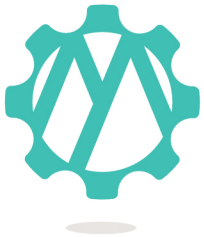
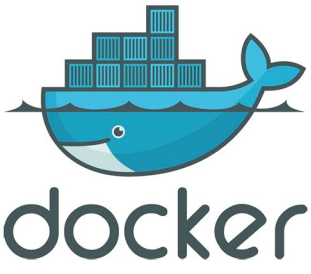


MECHANICAL ROCK

Intro To Docker



Chapter 1 - What is Docker?

Containers vs. Virtual Machines

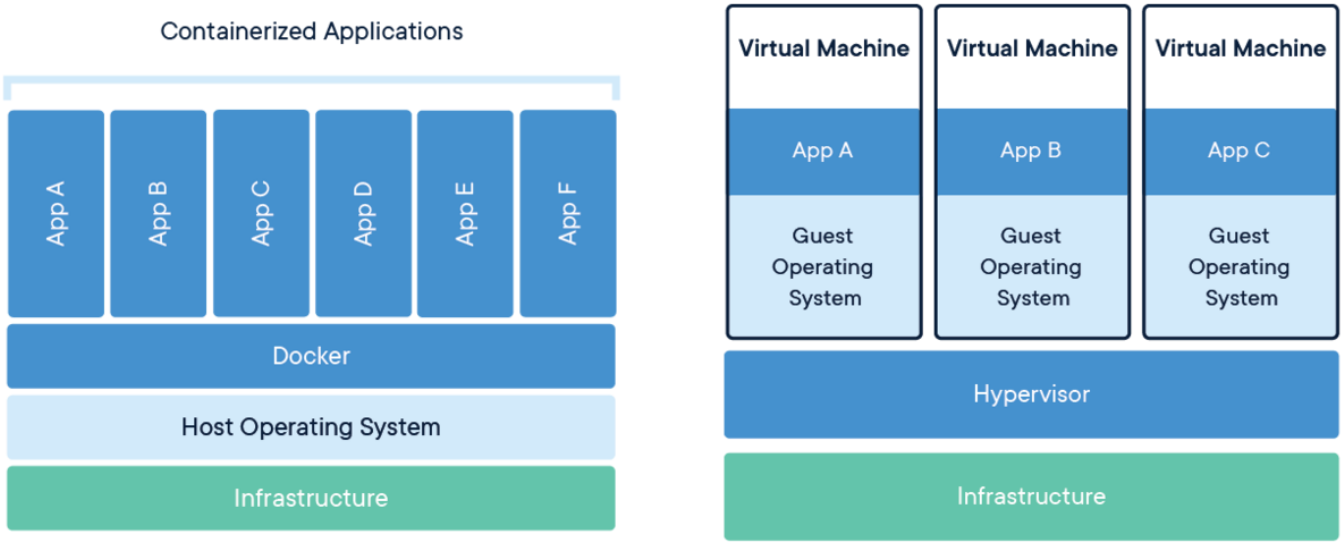
Virtual Machines

Virtual machines were created in order to:

- Get better hardware utilisation
- Application isolation
- Better failure tolerance

But they had drawbacks:

- Heavyweight, you're dragging around whole operating system
- Slow, starting an operating system from scratch
- Require specialised skills, application developers usually aren't experts in operating system automation



Containers

But what if we shared the operating system, and instead focused the isolation one level higher?

By using the "magic" of the Linux kernel, Docker containers were born. Instead of carting around an entire operating system, instead system calls are passed through. This means we can be ***fast***.

Although the kernel constructs that allow for containers were put there in part by Google in the early 2000's, the real trick that made Docker is the developer experience. The barrier to entry is much lower, without losing much of the power.

Containers in your Workplace

Discuss and write down how you think containers may help you and your organisation:

Chapter 2 - Running a Container

Basic Docker Terminology

Term	Definition
Image	Executable package containing everything to run an application
Container	A runtime instance of an image
Engine	The environment that allows for the creation and running of images and containers

Hello, World

You can't learn anything without doing 'Hello, World!'

Start the hello-world container:

```
docker run hello-world
```

Breaking It Down

After each section of the output, write what you think is happening:

```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
```

```
d1725b59e92d: Pull complete
Digest:
sha256:0add3ace90ecb4adbf7777e9aacf18357296e799f81cabc9fde470971e499788
Status: Downloaded newer image for hello-world:latest
```

```
Hello from Docker!
This message shows that your installation appears to be working correctly.
...
```

Chapter 3 - Docker Hub

The Power Of Community

Part of the power of Docker is the ecosystem that drives it.

> Standing on the shoulders of giants, *Isaac Newton*

Whether you're working in Java, Node.js, .Net or any other language, there'll be containers that do most of the hard work for you.

We'll be using a Node image later on as an example of how you this works.

Your Local Image Cache

```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
```

Whenever you build an image or pull one from a repository it is stored locally on your machine. Then, naturally, this cache is queried on all commands. This is where a lot of the speed of Docker comes from.

Because we're storing everything, all the time, this can start to take up significant amounts of room. There's a couple of convenience commands that will prove useful. We'll revisit them in the next chapter as they have other uses.

Command	Definition
`docker container prune`	Removes all the stopped containers
`docker image prune`	Removes all the images not required by containers

Searching Docker Hub

The biggest and default repository of Docker Images is at https://hub.docker.com/

Go there now and see what you can find for your favourite ecosystem.

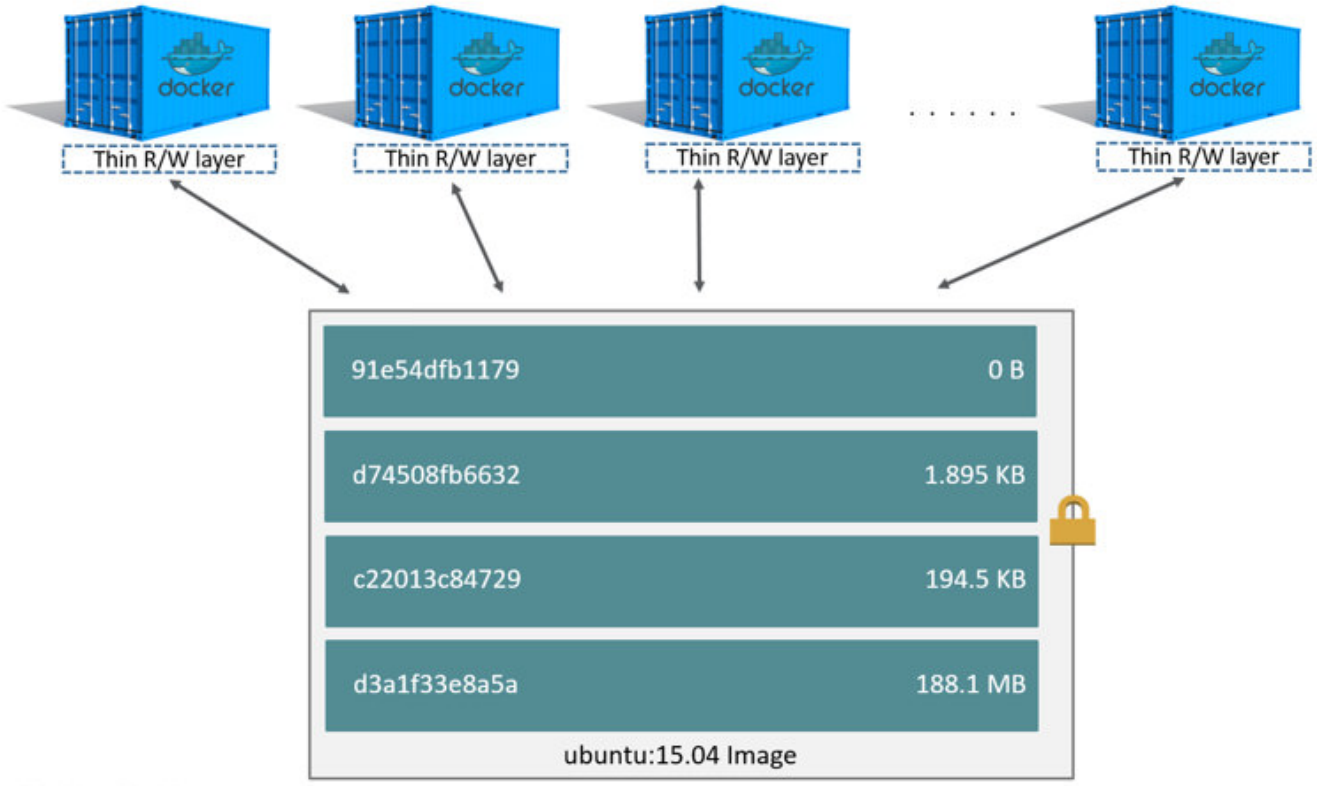
Chapter 4 - Layers

How Docker Images Work

Docker images work by stacking layers.

```
d1725b59e92d: Pull complete
Digest:
sha256:0add3ace90ecb4adbf7777e9aacf18357296e799f81cabc9fde470971e499788
Status: Downloaded newer image for hello-world:latest
```

The `hello-world` image we pulled down earlier had only one layer, but generally more useful images are made from 10 layers or more.



What Layers Get You

In the last chapter we discussed how we store images locally, but if we store two images that share a layer, we only store the layer once.

The main benefit is if we already have a layer, we don't have to rebuild it. That means, as we develop Docker images we only build the layers that have changed.

Getting Into A Running Container

Last time we ran a container, it outputted some text. But what if we want to go into one?

Start a container from the `alpine` image:

```
docker run -it alpine
```

You should be present with a prompt, you should be able to run commands like `ls` and `cd`

Drop out of the container:

```
exit
```

This will drop you out of the container back into the host shell.

What The 'i' And 't' Flags Do

The `-i` flag makes the container `interactive`. Effectively this means the `stdout` and `stdin` from your terminal are passed through to the container.

The `-t` flag allocates a `pseudo-tty` to the container, which is similar to how `ssh` works.

When you combine them, it's functionally similar to having `ssh`-ed onto a server.

Making Our Own Layers and Images

Start the container:

```
docker run -it alpine
```

Create a file:

```
touch myfile
```

Check the directory contents:

```
ls
```

Check there's now a file called `myfile` listed

Exit the container:

```
exit
```

List all the containers:

```
docker container ps -a
```

The top container will be the one you just exited, it is in a 'Stopped' state which we will cover in more detail later.

Copy the CONTAINER ID

Create a new Docker image:

```
docker commit <container id> myimage
```

Start a new container from the image:

```
docker run myimage ls
```

You should be able to see you file

Isn't There A Better Way?

Generally you won't use `docker commit`, but the command underlies Dockerfiles which are coming in Chapter 7. But understanding this mechanism is important for understanding how Dockerfiles work.

Chapter 5 - Tagging

Latest

A few times now, the word 'Latest' has been seen around. For referencing Docker images we generally use `Name:Tag`, e.g. `Node:8.11`. If you don't specify a tag, either when creating a container or referencing one, it'll by default be `latest`.

For working locally, leaving things tagged as `latest` is fine, however once you get to sharing, purposefully tagging your images becomes very important.

Looking At An Example

Looking at the official `Node` images, (https://hub.docker.com/_/node/), they use tagging to specify a variety of options.

First they specify the runtime version, e.g. `8.12.0`

Second they specify the underlying operating system, e.g. `jessie`, `alpine` or `slim`

`jessie` is a container optimised flavour of debian.

`alpine` is a super lightweight Linux kernel, that is used for size optimised images.

`slim` is a lighterweight version of `jessie`

I Thought Containers Removed The Operating System

Back in chapter one, we were saying that we got the benefits from containers by removing the operating system portion of a virtual machine. Now we're saying the you have to specify which operating system we want in the container.

In order to allow for applications to function in containers they still require a slimmed down kernel to function, although the majority of the `syscalls` are still passed through.

Chapter 6 - Persistence

Containers Are Designed To Be Ephemeral

Although we can start stopped containers, and pick up where we left off, that's an anti-pattern. Ideally we want repeatable operations on repeatable infrastructure.

Most worthwhile things are performing operations on data, and there are 3 ways to handle data volumes on containers.

Every time you run `docker run` it's a brand new container based on the `image`

The Container Lifecycle

Looking back at our `hello-world` container from earlier, **let's run it again:**

```
docker run --name myhelloworld hello-world
```

Now if we check the list of containers:

```
docker container ps
```
You'll notice it isn't there.

But if we get the list of all containers:

```
docker container ps -a
```

We can now see it:

```
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS                      PORTS               NAMES
f9850a139a45        hello-world         "/hello"                 32
seconds ago      Exited (0) 31 seconds ago
myhelloworld
```

We can see that it `Exited(0)`

This means the container is 'Stopped', stopped containers still exists on your disk, but are not using any other system resources. By default they will persist until you remove them.

Now compare that with our `alpine` containers:

```
docker run --name myalpine -dit alpine
```

Checking to see running containers:

```
docker container ps
```

We can see it's still running:

```
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES
a71762793e40        alpine              "/bin/sh"           5 seconds ago
Up 4 seconds                            myalpine
```

What's the difference? Write down your ideas of why these two containers appear to act differently.

Naming Containers

In the last section we started containers specifying their names, `myhelloworld` and `myalpine`. What happens if we try to start a container with the same name?

```
docker run --name myalpine -dit alpine
```

We get an error.

Well it makes sense we can't have two running containers of the same name. So let's stop the running container:

```
docker container stop myalpine
```

Now if we try to run again:

```
docker run --name myalpine -dit alpine
```

We still get an error.

The rule is we can't overwrite a container, if you read the error message it says we need to remove the container to be able to reuse the name. So let's do that:

```
docker container rm myalpine
```

What scenarios can you think of where this might be a problem? Can you find in `docker run --help` a way of managing this?

Chapter 7 - Volumes

Volume Mounts

Volume mounts are for sharing data between multiple containers, either in parallel or on sequential executions. Now let's run an example:

Create a volume:

```
docker volume create my-vol
```

Check that it exists:

```
docker volume ls
```

Start a container that uses the volume:

```
docker run -it -v my-vol:/app alpine
```
The `-v` flag is used to specify a mount, in the format `<volume name>:<mount point>`

Once in the shell, create a file:

```
touch /app/index.js
```

Drop out of the container:

```
exit
```

Start a new container, using the same volume:

```
docker run -it -v my-vol:/app alpine
```

Show that the file is still there:

```
ls /app
```

Bind Mounts

Although you can, with a large amount of effort, get the data in a `docker volume` back out into your host machine, there is a much easier way. Instead of specifying a volume name, you can specify a folder on the host machine:

First create a file:

```
touch hostfile
```

Start a container mounting the current directory:

```
docker run -it -v ${PWD}:/app alpine
```

`${PWD}` will resolve to the absolute path of the current directory

Show that the file is now available in the container:

```
ls /app
```

Tmpfs Mounts

There's a third type of data volume you can use, but only on Linux. As the name would suggest this mount in transitory, it's tied to the lifecycle of the container. I.e. Once the container is stopped, the mount will no longer exist. As this mount is handled purely in memory it's used for storing sensitive information that you don't want to be persisted on either the host or container filesystems.

Chapter 8 - Dockerfiles

Repeatable Images

Rather than building up images by hand, we can write `Dockerfiles`. These files are step by step instructions for building an image which are passed to the Docker Engine.

Let's create a quick html file to serve via Nginx:

> Create a file called mypage.html

> Add the following content

```
<html>
<body>
<p>Hello, Intro to Docker!</p>
</body>
</html>
```

> Create a new file called `Dockerfile`.

The FROM Statement

The first line in a Dockerfile is the `FROM` statement. This specifies the image to build on top of.

> Add `FROM nginx:1.15.3` as the first line of your Dockerfile

The COPY Statement

To add files to an image you can use the `COPY` statement.

> Append the line `COPY ./mypage.html /usr/share/nginx/html/mypage.html`

The ADD Statement

A second way to add files is with the `ADD` statement.
`COPY` is preferred but add has one extra bit of functionality, you can specify a web address to download something from.

> e.g. `ADD https://www.docker.com /usr/share/nginx/html/docker.html`

The RUN Statement

In order to run commands, such as installing packages, to a Docker image you use the `RUN` statement.

> Append the line `RUN apt-get update`

The ENV Statement

If you need environment variables as part of an image you can specify them with the `ENV` statement

> Append the line `ENV USER <your name here>`

The EXPOSE Statement

If you need to access the container via the network you need to specify what ports will be open. This is done with the `EXPOSE` statement.

> Append the line `EXPOSE 80`

The ENTRYPOINT Statement

The `ENTRYPOINT` statement is used to specify fairly stable default commands for a container. For example the `maven` Docker images use:
`ENTRYPOINT ["/usr/local/bin/mvn-entrypoint.sh"]`
To execute the referenced script when the container starts.

> Append the line `ENTRYPOINT ["nginx"]`

The CMD Statement

The `CMD` statement is used to specify the less stable default commands for a container. Often it is flags appended to the `ENTRYPOINT`

In our case we're going to pass a couple of options to our `nginx` call from our `ENTRYPOINT`

> Append the line `CMD ["-g", "daemon off;"]`

Building An Image From A Dockerfile

To build an image we run the `docker build` command

> Run `docker build -t mynginx .`

In this example we have passed `-t mynginx` which names the image so we can reference it easier.

The `.` at the end of the line sets the current directory as what is known as the `build context` for the image.

Build Contexts

To understand why build contexts are important, we need to understand how `docker build` functions.

- First it copies everything in the build context across to a folder in `/tmp`
- Then it performs the steps in the Dockerfile, performing a `docker commit` after every line
- Then it names and tags the image, making it available for use

Therefore, the build context are the files which are copied over to the temporary folder, and define what files are accessible for the statements in the `Dockerfile`

Running This Image

In order to run this image, we need to talk about networking, which is the topic of the next chapter.

Chapter 9 - Networking

Running Mynginx Image

Taking our `mynginx` image, the easiest way to access the server is to port-forward as we execute the `docker run` command.

Start the container from the last chapter:

```
docker run -d -p 80:80 mynginx
```

With the `-p` flag we have set so that port 80 on `localhost` is forward to port 80 on the container. Now we should be able to access `localhost` and see the file we put there earlier as part of the image build.

The `-d` flag says to run the container in `detached` mode, effectively in the background so we don't tie it to the current terminal session.

Get the content from the server:

```
curl localhost/index.html
```

Container To Container Networking

If you have two containers that need to talk to each other, at first it can seem complicated. The containers run without knowing they're a container, and you can't address your way back to your host machine easily.

To show that it doesn't work:

Start two alpine containers:

```
docker run -dit --name alpine1 alpine
docker run -dit --name alpine2 alpine
```

Attach to the first container:

```
docker attach alpine1
```

Ping the second container:

```
ping alpine2
```

Making It Work

Thankfully there's a construct called a docker network. This allows for simple intercontainer communication, by allowing addressing by container name.

To create a docker network, we can do it with a single command

Create a docker network:

```
docker network create mynetwork
```

Start two alpine containers:

```
docker run -dit --net mynetwork --name alpine3 alpine
docker run -dit --net mynetwork --name alpine4 alpine4
```

Attach to the first container:

```
docker attach alpine3
```

Ping the second container:

```
ping alpine4
```

You should now see responses from `alpine4`

Chapter 10 - What Next

Container Patterns

Developer Machine Homogenisation

You can effectively have your only dependency on a development machine as having Docker installed. You can have all the runtimes operating in containers, meaning that you're onboarding becomes:

```
Install Docker
Git clone the repo
Run a script which spins up the containers, and runs all the tests
```

Sidecar Containers

Let's say you're developing against a Postgres database, now with Docker you can stand up a local Postgres instance in seconds allowing on developer machine integration testing.

Mutli-Stage Docker Builds

A fairly recent addition to the Docker toolkit is the concept of multi-stage Docker builds. As a rule, you want to optimise to the smallest image size possible. It makes it faster to build, to run and to share.

Say you have a Java application, to run the build you will require additional dependencies and you will need the JDK installed. However if you're building an image to be run, then you would only want the JRE installed, with minimal dependencies.

By specifying a second `FROM statement` you can copy the build artifacts from the layers earlier in the Dockerfile.

Challenges

1. More Use Cases

   Back in Chapter 1, you put down what use cases you could see for containers at work. Are there any extra now that spring to mind?

2. Layer Invalidation

   Go back to the chapter on Dockerfiles, can you figure out what would cause the layer per statement to be invalidated? E.g. With the `RUN` statement, how does the engine know whether to run that again, or use the layer created previously?

3. Environment Variables in `Docker Build`

   Environment variables are not accessible as part of the `RUN` statements in a Dockerfile, they only take effect once the image is run. Can you find the statement you would need to have access to configurable variables during build. E.g. if you have credentials which need injecting to download packages.

4. Docker Compose

   Go and research Docker Compose, and what the use cases are. See if you can set up a basic web-server and database application.