Lab Exercise: Managing the Container Lifecycle

Preamble

This lab exercise provides some simple, basic container management examples, using the Docker CLI and its commands. It also makes use of the `docker attach` and `docker exec` commands, which aid in interacting with containers.

Wherever you need to provide the container's ID as an argument to a Docker CLI command, remember you can use as few of the 64 hex digits as would make the container uniquely identifiable to the Docker daemon.

Step 1 – run a new container

The first step in the exercise is to run a brand new container using the `docker run` command. We'll base it on the latest ubuntu image available, and provide a simple command for the container to execute.

Start a command shell, and execute the following command :

```
$ docker run ubuntu:latest echo hello from my container
```

The first thing you'll notice is that the image '`ubuntu:latest`' doesn't exist in your local repository, so Docker has to 'pull' the image from the Docker Hub. It downloads each image layer as a compressed tar archive, and then extracts the data into the Docker runtime location. Once the image has been stored in the local repository, the container is created and started, and you will see the output `hello from my container`. You could equally well have referred to the image simply as '`ubuntu`', as Docker assumes the tag '`latest`' if no tag is specified.

For something that is supposed to be quite lean and fast, that took a lot of work! But, now that the image is in the local repository on the Docker host, if you execute the command again to run a new container, the command will be significantly faster. Try it.

Step 2 – run a container in detached mode, then stop and remove

Our first container was very simple. Let's prepare another container that will endure longer than the previous example. This time we'll execute a simple loop, but we'll run the container in detached mode. We'll also specify that the container is interactive (`stdin` is attached), and provide a pseudo-TTY. Note that command line arguments specified using short options can be combined, i.e. `-it`). Start the container by executing the following command:

```
$ docker run -d -it ubuntu:latest sh -c \
"while true; do sleep 5; date; done"
```

The command returns immediately, and provides the 64 digit ID of the container, which we'll need to enable us to attach to the container. Attach to the container, by executing the following command (substituting `<container>` with the ID of the container):

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

You should now be able to see the scrolling output of the looping shell script.

We can pause the operation of the container by executing the following command in a separate command shell (substituting `<container>` with the ID of the container):

```
$ docker pause <container>
```

Having satisfied yourself that the container is no longer echoing the date to `stdout` every 5 seconds, resume the container's operation by executing the following command (substituting `<container>` with the ID of the container):

```
$ docker unpause <container>
```

Let's detach from the container again, by typing `CTL-P` followed by `CTL-Q` in the terminal where we're attached to the container.

We've left our container running in detached mode, and we now need to stop and remove the container. We can do this by executing the following command (substituting `<container>` with the ID of the container):

```
$ docker rm $(docker stop <container>)
```

This compound command first uses `docker stop` to halt the container's execution, which also returns the container's ID, which we used as an environment variable for the argument to the `docker rm` command, which removed the container from the set of the daemon's registered containers.

Step 3 – create a new container

Now we'll create a new container, but won't start it immediately. We will run a `bash` command shell in our container, give it a hostname (of your choice), and make the root filesystem read-only. Execute the following command in order to create the container ready for future execution (substituting `<hostname>` for your preference):

```
$ docker create -it -h <hostname> --read-only ubuntu:latest /bin/bash
```

The `docker create` command will return the ID of the container that it has created, and we can view its configuration by viewing the JSON object created for it in Docker's runtime data directory (substituting `<container>` with the full ID of the container):

```
$ sudo cat /var/lib/docker/containers/<container>/config.json | python \
-m json.tool | less
```

Similar output can be viewed for the host specific configuration data by substituting `config.json` with `hostconfig.json` in the command above.

Now that we've seen that our container is configured ready to start, we can invoke it, using the following command (substituting `<container>` with the ID of the container):

```
$ docker start -ai <container>
```

Docker will start the container and return a `bash` prompt, which will be configured with the hostname we specified with the `docker create` command. If we attempt to create a file within our container, we will receive an error message because we have created our container's filesystem read-only:

```
root@skut:/# touch /tmp/file
touch: cannot touch '/tmp/file': Read-only file system
```

We'll now run `top` from our `bash` command shell in the container, which will give us a view of the processes running inside the container, along with the resources they are using (make a note of the PID of the `top` process), and we'll leave it running whilst we detach from the container by typing `CTL-P CTL-Q`. If we wanted to terminate the container's `top` process, how many different ways can you think of for achieving this?

We will use the `docker exec` command to run the `kill` command inside the container. Execute the following command (substituting `<container>` with the ID of the container, and <pid> with the PID of the `top` process):

```
$ docker exec <container> kill <pid>
```

When this command returns, get a process listing of the container's processes by executing the following command (substituting `<container>` with the ID of the container):

```
$ docker exec <container> ps ax
```

The output of this command should show that just our original `bash` command shell and the `ps` command are the only processes running; the `top` process was terminated successfully.

<u>Step 4 – kill the container</u>

Finally, our container has served its purpose, so we will use the `docker kill` command to terminate it. Execute the following command (substituting `<container>` with the ID of the container):

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

Docker returns the ID of the container that was killed successfully.