Docker Bootcamp

CCT

09/03/2018 (v0.2)

Sponsored by :

Contents

A brief history of Docker and containers

What is Docker?

Docker is a self-sufficient runtime for linux containers. Docker is an engine in which containers can be run in isolation and in which portability between host systems is guaranteed.

What does Docker offer?

Docker have built the tools to enable developers and operators to build and deploy applications with ease.

Simplicity

Docker makes powerful tools for application creation and orchestration, accessible to everyone

Openness

Built with open source technology and a modular design makes it easy to integrate into your existing environment.

Independence

Docker creates a separation of concerns between developers and IT and between applications and infrastructure to unlock innovation.

Underlying technologies

CGroups (Control Groups)

CGroups was originally developed by a couple of engineers at Google and made available in linux kernel version 2.6.24. CGroups limits, accounts for and isolates the resource usage (CPU, memory, disk I/O, network, etc.) of a collection of processes.

- *Resource limitation:* groups can be set to not exceed a configured memory limit, which also includes the file system cache
- *Prioritization:* some groups may get a larger share of CPU utilization or disk I/O throughput
- *Accounting:* measures how much resources certain systems use, what may be used, for example, for billing purposes
- *Control:* freezing the groups of processes, their checkpointing and restarting

Namespaces

While namespaces are not part of the CGroups project they are similar in that they isolate processes. Groups of processes are separated such that they cannot "see" resources in other groups. For example, a PID namespace provides a separate enumeration of process identifiers within each namespace. Also available are mount, UTS, network and SysV IPC namespaces.

Union File System

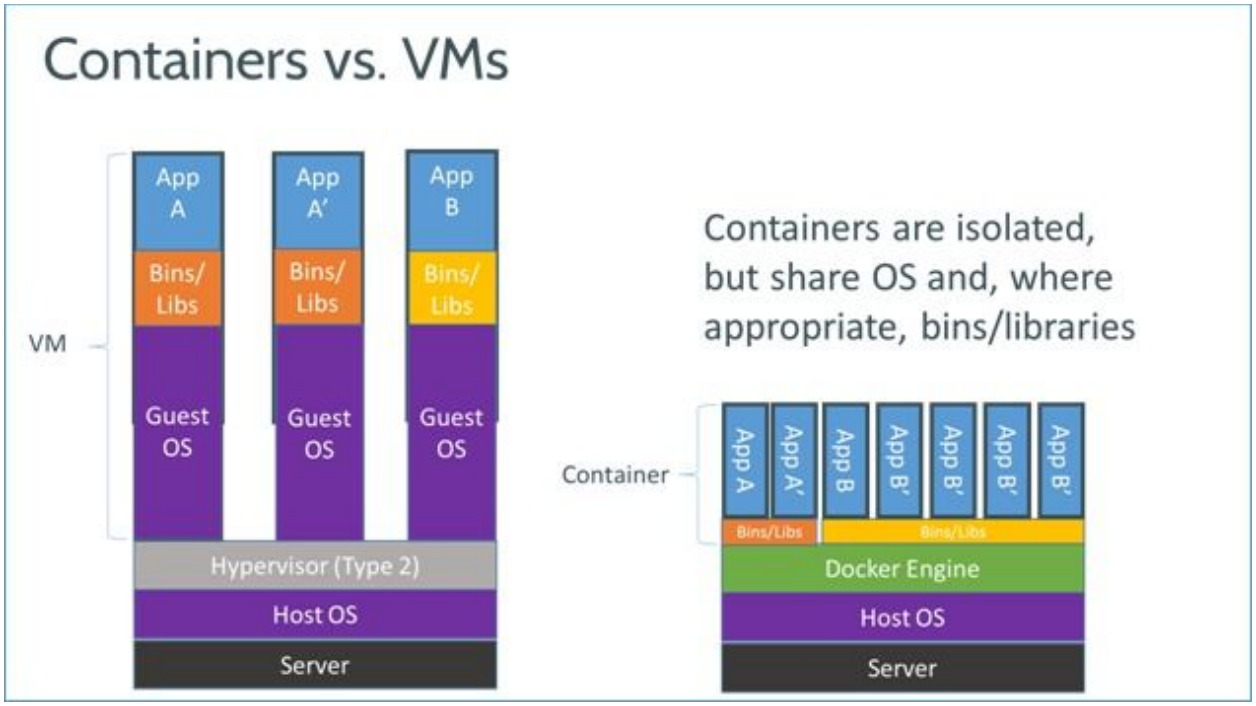
A union file system allows files and directories of separate file systems, known as branches, to be transparently overlaid, forming a single coherent file system.
The different branches may be both *read-only* and *read-write* file systems, so that writes to the virtual, merged copy are directed to a specific real file system. This allows a file system to appear as writable, but without actually allowing writes to change the file system, also known as copy-on-write.

Where did Docker come from?

Docker was started as an internal project at a PAAS called dotCloud. Docker was used to manage containers and dotCloud released the first open source version of Docker in March 2013.

As of October 17th, 2016, the project had almost 36,000 GitHub stars, over 10,000 forks, and over 1500 contributors.

Differences between Containers and VMs



Containers are similar to VMs in that they allow you to partition one physical computer into multiple small isolated partitions (called VMs or containers). The difference is in technique used for such partitioning.

Some of the major differences between VMs and containers, as well as their consequences, are outlined below.

Single kernel concept

Containers technology uses a single-kernel approach. There is only one single OS kernel running, and on top of that there are multiple isolated

instances of user-space programs. This approach is more lightweight than VM. The consequences are:

1. Waiving the need to run multiple OS kernels results in higher density of containers (compared to VMs)
2. Software stack that lies in between an application and the hardware is much thinner, this means higher performance of containers (compared to VMs)

Networking

An operating system deployed inside VM assumes it it is running on top of the real hardware, therefore VM needs to emulate a real physical network card (NIC). All the traffic should go through both the real NIC and the virtual NIC. This creates some noticeable overhead. In case of containers, no virtual NIC is needed (naturally, there can be no hardware drivers inside a container). Instead, a kernel is letting a container use a network interface, which may or may not be directly attached to a real network card. The difference from VM case is there is no hardware emulation of any kind.

Memory Management

In a VM every guest operating system kernel "thinks" it is managing some hardware, including memory (RAM). It performs memory allocation, accounting and so on.
In contrast, containers' memory is managed by a single entity -- the kernel. Therefore:

1. there is no need for virtual memory ballooning;
2. sharing memory pages between containers is trivial;
3. there is no need to preallocate the memory for a container when it starts -- memory is allocated on demand;
4. swapping is as fast as on a usual non-virtualized system.

File system

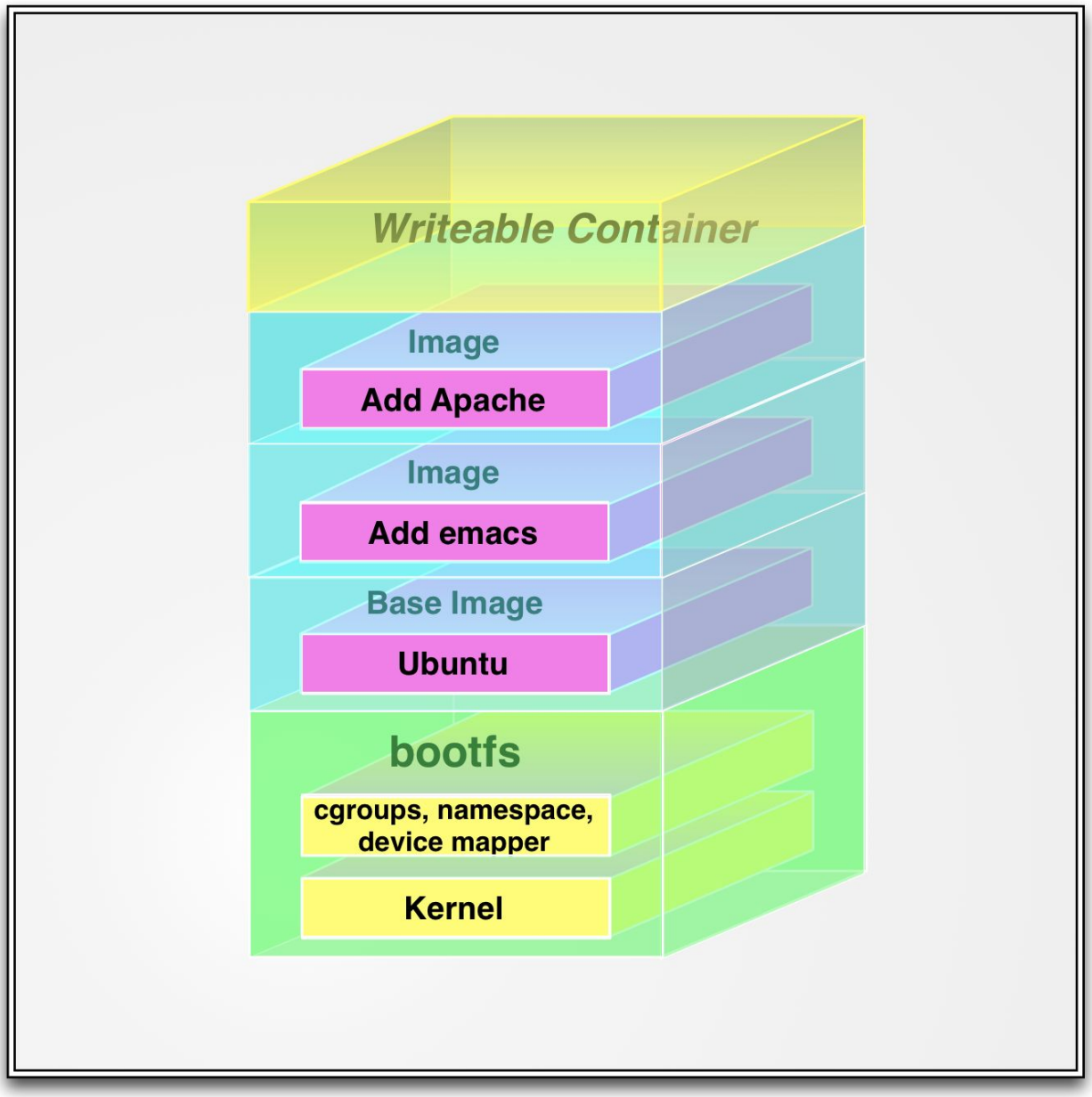
From file system point of view, a container is just a chroot() environment. In other words, a container file system root is merely a directory on the host system (usually /vz/root/$CTID/, under which one can find usual directories like /etc, /lib, /bin etc.). The consequences are:

1. there is no need for a separate block device, hard drive partition or filesystem-in-a-file setup
2. host system administrator can see all the containers' files
3. containers backup/restore is trivial
4. there is no I/O overhead (for VMs it can be as high as 1.5x to 3x, especially for small requests)
5. mass deployment is easy

Anatomy of a container Image

Layers

A container image is made up of layers. This allows for incremental changes and reduces the size of new commits to a base image.



Caching

When docker creates a new image layer during the build process it assigns a changeset to the layer. When rebuilding an image, docker will check for a cached version of the layer before rerunning a build instruction. We will cover this in more detail after Exercise 2.

Common Docker options

Before we start building images we will look at some common docker options. Depending on your setup you may need to use “sudo” when running the docker command.

Note:
Throughout the course we will be using the alpine:3.6 image. Alpine Linux is a security-oriented, lightweight Linux distribution based on musl libc and busybox. You can use most Linux distros as your base image. For a list of official repositories check out :
https://docs.docker.com/docker-hub/official_repos/

Ok let’s get started. To download the alpine image just run :

```
docker pull alpine:3.6
```

To view the image details run :

```
docker images
```

Example output :

```
alpine              3.6                 77144d8c6bdc        7 weeks ago
```

Each container image includes lots of useful metadata. You can view this metadata by running :

```
docker inspect alpine:3.6
```

Ok we have our image downloaded. Now let's start a container using our alpine:3.6 image. You don't need to worry about what the following command is doing as we will explain this in the next section :

```
docker run --name helloworld -d alpine:3.6 /bin/sh -c "while true;do echo "helloworld"; sleep 10;done"
```

The full container ID will appear in the terminal. Docker will accept the name of the container, in this case "helloworld" or the shortened version of the ID in the following commands.

What just happened? We started a container based on the alpine:3.6 image and ran a shell inside the container. It's important to realise that the container is only running as long as there is at least 1 process running inside the container. In the helloworld example the container will continue to run and output "helloworld" every 10 seconds before exiting. This may cause some confusion initially but is actually very important to understand this behaviour.

To view the output of the helloworld container run :

```
docker logs helloworld
```

To follow the logs :

```
docker logs -f helloworld
```

Let's stop our container :

```
docker stop helloworld
```

Let’s remove our container :

```
docker rm helloworld
```

Great!! You have just run your first container.

Let’s start another container and run some more advanced commands.

```
docker run --name demo -d alpine:3.6 /bin/sh -c "while true;do echo “Just a demo”; sleep 600;done"
```

List containers

List running containers :

```
docker ps
```

List running containers by id :

```
docker ps -q
```

List all containers

Lists all stopped and running containers :

```
docker ps -a
```

List all containers by id :

```
docker ps -aq
```

Inspect a container

Inspect the configuration of the container :

```
docker inspect demo
```

Review the output from docker inspect. In order to extract information from the inspect output you can use the --format option.

Example : Get IP address of “demo” container
docker inspect --format '{{ .NetworkSettings.IPAddress }}' demo

Stop and remove a container

Stop the demo container:
docker stop demo

Remove the demo container:
docker rm demo

Remove all stopped containers:
docker rm $(docker ps --filter "status=exited" -aq)

Note: Containers must be stopped before they can be removed.

To automatically remove a container after it has exited you can add the --rm option.

For example :

docker run --rm --name demo -t alpine:3.6 /bin/sh -c "echo Just a demo"

List Images

List all images stored locally
```
docker images
```

List image by ID
```
docker images -q
```

List image by full ID
```
docker images --no-trunc -q
```

List all untagged images
```
docker images --filter "dangling=true"
```

View the history of an image
```
docker history <image id>
```

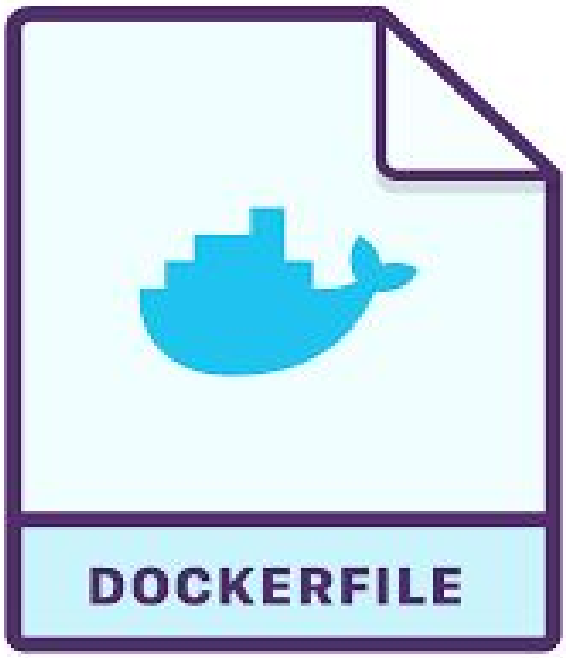
Remove an image

Remove image by ID
```
docker rmi <image id>
```

Remove all untagged images
```
docker rmi $(docker images -q --filter "dangling=true")
```

Note:
When an image is being accessed by a running container you will not be able to remove the image.

Building your first container image



Building your first image

This section will cover some of the most common instructions used in a Dockerfile. We will build a simple image to begin with and then add some more advanced build instructions. When you have mastered the art of image building then check out “Multi Stage Builds”. They are really useful and will keep the image size small.

What is a Dockerfile?

A Dockerfile is a set of instructions that docker runs sequentially. Each instruction in the Dockerfile corresponds to a new image layer.

The Dockerfile **must** contain the FROM instruction.

If any of the instructions fail or return a non-zero exit code then the build will exit.

In most cases, it's best to put each Dockerfile in an empty directory, and then add only the files needed for building that Dockerfile to that directory. When we run the docker build command the contents of the directory in which it is run is compressed. If you run docker build from your root directory then docker will attempt to tar the whole root directory.

To further speed up the build, you can exclude files and directories by adding a .dockerignore file to the same directory. You may want to exclude a large source code repository from a build but make it available to your containers as a shared volume. We will cover this later in the Shared Volumes section.

Exercise 1 :

For each exercise we will create a new directory under ~/bootcamp :

```
mkdir -p ~/bootcamp/exercise1
```

Warning : Avoid using your root directory, /, as the root of the source repository. If you create a Dockerfile in the root directory and attempt to build then docker will attempt to compress the entire contents of the root directory.

Create a Dockerfile inside the ~/bootcamp/exercise1 directory using vi/vim :

```
FROM alpine:3.6
MAINTAINER forename surname "email address"
ENTRYPOINT echo "Welcome to the Docker Bootcamp!! Have fun!"
```

Build an image from this Dockerfile and name it bootcamp :

```
docker build -t bootcamp .
```

This command will look for a Dockerfile in the current directory.

When the image has built successfully you will see output like :
Successfully built <image id>

Congratulations, you have built your first image!!!

Now let's start a container using our new image :

```
docker run bootcamp
```

You should see the following message :

Welcome to the Docker Bootcamp!! Have fun!

Congratulations. You have started a container based on the bootcamp image.

Question Time :

What TAG was assigned to the bootcamp image?
How many layers does the bootcamp image have?

A full list of build instructions can be found here :
https://docs.docker.com/reference/builder/

We will look at these in Exercise 4.

Build options

A full list of docker build options can be found here :
http://docs.docker.com/engine/reference/commandline/build/

We are going to look at --tag (-t shorthand) and --no-cache in more detail.

Tagging images

In the previous exercise we built an image and ran a container. Using the --tag option we can give our image a more human friendly name. The tag is made up of 2 parts : <repository>/<image>:<tag>

When we tag an image with just a repository name then Docker will check if that repository is unique and will set the tag to “latest” by default.

The next 2 commands will result in an image named bootcamp:latest :

```
docker build -t bootcamp:latest .
docker build -t bootcamp .
```

Exercise 2 :

Using the same Dockerfile from Exercise 1, build an image called bootcamp:latest and then change the tag to bootcamp:2

Start a container using the tagged image.

HINT : docker tag allows you to add a new tag to an image.

Caching

Story Time : “Building a Graphite container image for the first time.” Image caching is incredibly useful but it’s important to understand how it works to keep the image builds fast.

In Exercise 3 we will build a new image that contains the “htop” command.

Exercise 3 :

```
mkdir -p ~/bootcamp/exercise3
```

Create a Dockerfile inside the ~/bootcamp/exercise3 directory :

```
FROM alpine:3.6
MAINTAINER <your name in here> "<your email in here>"

RUN apk add --update
RUN apk add htop
RUN rm /var/cache/apk/*
ENTRYPOINT htop
```

Before we build this image let’s explain some of the commands ::

apk : Alpine Package manager. Similar to apt-get or yum install.
htop: Similar to the “top” command but for humans. This example is not optimized but we will cover this later.

Ok let’s build the image.

```
docker build -t htop .
```

The first time we build this image each command will be executed and each layer is given an unique id.

Sample Output :

```
Step 1 : FROM alpine:3.6
 ---> 4e38e38c8ce0
Step 2 : MAINTAINER <your name in here> "<your email in here>"
 ---> Running in 1158b6a95f9e
 ---> 2963116ff46b
Removing intermediate container 1158b6a95f9e
Step 3 : RUN apk add --update
 ---> Running in d89c3d5fe90f
fetch http://dl-cdn.alpinelinux.org/alpine/v3.6/main/x86_64/APKINDEX.tar.gz
fetch http://dl-cdn.alpinelinux.org/alpine/v3.6/community/x86_64/APKINDEX.tar.gz
OK: 5 MiB in 11 packages
 ---> 17a5dd235816
Removing intermediate container d89c3d5fe90f
Step 4 : RUN apk add htop
 ---> Running in 286ae0a79e98
(1/4) Installing ncurses-terminfo-base (6.0-r7)
(2/4) Installing ncurses-terminfo (6.0-r7)
(3/4) Installing ncurses-libs (6.0-r7)
(4/4) Installing htop (2.0.1-r0)
Executing busybox-1.24.2-r9.trigger
OK: 12 MiB in 15 packages
 ---> 9ea8c6090cba
Removing intermediate container 286ae0a79e98
Step 5 : RUN rm /var/cache/apk/*
 ---> Running in e94f10cc7fe8
 ---> aac92d08bef5
Removing intermediate container e94f10cc7fe8
Step 6 : ENTRYPOINT htop
 ---> Running in edafe2441761
 ---> b1471d530966
Removing intermediate container edafe2441761
Successfully built b1471d530966
```

When we build the image a second time, docker will use the cache as much as possible.

```
docker build -t htop .
```

Output from second build :

```
Sending build context to Docker daemon 2.048 kB
Step 1 : FROM alpine:3.6
 ---> 4e38e38c8ce0
Step 2 : MAINTAINER <your name in here> "<your email in here>"
 ---> Using cache
 ---> 2963116ff46b
Step 3 : RUN apk add --update
 ---> Using cache
 ---> 17a5dd235816
Step 4 : RUN apk add htop
 ---> Using cache
 ---> 9ea8c6090cba
Step 5 : RUN rm /var/cache/apk/*
 ---> Using cache
 ---> aac92d08bef5
Step 6 : ENTRYPOINT htop
 ---> Using cache
 ---> b1471d530966
```

The first build took 3.859 seconds. The second build took 0.070s. When writing your Dockerfile try and keep the most static commands at the top of the Dockerfile in order to use the cache as much as possible. For example if you have an instruction to setup some directories, users or environment variables then keeping these at the top of the Dockerfile will increase the odds of using the cache at build time.

Start the htop container : docker run -it htop

Caching is fantastic...mostly.

There is a potential pitfall to using caching. If we have a cached version of “RUN apk add --update” and we use this repeatedly then we are not pulling in the latest packages available in the package repository. While we may know that new package versions are available, docker will continue to use

the cached version. This is worth bearing in mind. To make sure the cache is not used we can take 2 approaches :

1. **Use the --no-cache build option**
   When building the image using “docker build” we can pass in the --no-cache option and this will ignore any previously cached layers. This will extend the build time but will also ensure that you are always pulling in the latest dependencies/source.

2. **Break the cache deliberately in the Dockerfile**
   There are lots of ways to trick docker into invalidating the cache. A quick google will return lots of posts and blogs on this topic. A common trick is to add && true after the initial RUN apt-get update && true.

More Dockerfile syntax

So far we have touched on the FROM, MAINTAINER, RUN and ENTRYPOINT build instructions. The MAINTAINER instruction is no longer a required instruction and is being replaced by LABEL but can be discussed later. In the next exercise we will look at some more build instructions and make our Dockerfile a bit more useful.

Exercise 4 :

Using this list of build instructions :
https://docs.docker.com/reference/builder/

Create a Dockerfile that will ADD a script called “hello.sh” into the container image. The contents of hello.sh are below :

```
#!/usr/bin/env sh
echo "Hello there, my name is ${NAME}"
```

Your Dockerfile should contain the NAME variable and this script should run automatically when the container is started.

HINT : You can set an environment variable in the Dockerfile using :

```
ENV NAME Bob
```

When the container runs this NAME variable will be used by the hello.sh script. The full solution can be found in the Appendix.

Ports

Ports are incredibly important. Without exposing the correct ports your application will be unreachable.

Exposing a port

Running a service inside a container often requires a port to be exposed to the host. If our container is running a webserver then we may EXPOSE 80 or if our container is running a database we may EXPOSE 3306. We can expose the port in the Dockerfile or when we start a container.

To expose port 80 in the Dockerfile we use this syntax :
EXPOSE 80

When starting a container we use the -p option to expose a specific port or -P to expose all ports that are explicitly defined in the Dockerfile.

For example :

To expose port 80 inside a container to localhost:80 :
docker run -d -p 127.0.0.1:80:80 webserver

To expose port 80 inside a container to a dynamically assigned port :
docker run -d -p 127.0.0.1::80 webserver

Exercise 5 :

Update the Dockerfile from Exercise 4 to include an exposed port 80.
Rebuild your image and start a container.
Verify that the port has been exposed by using docker ps and docker inspect.

Note : More details on exposing ports can be found here :
https://docs.docker.com/engine/reference/run/
https://docs.docker.com/userguide/dockerlinks/

Dockerfile optimization

Two of the core benefits of using a Dockerfile is that it is portable and reproducible. We can pass a Dockerfile around teams and everyone can build the same image. Simple Dockerfiles may only contain a few lines of

instructions but as the Dockerfile becomes more complex we may want to optimise the Dockerfile.

Chaining instructions

This fairly common trick will allow us to trim the number of layers.
If we combine two instructions we avoid making another layer so we're not storing intermediate (and maybe useless) states. However this could make the Dockerfile less readable.

Example :

```
FROM ubuntu:16.04
MAINTAINER <your name in here> "<your email in here>"
RUN apt-get update &&
apt-get install -y wget build-essential python python-dev python-pip &&
ADD mysource.tar /opt &&
tar xvzf /opt/mysource.tar
WORKDIR /data
```

Build smart and reusable Dockerfiles

Our last example wasn't a "smart" Dockerfile. Indeed, remember that each layer is an image that can be used by other Dockerfiles. If we just keep chaining our commands the way we just did, there will be some redundancy between images.
For instance, if, with our current Dockerfile, another image needs all the python packages we just installed, it would have to completely re-install them, whereas if we had a Dockerfile like the following one it wouldn't have to do so :

```
FROM ubuntu:14.04
MAINTAINER <your name in here> "<your email in here>"
RUN apt-get update &&
apt-get install -y wget build-essential python python-dev python-pip
RUN wget http://nodejs.org/dist/node-latest.tar.gz &&
```

```
tar xvzf node-latest.tar.gz &&
cd node-v* &&
WORKDIR /data
```

By adding a RUN just before the wget instruction we are creating an extra layer which is the ubuntu base image with all the python packages installed. This layer could be used by other Dockerfiles in the future and will substantially reduce the build process.

Container runtime

In this section we will look at more advanced runtime options beyond creating, starting and stopping containers. Containers are incredibly fast and flexible. We will look at how the lightweight nature of containers allows us to isolate dependencies and through container linking, create a more modular development environment.

Runtime options

Before we start spinning up containers it is important to look at some of the most common run options.

```
docker run -it alpine:3.6 /bin/sh
docker run -t alpine:3.6 /bin/sh
docker run -d alpine:3.6 /bin/sh
```

The “-i” option allows us to interact with the container. The “-t” option takes us to the terminal of the container and “-d” will daemonize the container. We will try each of these in the upcoming exercises.

Docker “run” creates and starts a container. Docker version 1.3.* introduced a “create” command that allows you to create a container but start it at a later time. This gives us better control over the containers. We will use docker “create” in a later exercise.

Exercise 6 :

In this exercise we will build an Nginx webserver container and host a simple HTML page.

This exercise will require 2 files; Dockerfile and nginx.conf.

HINT : The full solution can be found in the Appendix.

Firstly build an image called bootcamp:webserver.
Then start a container called “webserver” using the -d and -P options.

Verify that you can access the html page from the browser. HINT : To find the port that the container is exposing, use the “docker port” command.

Alternatively, test using curl:
curl 0.0.0.0:<port>

Cleanup container :
docker rm -f webserver

Debugging a running container

Docker provides a number of useful options to debug a running container. When we start a container it may not act exactly how we expect it to. Perhaps a process fails to start or dependencies are missing. To debug what has gone wrong we can use the following options :

docker top <container>
Shows the processes running inside a container.

docker stats <container>
Shows the memory, cpu and network I/O statistics for a container.

docker logs -f <container>
Tails the stdout from the container.

docker diff <container>
Shows the filesystem changes made in the containers. Example output from our webserver container :

Example :

docker diff webserver
C /run
A /run/apache2.pid
C /var
C /var/log
C /var/log/apache2
C /var/log/apache2/error.log

There are 3 events that are listed in the diff:

1. A - Add
2. D - Delete

3. C - Change

docker exec -it <container> /bin/bash
Logs you into a running container. This is for debugging and we don't encourage using this to make changes to a running container.

docker cp <container>:<filename> .
Copy a file from the container back to the host for inspection. Example :
docker cp web1:/var/log/apache2/error.log .

Container management

You can perform a number of actions on a running container :

docker pause <container>
docker unpause <container>
docker stop <container>
docker start <container>
docker rm <container>

Try these commands on your webserver container.

When changes have been made inside a container we can commit these changes to an image.

docker commit <container id> <image name>

This is incredibly useful when debugging and you want to incrementally commit changes to your base image.

Exercise 7 :

Start another container based on the bootcamp:webserver image.

Login to the running container using “docker exec” and update the index.html

Exit the container.

Stop the container and save the container ID to the bootcamp:webserver image using commit :
https://docs.docker.com/engine/reference/commandline/commit/

Now start a new container based on bootcamp:webserver and verify that the change you made to the index.html is available.

Data Volumes

A really useful feature in docker is the ability to share volumes between the host and containers. This allows us to share configuration, logs, secrets from a central location into one or more containers. In order to share a directory from the host into a container we use the “-v” option at run time.

Note : Be careful that you share the correct host directory.

Example :

```
mkdir -p ~/bootcamp/demo
cd ~/bootcamp/demo
```

Create a file :

```
touch demo
```

Start a container with a shared volume :

```
docker run -it -v ${PWD}:/var/tmp alpine:3.6 /bin/sh
```

Verify that the demo exists under /var/tmp inside the container.

** CAREFUL NOW **

You can remove the demo file but be wary that if you share a home directory as a volume or any other important directory you may think you are deleting a file in the container but are actually removing it from the host.

Since Docker 1.10 you can now create Data Volumes using the command :

```
docker volume create --name webserver_data
```

This newly created volume “webserver_data” can be attached to a single or multiple containers at run time :

```
docker run -d -v webserver_data:/www/data -P bootcamp:webserver
```

This is the prefered method of managing data volumes. The actual physical location of the data stored in the volume can be found by running :

```
docker volume inspect webserver_data
```

Example Output :


```
[
        {
        "Name": "webserver_data",
        "Driver": "local",
        "Mountpoint": "/var/lib/docker/volumes/webserver_data/_data",
        "Labels": {},
        "Scope": "local"
        }
]
```

Exercise 8 :

Based on what we have covered so far, create 3 nginx webserver containers which share the same index.html. Start the containers and verify that any changes made to index.html appear in each of the webservers.

Extras :

Now try pausing the containers and reload the webpage.
Now unpause the container.

Linking containers

This section is a little out of date but it's worth knowing why links exist and the alternatives available.

Links allow containers to discover each other and securely transfer information about one container to another container. When you set up a link, you create a conduit between a source container and a recipient container. The recipient can then access select data about the source. To create a link, you use the `--link` option.

The naming convention used is important when linking containers. To establish links, Docker relies on the names of your containers. Providing meaningful names for containers which you intend to link serves two purposes :

1. It can be useful to name containers that do specific functions in a way that makes it easier for you to remember them, for example naming a container containing a web application "web".

2. It provides Docker with a reference point that allows it to refer to other containers, for example, you can specify to link the container "web" to container "db".

Container linking is useful for a couple of containers but when we want many containers to be able to communicate we use the "docker network" command. We will cover this in Exercise 10.

Exercise 9 :

We are going to create two containers. One called “web” and one called “db”. For the purpose of brevity we won't be setting up a database and website. The purpose of this exercise is to show what is being exposed inside a linked container.

Start a dummy database container called “db” with port 3306 exposed :

```
docker run -d --name db -p 3306 alpine:3.6 /bin/sh -c \
"while true; do sleep 30;echo 'I am a Database';done"
```

Now start a dummy website container called “web” and link to the “db” container.

```
docker run -d --name web --link db:db -p 80 alpine:3.6 /bin/sh -c \
"while true; do sleep 30;echo 'I am a webserver';done"
```

Login to the “web” container and check which environment variables have been set using the “env” command.

```
docker exec -it web /bin/sh
```

Example Output :

```
DB_PORT=tcp://172.17.0.4:3306
HOSTNAME=8b7c410bada9
SHLVL=1
DB_PORT_3306_TCP=tcp://172.17.0.4:3306
HOME=/root
DB_NAME=/web/db
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
DB_PORT_3306_TCP_ADDR=172.17.0.4
PWD=/
DB_PORT_3306_TCP_PORT=3306
```

DB_PORT_3306_TCP_PROTO=tcp

Cleanup containers :

docker rm -f web db

Exercise 10 :

This exercise will create a number of containers and attach them to the same network. For more details on docker network checkout :

https://docs.docker.com/engine/reference/commandline/network_create

Ok let's start 5 containers and attach them to our "demo_network".

First we create the network :

```
docker network create demo_network
```

Verify the network was created successfully :

```
docker network ls
```

Now let's start 5 webservers and attach them to the demo_network. We will use the bootcamp:webserver image from Exercise 6 to create 5 containers.

Repeat this step 5 times :

```
docker run -d --network=demo_network -P bootcamp:webserver
```

Example Output :

```
c31948cf9407        bootcamp:webserver        "nginx -g 'daemon off"   14 seconds ago
        Up 13 seconds       0.0.0.0:32779->80/tcp                peaceful_goodall
62a5e169ca0e        bootcamp:webserver        "nginx -g 'daemon off"   15 seconds ago
        Up 14 seconds       0.0.0.0:32778->80/tcp                lonely_bassi
```

```
2cce690d139b        bootcamp:webserver        "nginx -g 'daemon off"   16 seconds ago
        Up 15 seconds       0.0.0.0:32777->80/tcp                hopeful_booth
0aadb322919c        bootcamp:webserver        "nginx -g 'daemon off"   17 seconds ago
        Up 16 seconds       0.0.0.0:32776->80/tcp                furious_minsky
b2cf14dae355        bootcamp:webserver        "nginx -g 'daemon off"   19 seconds ago
        Up 19 seconds       0.0.0.0:32775->80/tcp                adoring_bose
```

Ok we have 5 containers running and attached to the same network. Let's login to a container and verify they are accessible on the same network.

Using the container id listed above : **c31948cf9407**

Note: This container id will be different on your system.

Login to the container and verify that you can ping other containers by name and by IP address :

```
docker exec -it c31948cf9407 /bin/sh
```

```
# ping 62a5e169ca0e
PING 62a5e169ca0e (172.25.0.5): 56 data bytes
64 bytes from 172.25.0.5: seq=0 ttl=64 time=0.059 ms
```

```
/ # ping 172.25.0.5
PING 172.25.0.5 (172.25.0.5): 56 data bytes
64 bytes from 172.25.0.5: seq=0 ttl=64 time=0.068 ms
```

Exit the container and verify that there are 5 containers connected to the demo_network :

```
docker network inspect demo_network
```

That was the last exercise of the Bootcamp. The next few pages will cover some of the other tools in the Docker Ecosystem.

Before we continue you might want to remove any containers running on your system :

```
docker rm -f $(docker ps -q)
```

Orchestration

One of the most common complaints about Docker has been the lack of native orchestration tooling. Docker is great when running containers on one host but when we want to manage hundreds or thousands of containers across multiple hosts then we need better tooling.

We will not have time to cover tools such as Mesos or Kubernetes but there are lots of online tutorials that are worth looking at. We are going to look briefly at docker-compose and docker swarm mode.

Docker Compose : https://docs.docker.com/compose/

Compose is a tool that allows you to manage more complex containerised environments. A user can define a multi container environment within a docker-compose.yml file.

```
web:
  build: .
  command: python app.py
  links:
   - db
  ports:
   - "8000:8000"
db:
  image: mysql
```

In the example above, compose will build and start 2 containers and link them together. The “web” image will be built using a Dockerfile in the current working directory. The “db” image is already built and docker will use the local version of the mysql image.

Compose was formerly a tool called Fig which was bought by Docker in August 2014.

Compose is simple to install and supports most docker commands. It may not be suitable for large scale deployments but it is great for spinning up local development or test environments.

swarm mode: https://docs.docker.com/engine/swarm/

Swarm mode was released with Docker 1.12. It is built into native docker cli and enables the user to create and join existing swarms of docker engines.

Example :

Create a new Swarm :

```
docker swarm init --advertise-addr 192.168.99.121
```

Add more Docker Engines/Hosts to the Swarm :

```
docker swarm join --token SWMTKN-1-3pu6hshu2 192.168.99.121:2377
```

It's that simple. Docker swarm mode is incredibly powerful and enables the user to setup quite complex and large scale clusters with just a few commands. We will cover Swarm Mode in our Intermediate Bootcamp.

Docker Hub and Docker Registry

Docker Hub is the public repository for storing images. You can sign up for a free account that will give you one private repository and access to automated builds.

Docker Registry is commonly hosted internally and used behind the company firewall. Docker released a “Trusted Registry” in June 2015 which is a hosted version of the registry in which you pay a subscription.

Review

At this point we can perform the following :

- **Create a docker image using a Dockerfile**
- **Create and start one or more containers**
- **Manage the lifecycle of a container**
- **Expose ports, share volumes and link containers**
- **Create a network and attach containers to the network**

What we didn't cover but will be included in the Intermediate Bootcamp :

- **Docker volumes**
- **Docker plugins**
- **Docker Swarm mode and Services**

The Docker documentation is very good, the community slack channel is very active and we have a monthly Docker Meetup.

Some folks to follow :

@tomwillfixit
@DockerDublin
@Demonware

Appendix

Exercise 4 Solution

Dockerfile :

```
FROM alpine:3.6
MAINTAINER <your name in here> "<your email in here>"
ENV NAME Tom
ADD hello.sh /tmp/hello.sh
RUN chmod 755 /tmp/hello.sh

ENTRYPOINT ["/tmp/hello.sh"]
```

Build image :

```
docker build -t exercise:4 .
```

Start container :

```
docker run exercise:4
```

Start container and provide a new NAME :

```
docker run -e NAME=Sharon exercise:4
```

Exercise 6 Solution :

You need 2 files; Dockerfile and nginx.conf

Dockerfile :

```
FROM alpine:3.6
MAINTAINER <your name in here> "<your email in here>"

RUN apk --update add curl nginx && \
        mkdir -p /var/log/nginx && \
```

```
        mkdir -p /run/nginx/ && \
        touch /var/log/nginx/access.log && \
        mkdir -p /tmp/nginx


#Add nginx config to image
ADD nginx.conf /etc/nginx/

#Download bootcamp.html
RUN curl https://tomwillfixit.com/bootcamp.html --create-dirs -o /www/index.html

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]
```

Nginx.conf :

```
user root;
worker_processes  1;

events {
        worker_connections  1024;
}

http {
        include           mime.types;
        default_type  application/octet-stream;

        server {
        listen    80;
        server_name  localhost;

        root /www;
        index index.html index.htm;

        location / {
        try_files $uri $uri/ /index.html;
        }

        }
}
```

Build the bootcamp:webserver image :

```
docker build -t bootcamp:webserver .
```

Start the container :

```
docker run -d --name webserver -P bootcamp:webserver
```

Check the container is running :

```
docker ps |grep webserver
```

Open a browser to localhost:80 and you have a webpage being hosted in a container using nginx.