 Creating Images with Dockerfiles (2/2)

7.1 Default Commands via CMD

1. Add the following line to your Dockerfile from the last problem, at the bottom:

```
...
CMD ["ping", "127.0.0.1", "-c", "30"]
```

   This sets ping as the default command to run in a container created from this image, and also sets some parameters for that command.

2. Rebuild your image:

```
$ docker image build -t <username>/myimage:1.0 .
```

3. Run a container from your new image with no command provided:

```
$ docker container run <username>/myimage:1.0
```

4. You should see the command provided by the CMD parameter in the Dockerfile running. Try explicitly providing a command when running a container:

```
$ docker container run <username>/myimage:1.0 echo "hello world"
```

   Providing a command in docker container run overrides the command defined by CMD.

7.2 Default Commands via ENTRYPOINT

1. Replace the CMD instruction in your Dockerfile with an ENTRYPOINT:

```
...
ENTRYPOINT ["ping"]
```

2. Build the image and use it to run a container with no process arguments:

```
$ docker image build -t <username>/myimage:1.0 .
$ docker container run <username>/myimage:1.0
```

   What went wrong?

3. Try running with an argument after the image name:

```
$ docker container run <username>/myimage:1.0 127.0.0.1
```

   Tokens provided after an image name are sent as arguments to the command specified by ENTRYPOINT.

7.3 CMD and ENTRYPOINT Together

1. Open your Dockerfile and modify the ENTRYPOINT instruction to include 2 arguments for the ping command:

```
...
ENTRYPOINT ["ping", "-c", "3"]
```

2. If CMD and ENTRYPOINT are both specified in a Dockerfile, tokens listed in CMD are used as default parameters for the ENTRYPOINT command. Add a CMD with a default IP to ping:

```
...
CMD ["127.0.0.1"]
```

3. Build the image and run a container with a public IP as an argument to docker container run:

```
$ docker container run <username>/myimage:1.0 <some public ip>
```

4. Run another container without any arguments after the image name. Explain the difference in behavior between these two last containers.

7.4 Conclusion

In this exercise, we encountered the Dockerfile commands `CMD` and `ENTRYPOINT`. These are useful for defining the default process to run as PID 1 inside the container right in the Dockerfile, making our containers more like executables and adding clarity to exactly what process was meant to run in a given image's containers.

8 Dockerizing an Application

8.1 Setting up a Java App

For an alternative way of doing this see below (Optional Exercise)

1. Install Java 8:

```
$ sudo apt-get install openjdk-8-jdk
```

2. Create a directory called `javahelloworld`; in that directory, make a file called `HelloWorld.java`, containing the following code:

```
public class HelloWorld
{
    public static void main (String [] args)
    {
        System.out.println("hello world");
    }
}
```

3. Compile and run your new application:

```
$ javac HelloWorld.java
$ java HelloWorld
```

If all's gone well, you have a hello world program running in Java; next, we'd like to containerize this application by capturing its environment in an image described by a Dockerfile.

8.2 Dockerize your App

1. In your `javahelloworld` folder, create a Dockerfile that bases its image off of the `java:8` base image:

```
FROM java:8
```

2. Add your source code into your image using a new Dockerfile command, `COPY`:

```
...
COPY HelloWorld.java /
```

`COPY`'s syntax is `COPY <target> <destination>`, which copies files from the build context into the image. If `<target>` is a directory, all the contents of that directory will be copied to `<destination>`.

3. Compile your app in the image by appending to your Dockerfile:

```
...
RUN javac HelloWorld.java
```

4. Use `ENTRYPOINT` to run your application automatically when a container is launched from this image:

```
...
ENTRYPOINT ["java", "HelloWorld"]
```

5. Build the image, use it to run a container running the default `ENTRYPOINT` command, and observe the output.

8.3 Restructure your Application

1. After running a container from your image with the default command, try overriding the `ENTRYPOINT` command with `--entrypoint`; we'll run `bash`, so we can explore our containerized environment interactively with a bash shell:

```
$ docker container run -it --entrypoint bash <your java image ID>
```

2. Find where the `HelloWorld.java` source is, and where the compiled binary is. We can restructure our app to be a little more organized as follows; back on your host machine under the `javahelloworld` directory, make two subdirectories `src` and `bin`. Put your java source into this `src` directory, and recompile so the binary ends up in the `bin` directory:

```
$ javac -d bin src/HelloWorld.java
```

3. Run the application again with `java -cp bin HelloWorld` to make sure everything is still working; if it is, we're ready to modify our Dockerfile to reflect this organizational structure in our image.
4. Modify the `COPY` instruction to copy all files in the `src` folder on your host into `/home/root/javahelloworld/src` in your image:

```
...
COPY src /home/root/javahelloworld/src
```

5. Modify the `RUN` instruction to compile the code by referencing the correct `src` folder and to place the compiled code into the `bin` folder:

```
...
RUN  javac -d bin src/HelloWorld.java
```

6. Modify `ENTRYPOINT` to specify `java -cp bin` :

```
...
ENTRYPOINT ["java", "-cp", "bin", "HelloWorld"]
```

7. Try building an running your image now; you should get an error since your source code is now living under `/home/root/javahelloworld/src`, and the `RUN` command told the compiler to go looking under `./src`. By default, commands are run at the root of the image's filesystem, but we can modify this with the `WORKDIR` Dockerfile instruction, which sets the position in the filesystem for all subsequent commands.
8. Before the `RUN` instruction, add:

```
...
WORKDIR /home/root/javahelloworld
```

9. Try building your image and running the container with the default `ENTRYPOINT` command again. There's still an error, but a different one; modify your Dockerfile to fix this, rebuild your image and verify that everything is working correctly again.
10. Finally, override the default `ENTRYPOINT` command and run `bash` instead, like we did above. Verify that the directory structure you described in your Dockerfile is reflected in your image's filesystem.

8.4 Conclusion

In this exercise, you dockerized a simple application, and learned how to manipulate and navigate an image's filesystem with the Dockerfile commands `COPY` and `WORKDIR`. With some practice, writing a Dockerfile that builds up an application image follows a very similar pattern to setting up the app natively, expressing things with `RUN`, `COPY`, `WORKDIR` and `ENTRYPOINT` commands.

8.5 Optional Exercise

For an alternative way to create the Java application without having to install Java on the host you can use a Java container instead.

1. Run a Java container in interactive mode:

```
$ docker container run --rm -it java:8 bash
```

2. Install an editor inside the container, either VIM or nano

```
$ apt-get update && apt-get install -y vim
# or install nano
# apt-get update && apt-get install -y nano
```

3. Now still inside the container create a directory `javahelloworld` and create the file `HelloWorld.java` in it using the editor you just installed, e.g. with `vim` that would be:

```
$ mkdir javahelloworld
$ cd javahelloworld
$ vim HelloWorld.java
```

   Add the code to this file, save and quit the editor.

4. Still inside the Java container compile and run the application

```
$ javac HelloWorld.java
$ java HelloWorld
```

9 Managing Images

9.1 Tagging and Listing Images

1. Download the `ubuntu:16.04` image from Docker Store:

```
$ docker image pull ubuntu:16.04
```

2. Make a new tag of this image:

```
$ docker image tag ubuntu:16.04 my-ubuntu:dev
```

3. List your images:

```
$ docker image ls
```

4. You should have `ubuntu:16.04` and `my-ubuntu:dev` both listed, but they ought to have the same hash under image ID, since they're actually the same image.

9.2 Sharing Images on Docker Store

1. Next, let's share our image on Docker Store. If you don't already have an account there, head over to store.docker.com and make one.

2. Push your image to Docker Store:

```
$ docker image push my-ubuntu:dev
```

   You should get an `authentication required` error.

3. Login by doing `docker login`, and try pushing again. The push fails again because we haven't namespaced our image correctly for distribution on Docker Store; all images you want to share on Docker Store must be named like `<your Docker Store username>/<repo name>[:<optional tag>]`.

4. Retag your image to be namespaced properly, and push again:

```
$ docker image tag my-ubuntu:dev <docker store username>/my-ubuntu:dev
$ docker image push <docker store username>/my-ubuntu:dev
```


5. Visit your Docker Store page, find your new `my-ubuntu` repo, and confirm that you can see the `:dev` tag therein. Explore your new repo, and fill out the description and other fields you find there.
6. Next, write a Dockerfile that uses `<docker store username>/my-ubuntu:dev` as its base image, and installs any application you like on top of that. Build the image, and simultaneously tag it as `:1.0`:

```
$ docker image build -t <docker store username>/my-ubuntu:1.0 .
```

7. Push your `:1.0` tag to Docker Store, and confirm you can see it in the appropriate repo.
8. Finally, list the images currently on your node with `docker image ls`. You should still have the version of your image that wasn't namespaced with your Docker Store user name; delete this using `docker image rm`:

```
$ docker image rm my-ubuntu:dev
```

9.3 Private Images

1. Explore the UI on your Docker Store repo `<username>/my-ubuntu`, and find the toggle to make that repo private.
2. Pair up with someone sitting next to you, and try to pull their image:

```
$ docker image pull <partners docker store name>/my-ubuntu:1.0
```

   Their private repo should be invisible to you at this time.
3. Add each other as Collaborators to your `my-ubuntu` repo, and pull again; if all has gone well, you should now be able to pull their repo. Also check to see that you can see each other's repo on your 'Repositories' tab of your Docker Store profile.

9.4 Conclusion

In this exercise, we saw how to name and tag images with `docker image tag`; explored the correct naming conventions necessary for pushing images to your Docker Store account with `docker image push`; learned how to remove old images with `docker image rm`; and learned how to make repos private and share them with collaborators on Docker Store.

10 Cleanup Commands

Once we start to use Docker heavily on our system we want to understand what resouces the Docker engine is using. We also want ways on how we can cleanup and free unused resources. For all this we can use the `docker system` commands.

1. Find out how much memory Docker is using by executing:

```
$ docker system df
```

   The output will show us how much space images, containers and (local) volumes are occupying and how much of this space can be reclaimed.
2. Reclaim all reclaimable space by using the following command:

```
$ docker system prune
```

   Answer with y when asked if we really want to remove all unused networks, containers, images and volumes.
3. This is the most radical way of keeping our system clean. If we want to be more focused and only cleanup resources of a given type we can use the following commands:


```
$ docker image prune
$ docker container prune
$ docker volume prune
$ docker network prune
```

Try each of them out. If you are tired of answering y each time you can add the flag `-f` or `--force` to the command, e.g.:

```
$ docker system prune --force
```

This is especially useful if you want to run the above commands as part or an automation script.

11 Inspection Commands

11.1 System Information

1. We can find the `info` command under `system`. Execute:

   ```
   $ docker system info
   ```

2. From the output of the last command, identify:
   - how many images are cached on your machine?
   - how many containers are running or stopped?
   - what version of containerd are you running?
   - whether Docker is running in swarm mode?

11.2 System Events

1. There is another powerful system command that allows us to monitor what's happening on the Docker host. Execute the following command:

   ```
   $ docker system events
   ```

   Please note that it looks like the system is hanging, but that is not the case. The system is just waiting for some events to happen.

2. Open a second terminal and execute the following command:

   ```
   $ docker container run --rm alpine echo 'Hello World!'
   ```

   and observe the generated output in the first terminal. It should look similar to this:

   ```
   2017-01-25T16:57:48.553596179-06:00 container create 30eb63 ...
   2017-01-25T16:57:48.556718161-06:00 container attach 30eb63 ...
   2017-01-25T16:57:48.698190608-06:00 network connect de1b2b ...
   2017-01-25T16:57:49.062631155-06:00 container start 30eb63 ...
   2017-01-25T16:57:49.065552570-06:00 container resize 30eb63 ...
   2017-01-25T16:57:49.164526268-06:00 container die 30eb63 ...
   2017-01-25T16:57:49.613422740-06:00 network disconnect de1b2b ...
   2017-01-25T16:57:49.815845051-06:00 container destroy 30eb63 ...
   ```

3. If you don't like the format of the output then we can use the `--format` parameter to define our own format in the form of a Go template. Stop the events watch on your first terminal with CTRL+C, and try this:

   ```
   $ docker system events --format '--> {{.Type}}-{{.Action}}'
   ```

   now the output looks a little bit less cluttered when we run our alpine container on the second terminal as above.

